

Inhaltsverzeichnis

| | | |
|-------|--|----|
| 1. | Einführung in die Implementation..... | 9 |
| 1.1 | Grobstruktur von HU-Prolog..... | 10 |
| 1.2 | Einige Bemerkungen zum Programmierstil..... | 13 |
| 2. | Abstrakte Datentypen in HU-Prolog..... | 14 |
| 2.1 | Der Datentyp STRING..... | 15 |
| 2.1.1 | Die Komponenten des Datentyps STRING..... | 15 |
| 2.1.2 | Low-Level Verwaltung..... | 16 |
| 2.2 | Der Datentyp ATOM..... | 17 |
| 2.2.1 | Die Komponenten des Datentyps ATOM..... | 17 |
| 2.2.2 | Low-Level Verwaltung..... | 21 |
| 2.2.3 | Die Atomtabelle..... | 24 |
| 2.2.4 | High-Level Verwaltung..... | 28 |
| 2.3 | Der Datentyp TERM..... | 31 |
| 2.3.1 | Die Komponenten des Datentyps TERM..... | 31 |
| 2.3.2 | Low-Level Verwaltung..... | 34 |
| 2.3.3 | High-Level Verwaltung..... | 36 |
| 2.4 | Der Datentyp CLAUSE..... | 40 |
| 2.4.1 | Komponenten des Datentyps CLAUSE..... | 40 |
| 2.4.2 | Funktionen zur Verwaltung der Klauseln..... | 42 |
| 2.5 | Der Datentyp TRAIL..... | 43 |
| 2.5.1 | Die Komponenten des Datentyp TRAIL..... | 43 |
| 2.5.2 | Funktionen zur Verwaltung des Datentyp TRAIL..... | 43 |
| 2.6 | Der Datentyp ENV..... | 44 |
| 2.6.1 | Die Komponenten des Datentyps ENV..... | 44 |
| 2.6.2 | Funktionen zur Verwaltung des Datentyps ENV..... | 46 |
| 3. | Der Interpreterkern..... | 47 |
| 3.1 | Die Unifizierung in Prolog..... | 47 |
| 3.2 | Der Ableitungsmechanismus von HU-Prolog..... | 51 |
| 3.2.1 | Das Grundgerüst von EXECUTE()..... | 52 |
| 3.2.2 | Eine vollständige Version..... | 56 |
| 3.2.3 | Die endgültige Version..... | 61 |
| 4. | Realisierung der Build-In Prädikate..... | 64 |
| 4.1 | Schnittstellengestaltung..... | 64 |
| 4.2 | Fehlerbehandlung..... | 64 |
| 4.3 | Die Funktion CallEvalPred()..... | 66 |
| 4.4 | Beispiele für die Realisierung von Build- In's..... | 69 |
| 4.4.1 | Prädikate zur Termklassifizierung..... | 69 |
| 4.4.2 | Prädikate zur Termanalyse und Syntese..... | 71 |

| | | |
|-----------|---|----|
| 4.4.3 | Termvergleich..... | 75 |
| 4.4.4 | Listenverarbeitung..... | 77 |
| 4.4.5 | Das Prädikat $:=/2$ | 80 |
| 4.5 | Das File user.c..... | 82 |
| 5. | Globale Steuerung, Toplevel..... | 85 |
| 6. | Schlußbemerkungen..... | 88 |
| 7. | Literaturverzeichnis..... | 90 |
| Anhang A: | Realisierungen der abstrakten Datentypen..... | 91 |
| Anhang B: | Schnittstelle zum Betriebssystem..... | 96 |
| Anhang C: | HU-Prolog Quelltexte..... | 97 |

1. Einführung in die Implementation

Prolog-Systeme wurden in den letzten Jahren viele geschaffen. Dabei haben sich einige Grundprinzipien der Implementation herausgebildet, welche kurz diskutiert werden sollen.

Ein bei allen Programmiersprachen mit rekursiven Ausdrucksmitteln auftretenden Problem ist die Verwaltung der lokalen Variablen und der Informationen zur Programmsteuerung. Ähnlich wie in klassischen Programmiersprachen wird auch in Prolog für diese Daten ein Segment im (sogenannten lokalen) Stack angelegt. Der Unterschied zu klassischen prozeduralen Sprachen besteht darin, daß dieser Stack nur während des Backtrackings zurückgesetzt werden kann, da eine Prolog-Prozedur erst dann als beendet betrachtet werden kann. (Bestimmte Optimierungstechniken ermöglichen allerdings das Rücksetzen des lokalen Stacks auch zu anderen Zeitpunkten)

Ein Prologtypisches Problem sind die bei der Unifizierung entstehenden Variablenbindungen. Da sie im Falle des Backtrackings wieder gelöst werden müssen, werden sie im sogenannten Trail-Stack vermerkt. Eigentlich könnte man dafür auch den oben beschriebenen lokalen Stack nutzen, aus Effizienzgründen ist es aber günstiger einen extra Stack zu führen.

Eine bei der Implementation von Prolog zu treffende Entscheidung ist die Wahl zwischen einer sogenannten structure-sharing und einer structure-copying Variante. Was ist darunter zu verstehen?

Da Prolog einen Sprache mit rekursiven Ausdrucksmöglichkeiten ist, ist es vor der Benutzung von in der Datenbasis stehenden Variablen notwendig, eine Instanz der Variablen im lokalen Stack anzulegen. Dabei tritt nun das Problem auf, wie der Verweis von den Termen auf die sie enthaltenden Variablen erfolgt. Dafür sind in Prolog zwei Wege bekannt. Die klassische Methode ist dabei das sogenannte "structure-sharing". Zu jedem Term gehören dabei zwei Verweise. Einer auf die Struktur des zu repräsentierenden Termes, der andere auf den Block der Variablen dieses Termes. In der Termstruktur sind die Variablen als Offsets zum Variablenblock vermerkt.

Der Vorteil dieser Methode besteht in einer Speicherplatzersparnis für große Terme, da die Termstruktur nur einmal abgespeichert werden muß, und verschiedene Instanzen des Termes sich nur durch verschiedene Variablenblöcke unterscheiden. Nachteilig ist, daß für jeden Term zwei Verweise geführt werden müssen.

Einführung in die Implementation

Die zweite Methode ist das sogenannte "structure-copying". Dabei wird für jede neue Instanz eines Termes der gesamte Term von der Datenbasis in den Stack kopiert. Der Vorteil dieser Variante ist, daß zu jedem Term nur ein Verweis in den Stack gehört. Nachteilig ist, daß das notwendige Kopieren mehr Zeit benötigt.

Wie man sieht, haben beide Methoden ihre Vor- und Nachteile. Während ältere Prologimplementationen (z.B C-Prolog) eine reine structure-sharing Philosophie verfolgen, gehen viele neuere Implementation von einer gemischten Strategie aus. So auch HU-Prolog.

Für den Programmcode wird die structure-sharing Methode benutzt, das heißt, daß für jede aufgerufenden Klausel ein Variablenblock im lokalen Stack angelegt wird. Da immer nur eine Klausel aktiv ist, gibt es Systemweit nur einen globalen Verweis auf diesen Variablenblock.
Für Daten wird dagegen nach der Methode structure-copying verfahren. Wenn eine neue Instanz eines Termes benötigt wird, so wird der gesamte Term neu angelegt. Dies geschieht im sogenannten globalen Stack.

Auf den ersten Blick erscheint es unmotiviert, hierzu einen zweiten Stack zu benutzen. Dies ist aber notwendig, um die oben bereits erwähnten Optimierungen durchführen zu können.

HU-Prolog geht hier sogar noch etwas weiter. Alle Terme, also auch die lokalen Variablen werden im globalen Stack angelegt. Im lokalen Stack stehen nur noch die für die rekursive Verarbeitung benötigten Verwaltungsinformationen; dieser Stack trägt deshalb den Namen Environment-Stack. Dieses Verfahren hat sich für eine einfache Implementation als am effektivsten erwiesen.

Nun zum HU-Prolog-System selbst.

1.1 Grobstruktur von HU-Prolog

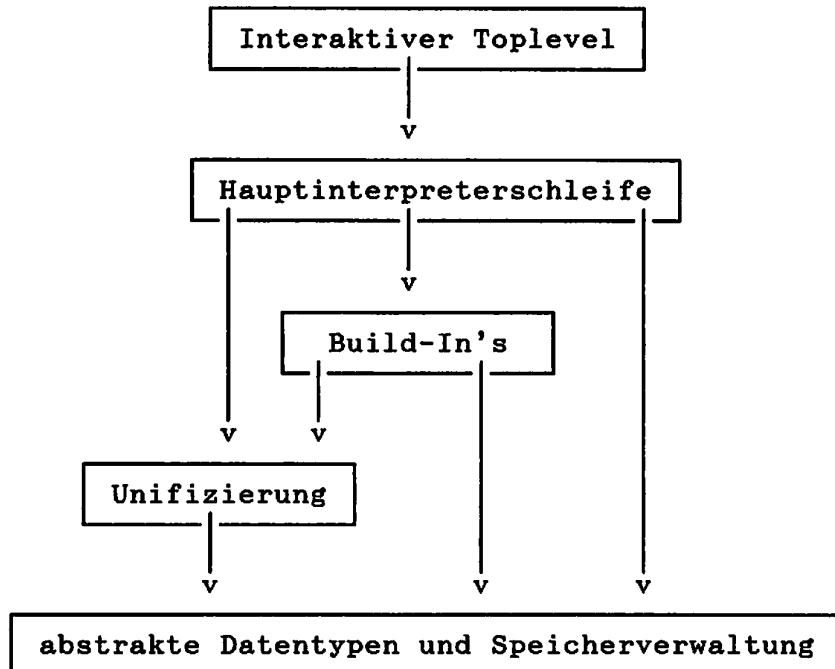
Das HU-Prologsystem besteht logisch aus folgende fünf wesentlichen Komponenten:

- den abstrakten Datentypen und den auf ihnen operierenden Funktionen;
- der Unifizierung als Grundoperation jedes Prologsystems;
- dem den eigentlichen Inferenzmechanismus realisierenden Interpreterkern;

Einführung in die Implementation

- dem dem Nutzer gegenüberstehenden interaktiven Toplevel;
- und
- der Realisierung der Build-In Prädikate

Diese stehen folgendermaßen in Beziehung:



Ein Pfeil gibt an, daß eine Komponente die andere benutzt

Diese fünf Komponenten werden in folgenden Quelltextfiles realisiert.

Abstrakte Datentypen und Speicherverwaltung:

| | |
|------------------|---|
| atoms.h | Deklaration aller vordefinierten Prolog-Atome |
| manager.h | Definition der Größen wichtiger Speicherbereiche |
| types.h | Deklaration wichtiger abstrakter Datentypen und des Zugriffs auf diese. Verschieden Varianten sind im Anhang A aufgeführt |
| atoms.c | Verwaltung der Atomtabelle |

Einführung in die Implementation

| | |
|-------------------|--|
| <i>database.c</i> | Verwaltung der HU-Prolog Datenbasis |
| <i>manager.c</i> | Low-level Verwaltung der abstrakten Datentypen |
| <i>memory.c</i> | Speicherplatzreservierung für abstrakte Datentypen |

Unifizierung:

uni.c

Build-In Prädikate:

| | |
|-----------------|---|
| <i>files.h</i> | Beschreibung der Filestruktur der I/O Komponente |
| <i>window.h</i> | Schnittstelle zum benutzten Windowsystem |
| <i>arith.c</i> | Realisierung der Arithmetik sowie der globalen Variablen |
| <i>eval.c</i> | Sprungverteiler für Build-In Prädikate sowie Implementation von nicht anders einzuordnenden Build-In Prädikaten |
| <i>io.c</i> | Realisierung der Ein/Ausgabe |
| <i>misc.c</i> | Verschiedene Hilfsfunktionen |
| <i>read.c</i> | Scanner und Parser für Prologterme |
| <i>sys.c</i> | Betriebssystemabhängige Funktionen |
| <i>user.c</i> | Nutzerdefinierte Prologprädikate; C-Interface |
| <i>win.c</i> | Realisierung des systemspezifischen Windowsystems |
| <i>write.c</i> | externe Darstellung von Prologtermen |

Hauptinterpreterschleife:

exec.c

Interaktiver Toplevel:

prolog.c

Einführung in die Implementation

1.2 Einige Bemerkungen zum Programmierstil

HU-Prolog wurde als ein portables System entwickelt, welches eine übersichtliche Struktur aufweist und deswegen (relativ) wartungsfreundlich ist.

Um die Portabilität zu unterstützen, wurde in der vorliegenden Implementation nur eine Teilmenge von C genutzt. Dies geschah mit dem Ziel, daß diese Teilmenge von allen (also auch den etwas älteren) C-Implementationen beherrscht wird. Das bedeutet zum Beispiel den Verzicht auf `struct` und `union` Komponenten als Funktionsparameter sowie auf `enum`-Aufzählungstypen. Des Weiteren wurde die Schnittstelle zur C-Bibliothek (im Anhang B beschrieben) so klein wie möglich gehalten.

Um die Übersichtlichkeit und Wartungsfreundlichkeit zu gewährleisten, wurde die konsequente Nutzung abstrakter Datentypen angestrebt. Darunter ist eine volkommende Trennung von Benutzung (Schnittstellenbeschreibung) und konkreter Implementation zu verstehen. Mit Hilfe des Makromechanismus des C-Präprozessors lassen sich diese abstrakten Datentypen auch effizient realisieren. Einen abstrakten Datentyp in HU-Prolog kann man sich grundsätzlich als eine Art Verweis auf eine Struktur vorstellen. Der Zugriff auf die einzelnen Komponenten der Struktur erfolgt über Makros. Durch eine Deklaration der Form:

`Abstrakter_Datentyp Variablename;`

erhält man also nur eine Variable, welche einen Verweis (einen Pointer) auf solch eine Struktur aufnehmen kann. Für die Struktur selbst wird kein Speicherplatz reserviert. Am besten wird das deutlich, wenn man sich mögliche konkrete Realisierungen der abstrakten Datentypen betrachtet. (siehe Anhang A)

Soweit zur Einleitung, es folgt die Beschreibung der wichtigsten abstrakten Datentypen.

2. Abstrakte Datentypen in HU-Prolog

Der folgende Abschnitt soll die grundlegende Struktur der von HU-Prolog benutzten Datentypen beschreiben.

In HU-Prolog gibt es sechs wesentliche Datentypen.

STRING: Der Datentyp **STRING** dient der Verwaltung der zu den Atomen gehörenden Zeichenketten.

ATOM: Atome sind in Prolog die elementaren Grundbausteine aus denen "alles" zusammengesetzt ist. In HU-Prolog ist ein Atom eindeutig durch seinen externen Namen (also die vom Nutzer eingegebene Zeichenkette) und durch seine "Stelligkeit" gekennzeichnet. Ein Atom in HU-Prolog ist also etwas, was in anderen Prolog-systemen mit dem Begriff Funktor bezeichnet wird.

TERM: Terme sind die grundlegende Repräsentationsform von Daten und Programmen in Prolog. Sie bilden eine Baumstruktur, deren einzelne Knoten durch Atome der Stellenzahlen 1 bis **MAXARITY**, und deren Blätter durch Atome der Stellenzahl 0, Zahlen oder ungebundene Variablen gebildet werden.

CLAUSE: Ein Prologprogramm besteht aus einer Menge von Klauseln, die aus syntaktischer Sicht Terme bilden. Der Datentyp **CLAUSE** realisiert die Verkettung dieser Klauseln und den Verweis auf die Terme, die die Klauseln repräsentieren.

TRAIL: Eine der wesentlichen Unterschiede von Prolog zu "klassischen" Programmiersprachen ist die Realisierung des Backtrackings als elementare Grundoperation. Während des Backtrackings wird der Abarbeitungsprozeß auf einen älteren Stand zurückgesetzt und mit einer neuen Alternative fortgesetzt. Dabei müssen natürlich alle inzwischen entstandenen Variablenbindungen von älteren Abarbeitungsvarianten wieder gelöst werden. Diese eventuell wieder zu lösenden Variablenbindungen werden im sogenannten Trail-Stack vermerkt.

ENV: In klassischen Programmiersprachen wird für jeden Aufruf einer Funktion ein Stacksegment angelegt, das neben den lokalen Variablen dieser Funktion noch Verwaltungsinformationen enthält. So ähnlich ist es auch in HU-Prolog. Während die lokalen Variablen einer Klausel im Term-Stack abgelegt werden, werden die vom Interpreter benötigten Verwaltungsinformationen in sogenannten Environments abgelegt.

Abstrakte Datentypen in HU-Prolog

Der Trail und die Environments sind in HU-Prolog als Stacks organisiert. Für Atome und Terme gibt es sowohl einen Stack, als auch eine Heap. Für Klauseln gibt es nur einen Heap. Der Grundgedanke dabei ist, daß alle Daten welche zum Aufbau der Prolog-Datenbasis mit beitragen (also das Programm), im Heap abgespeichert werden, und damit relativ statisch sind. In den Stacks werden dagegen die Daten verwaltet, die während der Abarbeitung eines Prologprogrammes entstehen bzw. zu dessen Verwaltung benötigt werden. Im gewissen Sinne gibt es also eine Analogie zwischen den Stacks in HU-Prolog und den Stacks in den klassischen Programmiersprachen. Das Rücksetzen der Stacks in Prolog erfolgt jedoch im Wesentlichen nur während des Backtrackings. Dabei wird praktisch ein Stück der Abarbeitung "vergessen", und mit einer neuen Alternative fortgesetzt.

Im folgenden soll der Aufbau der einzelnen Datentypen genauer erläutert werden. Die Beschreibung der Datentypen erfolgt durch Angabe der Makros, die auf die einzelnen Komponenten zugreifen. Diese Makros werden hier wie C-Funktionen deklariert. Ein in der Deklaration verwendeter Bezeichner *LVALUE* gibt an, daß das Macro einen L-Value liefert, also sowohl auf der linken als auch auf der rechten Seite von Zuweisungen stehen kann. Des Weiteren erfolgt die Schnittstellenbeschreibung der Basisfunktionen, die mit diesen Datentypen arbeiten.

2.1 Der Datentyp STRING

Der Datentyp *STRING* dient zur Abspeicherung von Zeichenketten, die dem Nutzer zugänglichen externen Namen der Prologprädikate repräsentieren. Aus Gründen der Einheitlichkeit der Darstellung der abstrakten Datentypen als auch der Mängel vieler C-Compiler ist der Datentyp *STRING* nicht identisch zu den in der Sprache C bekannten Strings.

2.1.1 Die Komponenten des Datentyps STRING

Der Datentyp *STRING* hat keine weiteren Komponenten. Ein vordefiniertes Element des Datentyps *STRING* ist die

Abstrakte Datentypen in HU-Prolog

Konstante *NIL_STRING*.

2.1.2 Low-Level Verwaltung

Zum Erzeugen bzw. Zugreifen auf Daten von Typ *STRING* gibt es folgende Funktionen:

STRING stackstring(s)
char *s;

erzeugt ein Datum vom Typ *STRING* an der Spitze des String-Stacks, und gibt ihm als Wert die C-Zeichenkette *s*.

STRING heapstring(s)
char *s;

erzeugt ein Datum vom Typ *STRING* im String-Heap, und gibt ihm als Wert die C-Zeichenkette *s*.

void init_string_stack()
STRING mark_string_stack(); und
void rel_string_stack(S)
 STRING S;

Die Funktion *init_string_stack()* initialisiert diesen Stack. Nach Abarbeitung dieser Funktion ist der Stringstack leer. Die Funktion *mark_string_stack()* liefert die Spitze des Stacks als Resultat. Mit *rel_string_stack(S)* wird der Stringstack auf den in *S* markiertet Wert zurückgesetzt. Danach ist *S* wieder das oberste Stackelement. Auf das durch *mark_string_stack()* gelieferte Datum vom Typ *STRING* sollte nur durch die Funktion *rel_string_stack()* zugegriffen werden.

char stringchar(S,i)
 STRING S;
 cardinal i;

liefert als Resultat das *i*-te Zeichen des Strings *S*. Das erste Zeichen der Zeichenkette erhält man mit *stringchar(S,0)*. Das Zeichen nach dem letzten Buchstaben des Strings ist das Zeichen '\0'.

*char *tempstring(S)*
 STRING S;

Konvertiert ein Datum vom Typ *STRING* in einen C-String. Um eine flexible Implementation dieser Funktion zu ermöglichen, ist folgendes zu beachten:

Abstrakte Datentypen in HU-Prolog

- Das Resultat von `tempstring()` kann ein Zeiger auf einen internen Puffer sein, der bei jedem Aufruf überschrieben wird.
- Das Resultat von `tempstring()` ist als "read-only" zu betrachten, es sollten also keine Modifikationen an diesem String vorgenommen werden.

2.2 Der Datentyp ATOM

2.2.1 Die Komponenten des Datentyps ATOM

Ein Datum vom Typ `ATOM` besteht in HU-Prolog aus folgenden Komponenten:

`LVALUE STRING atomstring(A)`
 `ATOM A;`

verweist auf eine Repräsentationsform der Zeichenkette, die dem Atom zugeordnet ist.

`LVALUE cardinal arity(A)`
 `ATOM A;`

bezeichnet die Stelligkeit des Atoms `A`; Dies ist eine natürliche Zahl im Bereich von 0 bis `MAXARITY`.

`LVALUE CLAUSE clause(A)`
 `ATOM A;`

ist der Beginn der Klauselkette der zum Atom `A` gehörenden Klauseln. (Der Datentyp `CLAUSE` wird weiter unten definiert) Wenn zu einem Atom keine Klauseln vorhanden sind, hat `clause(A)` den Wert `NIL_CLAUSE`.

`LVALUE ATOM nextatom(A)`
 `ATOM A;`

`LVALUE ATOM chainatom(A)`
 `ATOM A;`

beschreibt die Verkettung der Atome in einer zweidimensionale Atomtabelle untereinander. Diese Komponenten werden nur von den Funktionen zur Verwaltung der Atomtabelle genutzt. (Kapitel 2.2.3)

Die folgenden Komponenten des Datentyps `ATOM` sind Eigenschaften oder Attribute des oben spezifizierten Atoms. Es werden hier Funktionen zum Abfragen und Setzen dieser Attribute aufgeführt.

Abstrakte Datentypen in HU-Prolog

```
OCLASS oclass(A)
    ATOM A;
void setoclass(A,Value)
    ATOM A;
    OCLASS Value;
```

In Prolog können bekanntlich für die externe Darstellung von Termen Operatorschreibweisen genutzt werden. *oclass(A)* beschreibt ob, und wenn ja wie, das Atom *A* als Operator zu interpretieren ist. *setoclass(A,Value)* setzt dieses Attribut auf den entsprechenden Wert. Der Typ *OCLASS* ist dabei eine Aufzählung folgender Werte:

| | |
|-------------------------|---|
| <i>NONO</i> | das Atom wird nicht als Operator interpretiert; |
| <i>FXO, FYO</i> | das Atom wird als Prefix-Operator interpretiert; |
| <i>XFO, YFO</i> | das Atom wird als Postfix-Operator interpretiert; und |
| <i>XFXO, XFYO, YFXO</i> | das Atom wird als Infix-Operator interpretiert. |

Dabei bedeutet z.B. *FXO* einen Operator, der in Prolog mit dem Typ *fx* definiert werden würde.

Es ist darauf zu achten, daß die Stellenzahl des Atoms mit dem Typ des Operators in Übereinstimmung stehen muß, sonst kann es zu unerklärlichen Verhalten der Einleseroutinen kommen.

```
cardinal oprec(A)
    ATOM A;
void setoprec(A,Value)
    ATOM A;
    cardinal Value;
```

Auch dieses Attribut dient der Spezifikation des Atoms *A* als Operator. *oprec(A)* bestimmt den Vorrang den das Atom *A* als Operator hat. *setoprec(A,Value)* setzt diesen Wert für ein Atom *A* auf den Wert *Value*. Es können Werte von 0 bis *MAXPREC* angenommen werden. Aus Kompatibilitätsgründen zu anderen Prologsystemen sollte *MAXPREC* nicht kleiner als 1200 sein.

Abstrakte Datentypen in HU-Prolog

```
CLASS class(A)
    ATOM A;
void setclass(A,Value)
    ATOM A;
    CLASS Value;
```

Durch dieses Attribut wird eine Klasseneinteilung der Atome vorgenommen. Damit wird unterschieden, wie der Aufruf eines Prädikates mit diesem Atom als Hauptfunktor interpretiert wird. *CLASS* ist dabei ein Aufzählungstyp, der folgende Werte annehmen kann.

- NORMP** Ein normales Prologprädikat. Wenn ein Goal mit diesem Hauptfunktor abgearbeitet werden soll, so werden Klauseln dafür gesucht und zu neuen Zielen erklärt.
- EVALP** Ein "evaluable predicate". Es handelt sich hierbei um die eingebauten nicht-backtrackbaren Build-In Prädikate, deren Abarbeitung letztlich in den Aufruf von C-Funktionen mündet.
- BTEVALP** Ein "backtrackable EVALP". Für diese Build-In Prädikate ist ein Backtracking vorgesehen. (z.B. das Prädikat *clause/2*)
- GOTOP** Das Pseudoatom *GOTOT*. Näheres dazu im Kapitel über die interne Klauseldarstellung.(2.4.1)
- VARP** kennzeichnet die Pseudoatome *VART*, *UNBOUND* und *SKELT*. Näheres zur Bedeutung dieser Atome im Kapitel zum Datentyp *TERM*. (2.3.1)
- NUMBP** kennzeichnet die Pseudoatome, die als Zahlen interpretiert werden. Näheres zu diesen Atomen im Kapitel zum Datentyp *TERM*. (2.3.1)
- CUTP** klassifiziert die Atome, die als *!/0* interpretiert werden (unabhängig von ihrem eigentlichen Namen). In der jetzigen Implementation sind dies *!/0* und *'\$!'/0*. Das hier eine eigene Klasse definiert wurde dient lediglich der Effektivierung der Implementation des Cut's.(vgl. 3.2.2)
- ARITHP** Dient zur Sonderbehandlung der Symbolischen Arithmetik. Näheres dazu im Kapitel 4.3.5.

Abstrakte Datentypen in HU-Prolog

```
boolean private(A)
    ATOM A;
void setprivate(A)
    ATOM A;
void setnotprivate(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des `private()`-Attributes. Näheres dazu im Kapitel über den Aufbau und die Verwaltung der Atomtabelle.

```
boolean hide(A)
    ATOM A;
void sethide(A)
    ATOM A;
void setnothide(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des `hide()`-Attributes. Näheres dazu im Kapitel über den Aufbau und die Verwaltung der Atomtabelle.

```
boolean first(A)
    ATOM A;
void setfirst(A)
    ATOM A;
void setnotfirst(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des `first()`-Attributes. Näheres dazu im Kapitel über den Aufbau und die Verwaltung der Atomtabelle.

```
boolean system(A)
    ATOM A;
void setsystem(A)
    ATOM A;
void setnotsystem(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des `system()`-Attributes. Wenn dieses Attribut gesetzt ist, gilt das Atom als geschützt. Operationen wie das Löschen oder Hinzufügen von Klauseln zu diesem Atom werden von den entsprechenden Funktionen zurückgewiesen.

Abstrakte Datentypen in HU-Prolog

```
boolean spy(A)
    ATOM A;
void setspy(A)
    ATOM A;
void setnotspy(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des *spy()*- Attributes. Dieses Attribut wird zum Kennzeichnen der Atome verwendet, bei denen ein gezieltes Tracing erfolgen soll.

```
boolean ensure(A)
    ATOM A;
void setensure(A)
    ATOM A;
void setnotensure(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des *ensure()*- Attributes. Damit steuert das HU-Prolog Prädikat *ensure/3* seine Arbeit. Näheres dazu bei der Implementation des Prädikates *ensure/3* in den beiliegenden Quelltexten.

```
boolean rc(A)
    ATOM A;
void setrc(A)
    ATOM A;
void setnotrc(A)
    ATOM A;
```

Abfragen, Setzen bzw. Rücksetzen des *rc()*- Attributes. Dieses Attribut wird vom Prädikat *reconsult/1* verwendet. Näheres dazu bei der Implementation dieses Prädikates.

Eine vordefinierte Konstante des Typs *ATOM* ist *NIL_ATOM*.

2.2.2 Low-Level Verwaltung

Nachdem nun die Komponenten des Datentyps *ATOM* dargestellt wurden, folgen einige Funktionen zur Verwaltung dieses Datentyps. Wie schon erwähnt gibt es für Atome einen Stack und einen Heap. Normalerweise stehen alle für den Nutzer relevanten Atome im Heap. (Genauer sind dies alle Atome, die in der Prolog-Datenbasis vorkommen). Man könnte auch nur mit dem Heap auskommen. Allerdings wird für den Atom-Heap keine Speicherfreigabe unterstützt, da diese einen hohen Verwaltungsaufwand zur Laufzeit erfordern würde. Atome die im Heap stehen, können also nicht entfernt werden und beanspruchen Speicherplatz. Im Stack stehen deshalb alle

Abstrakte Datentypen in HU-Prolog

die Atome, die von Prologprädikaten wie `read/1` und `name/2` erzeugt wurden und nicht in der Prolog-Datenbasis vorkommen. Der Stack wird im Fall des Backtrackings zurückgesetzt. Dieser etwas komplizierte Mechanismus wurde implementiert, damit Programme wie

```
:-- repeat,  
    read(X),  
    transform(X,Y),  
    write(Y),  
    Y == end.
```

nicht zu einem Programmabruich wegen Speicherüberlauf im Atomheap führen.

Nach diesen Vorbemerkungen nun zu den einzelnen Funktionen:

ATOM heapatom()

liefert als Resultat ein neues Atom im Heap, während

ATOM stackatom()

ein neues Atom an der Spitze des Stacks liefert.

Um diesen Stack zu kontrollieren, gibt es die Funktionen

```
void init_atom_stack();  
ATOM mark_atom_stack(); und  
void rel_atom_stack(A)  
    ATOM A;
```

Die Funktion `init_atom_stack()` initialisiert den Atomstack. Nach Anwendung dieser Funktion ist der Stack leer. Die Funktion `mark_atom_stack()` liefert die Spitze des Stacks als Resultat. Mit `rel_atom_stack(A)` wird der Atomstack auf den in `A` markierten Wert zurückgesetzt. Das Atom `A` ist dann wieder oberstes Stackelement.

Oftmals ist es notwendig zu entscheiden, ob ein gegebenes Datum vom Typ `ATOM` sich im Stack oder im Heap befindet. Dazu dienen die beiden Klassifizierungsfunktionen

```
boolean is_heap_atom(A)  
    ATOM A;  
boolean is_stack_atom(A)  
    ATOM A;
```

Sie liefern einen Boolschen Wert, je nachdem ob sich `A` im Heap oder im Stack befindet. Dabei ist zu beachten, daß der Wert dieser Funktionen, angewandt auf das Atom `NIL_ATOM`, nicht definiert ist.

Abstrakte Datentypen in HU-Prolog

Desweiteren ist es notwendig, auf alle im Atomstack befindlichen Atome zugreifen zu können. Dafür gibt es die Funktion:

```
ATOM get_stack_atom(A)
    ATOM A;
```

Sie liefert zu einem gegebenen Stackatom *A* immer das nächste (in einer intern definierten Reihenfolge) Anfang und Ende dieser Ordnung ist das Atom *NIL_ATOM*. Folgendes kleines Beispiel mag die Benutzung dieser Funktion verdeutlichen:

```
ATOM A = NIL_ATOM;
while((A = get_stack_atom(A)) != NIL_ATOM)
{
    /*
    ** hier kann A ausgewertet werden, darf
    ** aber nicht verändert werden
    */
}

boolean norm_atom(A)
    ATOM A;
boolean func_atom(A)
    ATOM A;
```

Wie schon angedeutet wurde, werden einige vordefinierte Atome als eine Art "tag-field" benutzt. Um diese zu klassifizieren, wurde die beiden oben genannten Funktionen geschaffen.

norm_atom(A) wird wahr, wenn es sich bei *A* um einen "normales" Atom handelt, *A* also keine Nebenbedeutung als tag-field hat. Dies sind in der vorliegenden Implementation alle Atome außer *UNBOUND*, *VART* und *SKELT* zur Kennzeichnung von Variablen und *INTT*, *LONGT* und *REALT* zur Kennzeichnung von Zahlen.

Dabei nehmen die Atome *LONGT* und *REALT* eine Zwischenstellung ein. Einerseits werden sie von bestimmten Funktionen (Ein-/Ausgabe, Arithmetik, Termvergleich) gesondert als numerischer Wert interpretiert, andererseits stellen sie aber für andere Funktionen (z.B. Unifizierung) ganz normale Funktoren dar. Ob ein Atom als Funktor zu interpretieren ist (also ob die im Kapitel 2.3.1 beschriebene Argumentstruktur vorhanden ist), wird durch *func_atom(A)* entschieden.

Abstrakte Datentypen in HU-Prolog

2.2.3 Die Atomtabelle

Zur Verwaltung der im Heap abgespeicherten Atome dient eine spezielle Datenstruktur, die Atomtabelle. Diese hat mehrere Funktionen:

- a. Eine Verwaltung der Atome, welche den Zugriff auf alle Atome in geordneter Reihenfolge (geordnet nach dem String, der das Atom repräsentiert) ermöglicht
- b. Schneller Zugriff zu den einzelnen Atomen über ihren Namen und ihre Stellenzahl.(Hashtabelle)
- c. Schneller Zugriff zu allen Atomen, die sich von einem gegebene Atom nur durch die Stellenzahl unterscheiden. Speziell ist für die effektive Realisierung des HU-Prolog Prädikates `:=/2` der schnelle Zugriff auf das Atom mit einer um 1 höheren Stellenzahl nötig.
- d. Realisierung verschiedener Sichtbarkeitsbereiche von Namen (vgl. die HU-Prolog Prädikate `private/1` und `hide/1`)

Bei der Atomtabelle handelt es sich um eine Datenstruktur, die aus der Verkettung der Daten vom Typ `ATOM` über die Komponenten `nextatom()` und `chainatom()` gebildet wird.

2.2.3.1 Aufbau der Atomtabelle

Die Atome verschiedener Stellenzahl aber demselben Namen sind über die Komponente `chainatom()` als Liste verkettet. Um einen schnellen Zugriff zum Atom mit demselben Namen aber einer um 1 höheren Stellenzahl zu erhalten, sollte diese Liste nach aufsteigender Stellenzahl geordnet sein. Aus Gründen, die später noch beschrieben werden, kann aber das zuerst in die Atomtabelle aufgenommene Atom dieses Namens nicht mehr von der ersten Position dieser Liste entfernt werden. Ein Ausweg aus diesem Dilemma wäre, beim Einfügen eines Atoms mit einem neuen Namen in die Atomtabelle immer gleichzeitig das Atom mit demselben Namen und der Stellenzahl `0` in die Atomtabelle einzufügen. Dieses führt aber im allgemeinen zu einer Speicherplatzverschwendungen. Deswegen ist diese Liste zyklisch organisiert.

`chainatom(A)` verweist also immer auf

- ein Atom mit demselben Namen, aber mit höheren Stellenzahl als `A` - wenn dieses existiert, sonst auf
- das Atom mit demselben Namen und der kleinsten vorhandenen Stellenzahl zu diesen Namen. Das kann auch

Abstrakte Datentypen in HU-Prolog

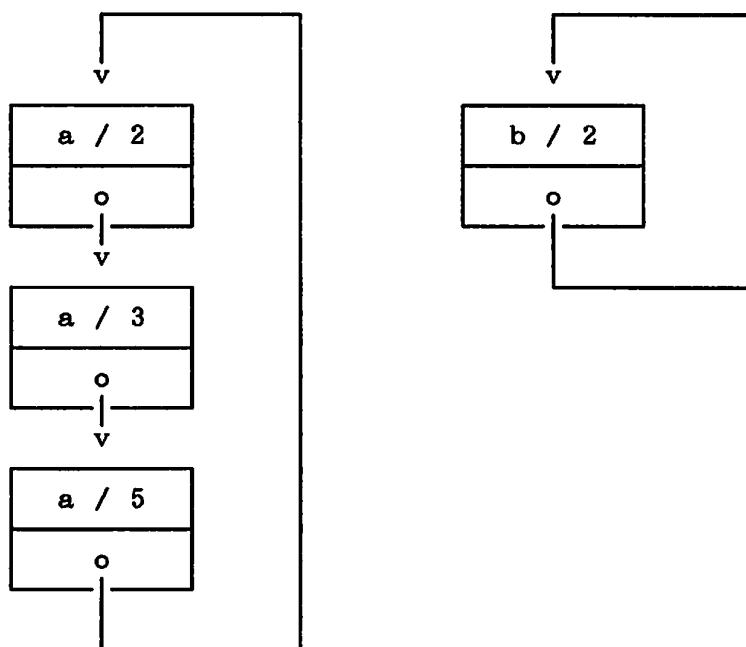
das Atom A selber sein.

Beispiel:

Ein Atom sei durch
nebenstehende
Struktur dargestellt

| |
|------------|
| name/arity |
| chainatom |

In der Atomtabelle seien die Atome $a/2$, $a/3$, $a/5$ und $b/2$ vorhanden. Daraus ergibt sich folgende Struktur:



Diese einzelnen Listen von Atomen gleichen Namens sind nun wiederum untereinander über die Komponenten `nextatom()` linear verkettet. Um diese Verkettung zu realisieren, ist in jeder der oben beschriebenen zyklischen Liste ein Atom ausgezeichnet, welches einen Verweis auf das ausgezeichnete Atom einer anderen zyklischen Liste enthält. Bei dem ausgezeichneten Atom handelt es sich dabei immer um das zuerst eingefügte Atom dieses Namens in die Atomtabelle. Diese wird durch das Attribut `first()` gekennzeichnet. Die lineare Verkettung dieser Listen über die Komponente `nextatom()` erfolgt dabei immer zu einem Atom, mit einem "größeren" Namen. Als Ordnungsfunktion wird dabei der

Abstrakte Datentypen in HU-Prolog

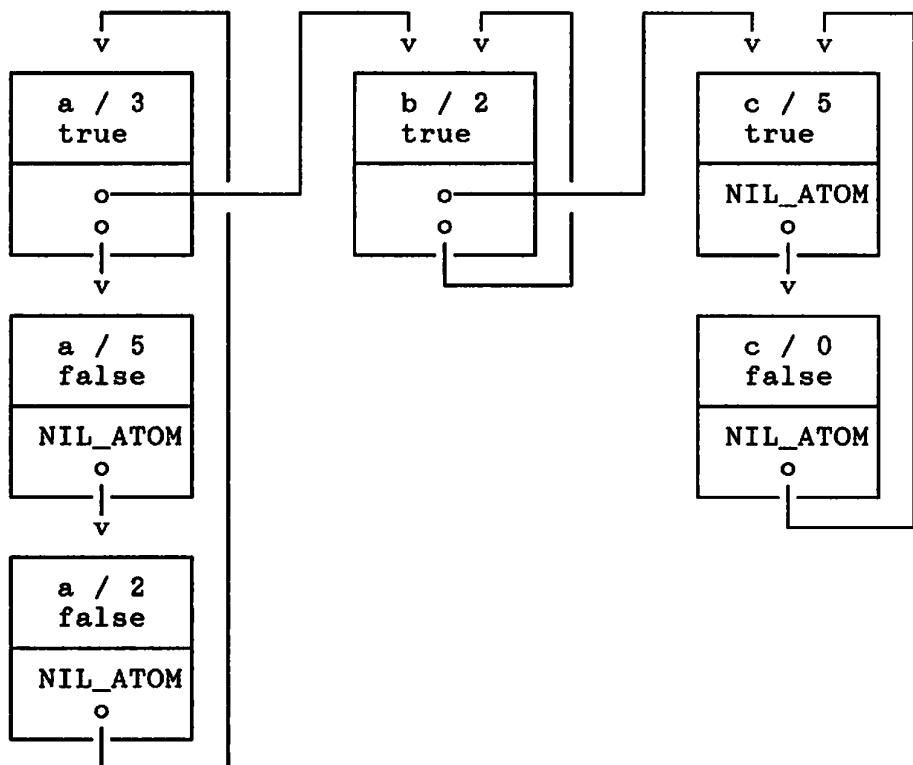
ASCII-Code zugrunde gelegt.

Beispiel:

Ein Atom sei durch die nebenstehende Struktur dargestellt

| |
|------------|
| name/arity |
| first |
| nextatom |
| chainatom |

In die Atomtabelle seien die Atome $c/5$, $c/0$, $a/3$, $a/2$, $a/5$ und $b/2$ in dieser Reihenfolge eingetragen worden. Daraus ergibt sich folgende Struktur.



Damit beim Suchen von Atomen in dieser Datenstruktur nicht immer die ganze über die Komponente `nextatom()` realisierte Liste durchsucht werden muß, wurde sie mit Hilfe einer Hashtabelle

Abstrakte Datentypen in HU-Prolog

```
ATOM HASHTAB[ HASHSIZE ];
```

in *HASHSIZE* viele dieser über *nextatom()* verketteten Listen verlegt. Den Eintrittspunkt in diese Hashtabelle liefert zu einem gegebenen Namen die Funktion:

```
cardinal strhash(s)
    char *s;
```

Damit weiterhin die Möglichkeit besteht, alle Namen von in der Atomtabelle enthaltenen Atomen in geordneter Reihenfolge zu erhalten, ist für die Hashfunktion *strhash()* folgendes festgelegt:

```
Für zwei Zeichenketten s1 und s2 mit s1 kleiner s2
( in C : strcmp(s1,s2) < 0 )
gilt immer strhash(s1) <= strhash(s2)
```

Nun noch ein kleines Beispiel zur Demonstration des Aufbaus der Atomtabelle. Die folgende Funktion gibt alle Namen von in der Atomtable stehenden Atomen in geordneter Reihenfolge aus.

```
void print_all_names()
{
    void ws(); /* Eine Funktion zur Ausgabe von C-Strings */
    cardinal hashval;
    ATOM A;

    for( hashval = 0 ; hashval < HASHSIZE ; ++hashval )
        for(A = HASHTAB[hashval] ; A != NIL_ATOM ; A = nextatom(A))
    {
        ws(tempstring(atomstring(A)));
        ws("\n");
    }
}
```

2.2.3.2 Die Realisierung verschiedener Sichtbarkeitsebenen

Um verschiedene Sichtbarkeitsebenen von Namen in HU-Prolog zu realisieren, muß der oben beschriebene Aufbau der Atomtabelle etwas korrigiert werden. Es wurde gesagt, daß die Atome gleichen Namens über die Komponente *chainatom()* miteinander verkettet sind. Damit sind jedoch nur die Namen eines Sichtbarkeitsbereiches gemeint. Zu ein und demselben Namen können also durchaus mehrere (über *chainatom()* verkettete) zyklische Listen existieren. Von diesen gilt aber nur eine als zum jeweiligen Zeitpunkt sichtbar. Sie ist dadurch gekennzeichnet, daß die in ihr enthaltenen Atome weder das *private()* noch das *hide()* Attribut gesetzt haben. Zur genauen Beschreibung der Sichtbarkeit sei auf die

Abstrakte Datentypen in HU-Prolog

Implementation der auf der Atomtabelle operierenden Funktionen verwiesen. (*LOOKUP()*; *LOOKATOM()*; *PRIVATE()*; *HIDE()* und *GetAtom()* im File *atoms.c* der beiliegenden Quelltexte).

2.2.4 High-Level Verwaltung

Nachdem der Aufbau und die Funktion der Atomtabelle erläutert wurde, folgen nun einige 'High-Level' Funktionen. Diese bilden die eigentliche Schnittstelle zu der Verwaltung der Atomtabelle.

```
ATOM LOOKUP(str,ar,create,in_heap)
    char *str;      /*Zeichenkette des Atoms*/
    cardinal ar;   /*Stellenzahl des Atoms*/
    boolean create; /*Wenn nicht vorhanden, neu erzeugen*/
    boolean in_heap; /*nur im Heap suchen und erzeugen*/
```

Diese Funktion liefert als Resultat das Atom, welches durch den C-String *st* und der Stelligkeit *ar* definiert ist. Die Parameter *create* und *in_heap* steuern die Abarbeitung. Dabei wird folgendermaßen vorgegangen:

- a. Es wird in der Atomtabelle nach einem Atom gesucht, dessen Name und Stellenzahl mit den spezifizierten Argumenten übereinstimmen. Wenn es gefunden wird, so ist dieses Atom das Resultat der Funktion. Wird es nicht gefunden, und hat der Parameter *in_heap* den Wert *true*, so wird bei d) fortgesetzt; hat *in_heap* den Wert *false*, wird bei b) fortgesetzt
- b. Es wird der Atomstack nach einem Atom durchsucht, dessen Name und Stellenzahl mit den spezifizierten Argumenten übereinstimmen. Wenn es gefunden wird, so ist dieses Atom das Resultat der Funktion. Wird es nicht gefunden, und der Parameter *create* hat den Wert *true*, so wird bei c) fortgesetzt. Andernfalls liefert die Funktion den Resultatwert *NIL_ATOM* und zeigt damit an, daß kein Atom gefunden wurde.
- c. Es wird ein neues Atom im Stack erzeugt und mit den übergebenen sowie mit Standardparametern initialisiert. Dieses neue Atom ist der Resultatwert der Funktion.
- d. Wenn der Parameter *create* den Wert *true* hat, so wird ein neues Atom im Heap erzeugt, mit den übergebenen sowie mit Standardparametern initialisiert sowie in die Hashtabelle eingekettet. Dieses neue Atom ist der Resultatwert der Funktion. Andernfalls liefert die Funktion den Resultatwert *NIL_ATOM* und

Abstrakte Datentypen in HU-Prolog

zeigt damit an, das kein Atom gefunden wurde.

In der Regel sollte *in_heap* den Wert *false* haben, das Atom also immer im Stack angelegt werden. Um ein Atom zwangsweise im Heap zu erzeugen, sollte man die weiter unten beschriebene Funktion *CopyAtom()* nutzen, da sonst eventuell Konsistenzprobleme im Stack entstehen können. (siehe auch *CopyAtom()*)

```
ATOM LOOKATOM(A,ar,create)
ATOM A;
cardinal ar; /*die Stellenzahl des neuen Atoms*/
boolean create; /*Wenn nicht vorhanden, neu erzeugen*/
```

Diese Funktion liefert als Resultat das zu *A* gehörige Atom mit der Stellenzahl *ar*. Dabei sind folgende Fälle zu unterscheiden:

- a. *A* liegt im Heap und ist als *private* oder *hide* gekennzeichnet. Das zu erzeugende Atom wird in der zu *A* gehörenden (durch *chainatom()* gebildeten) Kette in der Atomtabelle gesucht und, wenn es nicht vorhanden ist und *create* den Wert *true* hat, in dieser Kette neu erzeugt.
- b. In jedem anderen Falle ist die Funktion semantisch äquivalent zu

```
LOOKUP(tempstring(atomstring(A)),ar,create,false)
```

nur eventuell effektiver implementiert.

Ein wesentlicher Unterschied von *LOOKUP()* und *LOOKATOM()* besteht also darin, daß *LOOKATOM()* auch Atome bearbeiten und erzeugen kann, die mit *private* oder *hide* gekennzeichnet sind.

```
ATOM CopyAtom(A)
ATOM A;
```

Diese Funktion dient dem Kopieren eines Atoms aus dem Atomstack in den Atomheap. Dabei wird gleichzeitig der Termstack traversiert und alle dort vorhandenen Verweise auf das Atom *A* in Verweise auf das neue Atom umgewandelt (vgl. Beschreibung des Datentyps *TERM*). Dadurch bleibt die Konsistenz gewahrt.

Abstrakte Datentypen in HU-Prolog

```
ATOM GetAtom(A)
ATOM A;
```

Diese Funktion bietet eine einfache Möglichkeit die gesamte Atomtabelle zu traversieren. *GetAtom()* liefert zu einem gegebenen Atom *A* das in der Atomtabelle folgende Atom. Dabei werden Atome welche mit *private()* oder *hide()* gekennzeichnet sind übergangen. Als erstes und letztes Atom ist *NIL_ATOM* angesetzt. Damit ist die Bearbeitung aller (sichtbaren) Atome der Atomtabelle wie folgt möglich:

```
ATOM A;
...
A = NIL_ATOM;
while((A = GetAtom(A)) != NIL_ATOM)
{
    /*
    ** Beliebige Aktionen, die A nicht ändern
    */
}
void PRIVATE(A)
ATOM A;
```

Diese Funktion eröffnet eine neue Sichtbarkeitsebene für das Atom *A* und alle Atome mit demselben Namen und anderer Stellenzahl (Genauer: für alle Atome die in der Atomtabelle über *chainatom()* verkettet sind). In vorhandenen Atome mit diesem Namen wird das *privat()*-Attribut gesetzt. Es ist zu beachten, daß diese Funktion eventuell Inkonsistenzen im Stack hervorrufen kann. Daraus resultiert die Forderung, daß das HU-Prolog Prädikat *private/1* nur auf dem Toplevel (bei leeren Stacks) gerufen werden sollte.

```
void HIDE(A)
ATOM A;
```

Diese Funktion beendet die Sichtbarkeit eines Atoms (indem das *hide()*-Attribut gesetzt wird), und macht vorher durch *PRIVATE()* verdeckte Atome wieder sichtbar (indem auf den zuletzt mit *PRIVATE()* verdeckten Atomen das *private()*-Attribut wieder gelöscht wird. Für weitergehende Informationen sei auf die Implementation dieser Prädikate verwiesen.

Abstrakte Datentypen in HU-Prolog

2.3 Der Datentyp TERM

Aufbauend auf den Datentyp *ATOM* kann nun der Datentyp *TERM* beschrieben werden.

2.3.1 Die Komponenten des Datentyps TERM

```
LVALUE ATOM name(T)
      TERM T;
```

Das Atom (der Hauptfunktor) des Termes *T*. Zusätzlich kann diese Komponente noch einige Werte annehmen, die von allen anderen Atomen verschieden sind. Diese dienen gewissermaßen als "tag-field", wenn es sich bei dem Term um einen Endknoten des Baumes handelt, der kein Atom im üblichen Sinne ist. Also zum Beispiel eine Variable oder eine Zahl.

```
LVALUE TERM son(T)
      TERM T;
LVALUE TERM val(T)
      TERM T;
LVALUE int ival(T)
      TERM T;
LVALUE cardinal offset(T)
      TERM T;
```

Diese 4 Komponenten sind als *union* organisiert, das heißt es wird immer höchstens eine belegt. Die Entscheidung, welcher der Komponenten gerade aktuell ist, ergibt sich aus dem Wert von *name(T)*.

- a. Wenn *name(T)* den Wert *INTT* hat, handelt es sich bei dem Term um die Darstellung einer ganzen Zahl und *ival(T)* ergibt den numerischen Wert dieser Zahl.
- b. Wenn *name(T)* den Wert *SKELT* hat, so handelt es sich um einen Platzhalter für eine Variable in einer Klausel. Ähnlich wie in klassischen Programmiersprachen wird auch in Prolog für Variablen die im "Programm" (sprich in den Klauseln der Prolog-Datenbasis) stehen beim "Aufruf der Funktion" (sprich Abarbeitung der Klausel) ein Stacksegment angelegt, welches den Speicherplatz für die lokalen Variablen enthält. *offset(T)* enthält in diesem Fall einen Offset zum Beginn dieses Stacksegmentes und identifiziert damit die Variable.
Da das Programm nur im Heap steht, tritt auch dieser Wert für *name(T)* nur bei Termen im Heap auf.

Abstrakte Datentypen in HU-Prolog

- c. Wenn `name(T)` den Wert `VART` hat, so handelt es sich bei dem Term um eine gebundene Variable. `val(T)` verweist dann auf den Wert dieser Variable, das heißt, auf den Term an den `T` gebunden ist. Dieser Wert von `name(T)` tritt nur bei Termen im Stack auf. (im Heap stehen nur Offsets zum aktuellen Stacksegment - siehe `SKELT`) `val(T)` verweist ebenfalls auf einen Term im Stack.
- d. Wenn `name(T)` den Wert `UNBOUNDT` hat, so handelt es sich um eine ungebundene Variable, und es wird keine der vier Komponenten belegt. Dieser Wert für `name(T)` tritt nur bei Termen im Stack auf.
- e. In allen anderen Fällen handelt es sich bei `name(T)` um einen normalen Funktor (`func_atom(name(T)) != false`) und `son(T)` verweist auf die Argumente des Termes (falls solche vorhanden sind).

Noch einige Bemerkungen zu den drei Arten von Variablen. Um den Wert einer Variablen `T`, das heißt den Term, auf den sie letztendlich verweist, zu ermitteln, muß wie folgt vorgegangen werden.

```
if(name(T) == SKELT)
    T = n_brother(VAR_BLOCK,offset(T));
```

In diesem Fall handelt es sich bei `T` um den Platzhalter für eine Variable in einer Klausel. Die eigentliche Variable wird zur Abarbeitungszeit im Stack angelegt. `VAR_BLOCK` enthält den Anfang des entsprechenden Stacksegmentes, `n_brother()` ist eine weiter unten beschriebene Funktion.

```
while(name(T) == VART)
    T = val(T);
```

`T` ist eine gebundene Variable. Der Wert von `T` ist `val(T)`. Dieses Verfahren wird solange fortgesetzt, bis `name(T)` einen anderen Wert als `VART` hat. Jetzt ist `T` entweder eine ungebundene Variable (`name(T) == UNBOUNDT`) oder ein "normaler" Term. Dieses hier angegebene Verfahren nennt man dereferenzieren. Dafür wurde in HU-Prolog ein Makro eingeführt.

```
#define deref_(T,VAR_BLOCK) {
    if(name(T)==SKELT)
        T = n_brother(VAR_BLOCK,offset(T));
    while(name(T) == VART)
        T = val(T); }
```

Abstrakte Datentypen in HU-Prolog

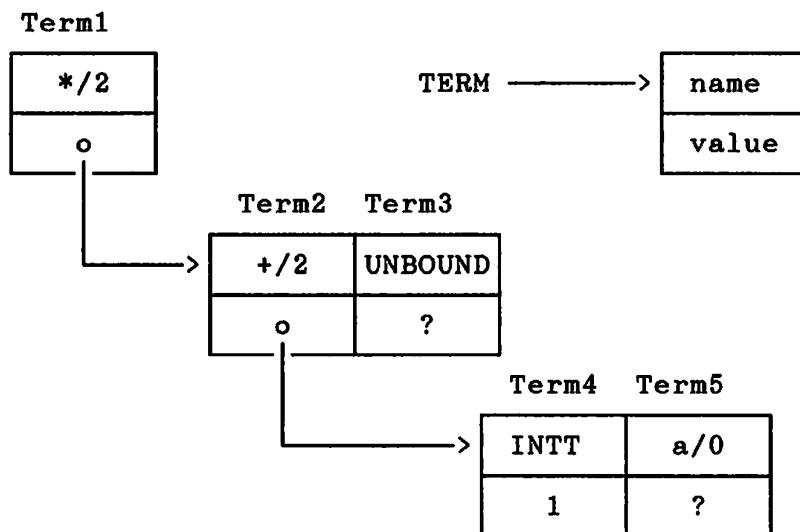
```
TERM brother(T)
TERM T;
```

Bei *brother(T)* handelt es sich um die Verkettung der Argumente eines strukturierten Termes. Es ist zu beachten, daß dieses Makro keinen L-Value darstellt. Diese Komponente ist auch nicht physisch vorhanden. In der Termdarstellung wird davon ausgegangen, daß die Terme, die die Argumente eines strukturierten Terms bilden, im Speicher physisch nebeneinanderliegen. Der Zugriff über *brother(T)* dient lediglich dazu, den Zugriff auf den "nächsten" Term sauber zu definieren.

Die Benutzung der *brother()*-Komponente wird am folgende Beispiel demonstriert.

Beispiel:

Der Term `'*'('+'(1,a), _)` (eine andere Schreibweise für `((1 + a) * _)`) sieht folgendermaßen aus: (Verweise auf Atome werden in der Kurzform *name/arity* angegeben; ein ? kennzeichnet einen beliebigen Wert)



Dabei gelten folgende signifikanten Beziehungen:

```
son(Term1) == Term2
brother(Term2) == Term3
son(Term2) == Term4
brother(Term4) == Term5
```

Abstrakte Datentypen in HU-Prolog

Es ist zu beachten, daß zum Beispiel der Ausdruck *brother(Term3)* einen nicht definierten Wert hat. Die Ermittlung des letzten Argumentes eines Funktors muß über die Stellenzahl des Funktors erfolgen.

```
MACRO TERM next_brother(T)
  TERM T;
TERM n_brother(T,N)
  TERM T;
  cardinal N;
```

next_brother(T) ist semantisch äquivalent zu
T = brother(T).

kann aber eventuell effektiver implementiert werden. Da dieser Aufruf sein Argument ändert, ist er immer als Makro implementiert. Dies soll durch die Deklaration als MACRO dokumentiert werden.

n_brother(T,N) erlaubt den Zugriff auf den *N*-ten *brother()* von *T*. Eine mögliche Variante der Realisierung ist:

```
TERM n_brother(T,N)
  TERM T;
  cardinal N;
{
  while( N-->0)
    next_brother(T);
  return T;
}
```

Eine vordefinierte Konstante des Datentyps TERM ist *NIL_TERM*.

2.3.2 Low-Level Verwaltung

Wie schon bei der Beschreibung der Termstruktur erwähnt, wird die Verkettung der Argumente eines Funktors dadurch realisiert, daß diese Terme im Speicher physisch nebeneinander liegen. (Verkettung über die Pseudokomponente *brother()*) Deswegen folgt die Verwaltung von Termen der Blockstruktur.

```
TERM stackterm(N)
  cardinal N;
```

Ein Aufruf von *stackterm(N)* liefert als Resultat den Anfang eines Blockes aus *N* Termen. (In anderer Terminologie: *N* über das *brother()*-Feld verkettete Terme) Dieser Block liegt an der Spitze des Term-Stacks.

Abstrakte Datentypen in HU-Prolog

```
TERM heapterm(N)
    cardinal N;
```

Die Funktion *heapterm()* erledigt analoges für den Term-Heap.

Im Gegensatz zum Atom-Heap ist für den Term-Heap eine Freispeicherverwaltung realisiert.

```
void freeblock(T,N)
TERM T;
cardinal N;
```

Diese Funktion gibt einen bei *T* beginnenden Block von *N* Termen wieder frei, das heißt, daß dieser Block von eventuell folgenden Aufrufen von *heapterm()* geliefert werden kann. Es ist unbedingt darauf zu achten, daß es sich bei dem Term *T* um einen durch *heapterm(N)* erzeugten Term handelt.

Auch für die Terme gibt es die (schon langsam gewohnten) Funktionen zur Kontrolle des Stacks.

```
void init_term_stack();
TERM mark_term_stack(); und
void rel_term_stack(T)
TERM T;
```

init_term_stack() initialisiert den Termstack. Nach Aufruf dieser Funktion ist der Termstack leer. Die Funktion *mark_term_stack()* liefert die Spitze des Stacks als Resultat. Mit *rel_term_stack(T)* wird der Termstack auf den in *T* markierten Wert zurückgesetzt. *T* ist danach wieder oberstes Element des Termstacks. Auf den durch *mark_term_stack()* gelieferten Term, sollte nur durch die Funktion *rel_term_stack()* zugegriffen werden.

Um zu entscheiden, ob es sich bei einem gegebenen Term um einen Term im Stack oder einen Term im Heap handelt, wurden die folgenden Funktionen eingeführt:

```
boolean is_heap_term(T)
TERM T;
boolean is_stack_term(T)
TERM T;
```

Die Anwendung dieser Funktionen auf die Konstante *NIL_TERM* ist nicht definiert.

Desweiteren ist es für die Unifizierung von Interesse, welcher von zwei Stackterminen "älter" ist, daß heißt, welcher

Abstrakte Datentypen in HU-Prolog

früher im Stack erzeugt wurde. Dazu gibt es die Funktion

```
boolean is_older_term(Older, Younger)
    TERM Older, Younger;
```

In bestimmten Fällen ist es auch notwendig, auf alle Terme im Termstack zugreifen zu müssen. Dazu gibt es die Funktion

```
TERM get_stack_term(T)
    TERM T;
```

Diese Funktion arbeitet auf dem Termstack analog zu `get_stack_atom()` auf dem Atomstack. (siehe Kapitel 2.2.2) Es wird jeweils der auf `T` "folgende" Term zurückgegeben. Anfang und Ende dieser Ordnungsrelation bildet die Konstante `NIL_TERM`.

2.3.3 High-Level Verwaltung

Zur High-Level Verwaltung von Termen gibt es zwei Sorten von Funktionen. Zum einen sind das Funktionen, zum Aufbauen von oft benötigten Strukturen. Zum anderen eine Funktion zum Kopieren und Transformieren von Stackterminen in Heapterme.

```
TERM mkfreevar()
```

Diese Funktion liefert als Resultat einen Term im Stack, der eine ungebundene Variable repräsentiert. Der Aufruf von

```
T = mkfreevar();
```

ist äquivalent zu

```
T = stackterm(1);
name(T) = UNBOUNDT;
```

```
TERM mkint(I)
    int I;
```

Diese Funktion liefert als Resultat einen Term im Stack, der eine Zahl darstellt. Der Aufruf von:

```
T = mkint(i);
```

ist äquivalent zu

```
T = stackterm(1);
name(T) = INTT;
ival(T) = i;
```

Abstrakte Datentypen in HU-Prolog

TERM *mkfunc(A,T)*
ATOM *A*;
TERM *T*;

Diese Funktion liefert als Resultat einen Term im Stack, der einen normalen Funktor darstellt. Der Aufruf

T = mkfunc(A,T1);

ist äquivalent zu

T = stackterm(1);
name(T) = A;
son(T) = T1;

TERM *mk2sons(A1,S1,A2,S2)*
ATOM *A1,A2*;
TERM *S1,S2*;

Diese Funktion liefert als Resultat den ersten Term eines aus 2 Elementen bestehenden Termblockes (In anderer Terminologie: zwei über das *brother()* Feld verbundene Terme) Der Aufruf

T = mk2sons(A1,S1,A2,S2);

ist äquivalent zu

T = stackterms(2);
name(T) = A1;
son(T) = S1;
name(brother(T)) = A2;
son(brother(T)) = S2;

TERM *stackvar(N)*
cardinal *N*;

Diese Funktion gibt als Resultat ein Block von *N* ungebundenen Variablen zurück. (Im Gegensatz zu *stackterm(N)* werden die Terme also auf UNBOUNDT initialisiert)

Abstrakte Datentypen in HU-Prolog

```
TERM arg1(T)
      TERM T;
TERM arg2(T)
      TERM T;
TERM arg3(T)
      TERM T;
TERM arg4(T)
      TERM T;
```

Diese 4 Funktionen ermöglichen den komfortablen Zugriff auf die ersten 4 Argumente eines strukturierten Terms. Der Resultatsterm ist dabei schon dereferenziert worden, das heißt, daß er weder eine gebundene Variable noch ein "Platzhalter" in einer Klausel ist.

Die Benutzung dieser Funktionen soll am folgenden Beispiel erläutert werden. Es soll der (schon oben erwähnte) Term $((1 + a) * _)$ konstruiert werden.

```
A = LOOKUP("a", 0, true, false);
/* Das Atom a/0 wurde konstruiert. */
Term4 = mk2sons(INTT, 1, A, nil);
Term2 = mk2sons(PLUS_2, Term4, UNBOUNDT, nil);
Term1 = mkfunc(TIMES_2, Term2); /* Resultatsterm */
```

Bemerkungen:

- *PLUS_2* und *TIMES_2* sind zwei vordefinierte Atome für $'+'/2$ und $'*'/2$
- Die Namen von *Term1*, *Term2* und *Term4* sind entsprechend dem obriegen Bild gewählt

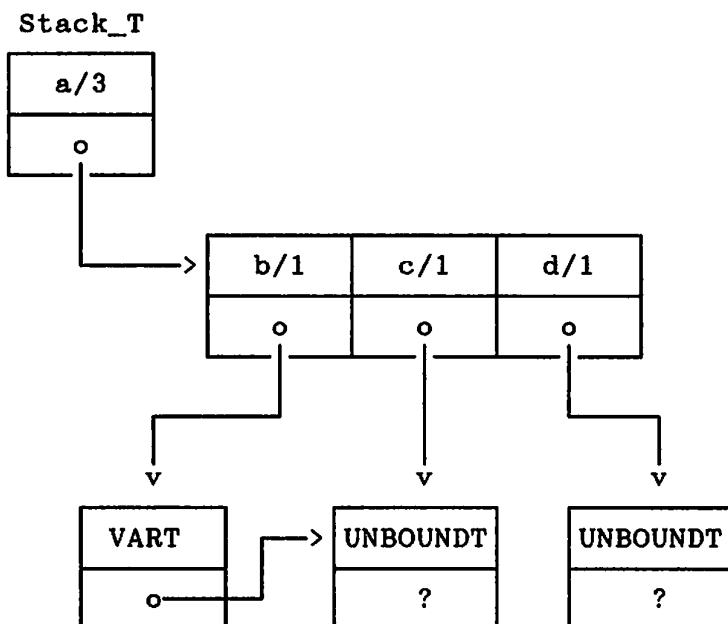
```
void Init_Skeleton();
TERM Skeleton(T, N)
TERM T;
cardinal N;
cardinal Var_Count();
```

Die Funktion *Skeleton()* dient dem Kopieren eines Blockes aus *N* Termen aus dem Stack in den Heap. Da im Heap keine Variablen vorkommen dürfen, sondern nur Platzhalter (siehe Beschreibung der Komponente *offset()* des Types *TERM*), werden von der Funktion *Skeleton()* Variablen durch solche Platzhalter ersetzt. Dabei wird garantiert, daß gleiche Variablen auch zu gleichen Platzhaltern transformiert werden. Dies funktioniert auch über mehrere *Skeleton()*-Aufrufe hinweg, das heißt, daß die Variablenzuordnungen nach Beendigung von *Skeleton()* nicht verworfen werden. Um diese Zuordnung von

Abstrakte Datentypen in HU-Prolog

Variablen und Platzhaltern neu zu initialisieren dient die Funktion *Init_Skeleton()*. Des weiteren gibt die Funktion *Var_Count()* die Anzahl der verschiedenen transformierten Variablen zurück. Das alles wird im folgenden Beispiel demonstriert.

Es sei der Stackterm $a(b(X), c(X), d(Y))$ gegeben. Er hat intern folgende Struktur:

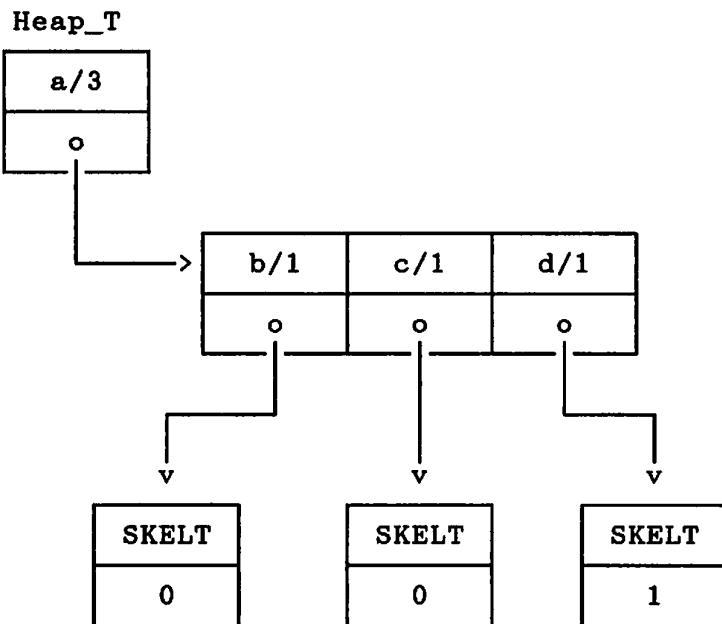


Das Resultat des Aufrufs von

```
Init_Skeleton();
Heap_T = Skeleton( Stack_T, 1);
```

Abstrakte Datentypen in HU-Prolog

baut einen Term der folgenden Form auf:



Das selbe Resultat erhielte man, wenn nach einmaligem Aufruf von *Init_Skeleton()* die Argumente von *a/3* durch einzelne Aufrufe von *Skeleton()* erzeugt würden. Es scheint erst einmal unnatürlich, *Skeleton()* mehrfach aufzurufen, doch beim Aufbau von Klauseln ist dies sinnvoll. (Näheres dazu beim Datentyp *CLAUSE*)

2.4 Der Datentyp CLAUSE

Ein Prologprogramm besteht aus einer Menge von Klauseln, die aus syntaktischer Sicht Terme bilden. Der Datentyp *CLAUSE* realisiert die Verkettung dieser Klauseln und den Verweis auf die Terme, die die Klauseln repräsentieren.

2.4.1 Komponenten des Datentyps CLAUSE

Ein Datum vom Typ *CLAUSE* besitzt folgende Komponenten:

TERM head(C)
 CLAUSE C;

Diese Komponente ist der Term, der den Kopf der Klausel bildet. Dieser Term ist im Heap abgelegt.

Abstrakte Datentypen in HU-Prolog

TERM body(C)
CLAUSE C;

Ist eine Termkette, die den Klauselkörper repräsentiert. Um sowohl die Abarbeitungszeit als auch den Speicherplatz zu optimieren, wird die Struktur des Klauselkörpers modifiziert. Wenn im Klauselkörper das Atom ','/2 als Hauptfunktor auftritt, wird es eliminiert und die einzelnen Aufrufe werden über das *brother()*-Feld der Termstruktur miteinander verkettet. Da das Ende des so entstehenden Termes nun nicht mehr über die Stellenzahl des Hauptfunktors ermittelt werden kann, endet so eine Kette mit einem Term, dessen Name entweder das Atom *NIL_ATOM* oder das Atom *GOTOT* bildet. Wenn die Kette mit einem Term mit dem Atom *GOTOT* endet, wird der Klauselkörper mit dem Term fortgesetzt, auf den die *son()*-Komponente zeigt.

LVALUE cardinal nvars(C)
CLAUSE C;

ist die Anzahl der Variablen in dieser Klausel. Diese Information wird benötigt, um bei der Abarbeitung ein entsprechend großes Stacksegment für die lokalen Variablen der Klausel anzulegen.

LVALUE CLAUSE nextcl(C)
CLAUSE C;

ist die Verkettung der zu einem Prädikat gehörenden Klauseln untereinander.

Beispiel:

Ein Datum vom
Typ CLAUSE
sei durch
nebenstehende
Struktur
dargestellt

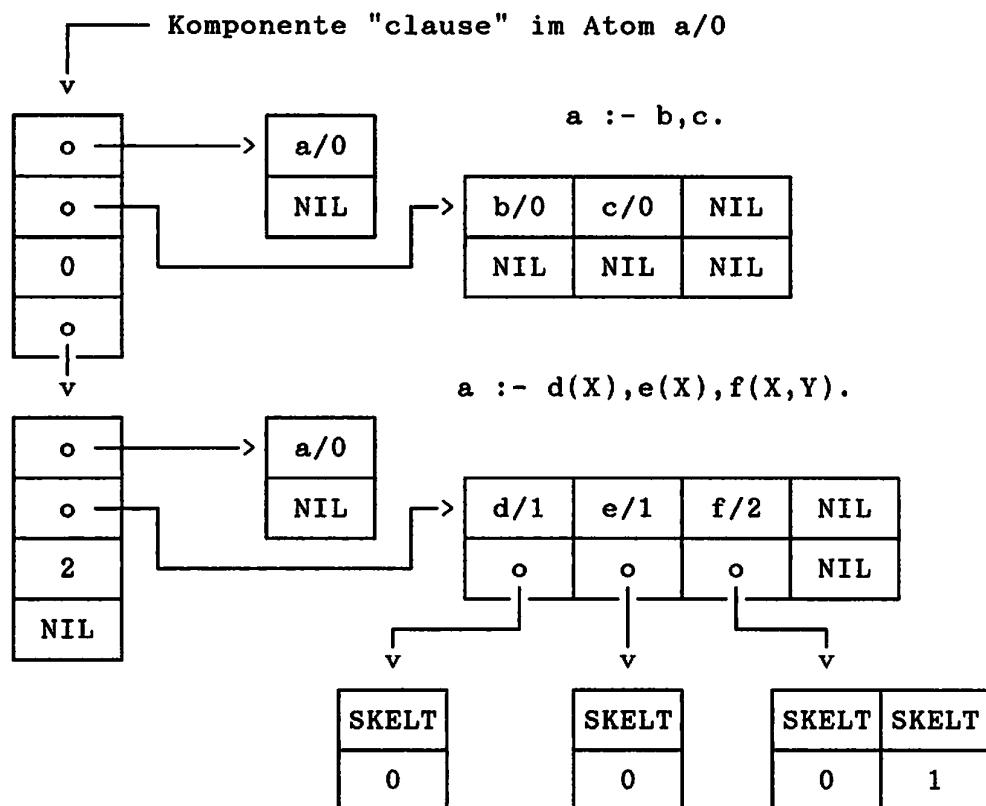
| |
|--------|
| head |
| body |
| nvars |
| nextcl |

Abstrakte Datentypen in HU-Prolog

Die Prolog-Datenbasis enthalte folgende Klauseln zum Prädikat *a/0*:

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

Dies sieht intern wie folgt aus:



2.4.2 Funktionen zur Verwaltung der Klauseln

Zum Erzeugen, Analysieren und Freigeben von Klauseln gibt es folgende Funktionen:

```
CLAUSE NewClause( Head, Body )
TERM Head, Body;
```

Diese Funktion konstruiert eine neue Klausel mit dem angegebenen Kopf und Körper. Da dieser Term Programmcode werden soll, wird er im Heap angelegt. Dabei werden evtl. in *Head* und *Body* auftretenden Variablen in entsprechende Offsets transformiert. Weiterhin wird die

Abstrakte Datentypen in HU-Prolog

oben beschriebene Elimination des ','/2 Operators durchgeführt.
Die beiden Terme *Head* und *Body* müssen Stackterme sein.
Wenn *Body* den Wert *NIL_TERM* hat, so handelt es sich bei der entstehenden Klausel um einen Fakt.

```
void FreeClause( C )
    CLAUSE C;
```

Diese Funktion gibt den von der Klausel beanspruchten Speicherplatz wieder frei. Das betrifft sowohl die Datenstruktur *CLAUSE* als auch Terme, auf die diese Datenstruktur verweist.

2.5 Der Datentyp TRAIL

Der durch den Datentyp *TRAIL* gebildete Trail-Stack dient der Verwaltung der im Falle des Backtrackings zu lösenden Variablenbindungen.

2.5.1 Die Komponenten des Datentyp TRAIL

```
LVALUE TERM boundvar( T )
    TRAIL T;
```

Dies ist ein Verweis auf einen Term, der eine ehemalige ungebundene Variable darstellt, welche im Laufe der Unifizierung gebunden wurde.

2.5.2 Funktionen zur Verwaltung des Datentyp TRAIL

Da der Trail als Stack organisiert ist, werden hier wieder die Funktionen zur Kontrolle des Stacks benötigt.

```
void init_trail_stack();
TRAIL mark_trail_stack()
void rel_trail_stack( T )
    TRAIL T;
```

init_trail_stack() setzt den Trail-Stack auf den leeren Stack zurück. Die Funktion *mark_trail_stack()* liefert die Spitze des Stacks als Resultat. Mit *rel_trail_stack(T)* wird der Trailstack auf den in *T* markierten Wert zurückgesetzt. Danach ist *T* wieder das oberste Stackelement. Gleichzeitig werden alle in dem von der Rücksetzung des Stacks betroffenen Trailelementen aufgezeichneten Terme wieder zu ungebundenen Variablen. (d.h. für alle frei werdenden Traileinträge wird folgende Operation ausgeführt:

Abstrakte Datentypen in HU-Prolog

```
name(boundvar(Released_Trail_entry)) = UNBOUNDT;
```

Auf das durch `mark_trail_stack()` gelieferte Datum vom Typ `TRAIL` sollte nur durch die Funktion `rel_trail_stack()` zugegriffen werden.

```
void trailterm(T)
TERM T;
```

Zeichnet den Term `T` im Trail auf. Dazu wird ein neues Element `New_Trailer` im Trailstack erzeugt. Anschließend wird in `boundvar(New_Trailer)` der Term `T` eingetragen.

2.6 Der Datentyp ENV

In klassischen Programmiersprachen wird für jeden Aufruf einer Funktion ein Stacksegment angelegt, das neben den lokalen Variablen dieser Funktion noch Verwaltungsinformationen enthält. So ähnlich ist es auch in HU-Prolog. Während die lokalen Variablen einer Klausel im Term-Stack abgelegt werden, werden die vom Interpreter benötigten Verwaltungsinformationen im sogenannten Environments abgelegt. (vergleiche Kapitel 1.)

2.6.1 Die Komponenten des Datentyps ENV

Nicht alle der folgenden Komponenten sind in jeder Prologimplementation vorhanden. In der Regel wird in anderen Systemen auch zusätzlich zwischen Environment und Choicepoint unterschieden. Zur genaueren Klärung sei auf die Realisierung des eigentlichen Inferenzprozesses in HU-Prolog verwiesen. (Funktion `EXECUTE()` im Kapitel 3.2.)

```
LVALUE TERM call(E)
ENV E;
```

`call(E)` enthält einen Verweis auf den Term im Programmspeicher, der den aktuellen Aufruf ausgelöst hat.

```
LVALUE CLAUSE rule(E)
ENV E;
```

Enthält die Klausel, die für diesen Aufruf als letzte probiert wurde. Im Backtracking würde (`nextcl(rule(E))`) für die Fortsetzung der Abarbeitung gewählt.

```
LVALUE TERM bct(E)
ENV E;
```

Abstrakte Datentypen in HU-Prolog

wird für backtrackbare Build-In Prädikate benutzt.

LVALUE TERM base(E)
ENV E;

base(E) ist ein Verweis in den Term-Stack. Er kennzeichnet den Beginn des Bereiches der lokalen Variablen. Wie schon oben erwähnt stehen in den Klauseln anstelle von Variablen nur Offsets zu *base(E)*

LVALUE ENV env(E)
ENV;

ist ein Verweis auf das Environment der rufenden Klausel.

LVALUE ENV choice(E)
ENV;

verweist auf das Environment, bei dem im Falle von Backtracking fortgesetzt werden soll. (Der sogenannte Choicepoint)

LVALUE TRAIL trail(E)
ENV;

verweist auf den Stand des Trail-Stacks beim aktuellen Aufruf. Wenn Backtracking zu einem Environment X augelöst wird, muß der Trail-Stack bis zum Stand *trail(X)* geleert werden, und die Bindungen der dabei entfernten Variablen aufgelöst werden.

LVALUE ATOM atomtop(E)
ENV;

verweist auf den Stand des Atom-Stacks. (vgl. *mark_atom_stack()* und *rel_atom_stack()* im Kapitel 2.2.2)

LVALUE STRING stringtop(E)
ENV;

verweist auf den Stand des String-Stacks . (vgl. *mark_string_stack()* und *rel_string_stack()* im Kapitel 2.2.1)

LVALUE TERM termtop(E)
ENV E;

verweist auf den Stand des Term-Stacks . (vgl. *mark_term_stack()* und *rel_term_stack()* im Kapitel 2.2.3)

Abstrakte Datentypen in HU-Prolog

2.6.2 Funktionen zur Verwaltung des Datentyps ENV

Zur Verwaltung der Environmentstacks gibt es folgende Funktionen:

ENV newenv()

legt ein neues Environment auf dem Environmentstack an.

Zur Kontrolle des Stacks gibt es wieder drei Funktionen.

```
void init_env_stack();
ENV mark_env_stack();
void rel_env_stack(E);
    ENV E;
```

Die Funktion *init_env_stack()* initialisiert den Environment-Stack. Nach Aufruf dieser Funktion ist der Environment-Stack leer. Die Funktion *mark_string_stack()* liefert die Spitze des Stacks als Resultat. Mit *rel_env_stack(E)* wird der Environmentstack auf den in *E* gespeicherten Wert zurückgesetzt. Das Environment *E* ist dann wieder das oberste Stackelement.

```
boolean is_older_env(E1,E2)
    ENV E1,E2;
```

entscheidet, welches von zwei Environments älter als das andere ist.

3. Der Interpreterkern

In diesem Kapitel wird die Funktionsweise des Interpreterkerns von HU-Prolog erklärt. Dabei handelt es sich sowohl um die Unifizierung, als auch um den Inferenzmechanismus.

3.1 Die Unifizierung in Prolog

Eine der grundlegendsten Operationen eines Prologsystems ist die Unifizierung zweier Terme. Zwei Terme werden miteinander unifiziert, indem durch Einführung von Variablenbindungen beide Terme identisch gemacht werden. (vgl. theoretische Hintergründe der Unifizierung in [4] und [5])

Da die zu unifizierenden Terme in HU-Prolog durch eine rekursive Datenstruktur beschrieben sind, lässt sich die Unifizierung auch rekursiv gut beschreiben.

Im folgenden soll die Unifizierung von zwei Termen anhand eines C-Quelltextes erläutert werden. Dieser ist so gewählt, dass er leicht verständlich ist, dafür lässt seine Effizienz aber zu wünschen übrig. Durch Modifizierung des Algorithmus lässt sich eine Beschleunigung des HU-Prologsystems um den Faktor zwei erreichen.

In der folgenden Funktion sind *Term1* und *Term2* die zu unifizierenden Termblöcke. Da dies eventuell Terme in Klauseln (also in der Datenbasis) sein können, wird zu jedem Term noch der Anfang der im Stack angelegten lokalen Variablen benötigt. (*B1* bzw. *B2*) Es ist dies in der Regel *base("Environment der Klausel")*. Wenn es sich bei dem Term um einen Stackterm oder um einen Term ohne Variablen handelt, werden diese Argumente nicht benutzt und könnten als *NIL_TERM* übergeben werden.

```
GLOBAL boolean UNIFY(N,Term1,Term2,B1,B2)
      cardinal N;
      TERM Term1,Term2;
      TERM B1,B2;

{
```

```
    TERM T1,T2;
    TRAIL TRAIL_TOP;
    TERM TERM_TOP;
```

Im Verlaufe der Unifizierung entstehen eventuell Variablenbindungen, die im Trail vermerkt werden. Andererseits werden im Term-Stack neue Strukturen angelegt. Wenn die Unifizierung nicht erfolgreich ist, muß der

Der Interpreterkern

Termstack zurückgesetzt werden, und die Variablenbindungen gelöst werden. Deswegen wird sich hier der Stand der Stacks vermerkt

```
TRAIL_TOP = mark_trail_stack();
TERM_TOP = mark_term_stack();
```

Für jedes Elementpaar aus der Termkette wird auf Unifizierung getestet:

```
for( ; N-- > 0 ; next_br(Term1),next_br(Term2))
{
    /* Die Argumente werden dereferenziert */
    T1 = Term1; T2 = Term2;
    deref_(T1,B1);deref_(T2,B2);
```

Wenn einer der Terme T_1 oder T_2 eine ungebundene Variable ist, so erfolgt die Unifizierung durch Binden des anderen Terms an diese Variable.

```
if(name(T1) == UNBOUNDT)
    BIND(T1,T2,B2);
else if(name(T2) == UNBOUNDT)
    BIND(T2,T1,B1);
```

Wenn keiner der beiden Terme eine ungebundene Variable ist, müssen ihre `name()`-Komponenten übereinstimmen:

```
else if(name(T1) != name(T2))
    goto fail;
```

Wenn die Terme Zahlen repräsentieren sind sie genau dann unifizierbar, wenn ihre numerischen Werte übereinstimmen:

```
else if(name(T1) == INTT) /* name(T2) == INTT */
{
    if(ival(T1) != ival(T2))
        goto fail;
}
```

In allen anderen Fällen handelt es sich um einen strukturierten Term, und die Argumente müssen unifiziert werden:

```
else if(!UNIFY(arity(name(T1)),son(T1),son(T2),B1,B2))
    goto fail;
}
```

Der Interpreterkern

Alle Unifikationsschritte waren erfolgreich:

```
return true;
```

Wenn ein Teilterm nicht unifizierbar war, müssen die Variablenbindungen wieder gelöst und der Term-Stack zurückgesetzt werden

```
fail:
    rel_trail_stack( TRAIL_TOP );
    rel_term_stack( TERM_TOP );
    return false;
}
```

Die folgende Funktion *BIND()* führt die eigentliche Variablenbindung aus.

V ist die zu bindende Variabel (*name(V) == UNBOUNDT*); *T* ist der an *V* zu bindende Term; und *B* ist der Term, der den Anfang der lokalen Variablen von *T* im Termstack darstellt. Eine Variable wird nur dann im Trail aufgezeichnet, wenn sie älter als der Choicepoint ist. Variablen die jünger als der Choicepoint sind, werden beim Backtracking automatisch durch Rücksetzen des Term-Stacks entfernt.

```
#define trailvar(V) if(is_older_term(V,base(CHOICEPOINT)))\
                        trailterm(V);\
                    else /* empty */;

LOCAL void BIND(V,T,B)
    TERM V,T,B;
{
```

Wenn es sich auch bei *T* um eine ungebundene Variable handelt, muß die Entscheidung getroffen werden, in welcher Richtung die Bindung auszuführen ist. Dabei erhält immer die "jüngere" Variable die ältere als Wert:

```
if(name(T) == UNBOUNDT)
{
    if(T == V) /* T und V sind trivial gleich */
        return;
    if(is_older_term(V,T))
    {
        /* Variable binden */
        name(T) = VART;
        val(T) = V;
        /* Variable im Trail-Stack aufzeichnen */
        trailvar(T);
    }
}
```

Der Interpreterkern

```
else /* is_older_term(T,V) */
{
    /* Variable binden */
    name(V) = VART;
    val(V) = T;
    /* Variable im Trail-Stack aufzeichnen */
    trailvar(V);
}
}
```

Wenn der Term T eine Zahl repräsentiert, so wird V einfach zu einer Zahl mit demselben numerischen Wert:

```
else if(name(T) == INTT)
{
    name(V) = INTT;
    ival(V) = ival(T);
    trailvar(V);
}
```

In jedem anderen Falle wird $name(V)$ auf den Wert von $name(T)$ gesetzt, und anschließend $son(V)$ gesetzt. Wenn T ein Heapterm ist, muß er in den Stack kopiert werden, damit die in ihm enthaltenen Variablen unabhängig von der aktuelle Umgebung (dem Berechnungszustand) werden:

```
else
{
    /* V im Trailstack aufzeichnen */
    trailvar(V);
    name(V) = name(T);

    if(is_heapterm(T))
    {
        son(V) = copy_block(son(T),arity(name(T)),B);
    }
    else
        son(V) = son(T);
}
```

Die folgende Funktion `copy_block()` kopiert den als Argument angegebenen Termblock in den Stack.

Der Interpreterkern

```
TERM copy_block(T,ar,B)
  TERM T;
  cardinal ar;
  TERM B;
{
  TERM Result;
  TERM Tmp;

  Result = Tmp = stackterm(ar);
  while(ar-->0)
  {
    TERM TT;
    TT = T ;
    deref_(TT,B);
    if((name(Tmp) = name(TT)) == UNBOUNDT)
    {
      name(Tmp) = VART;
      val(Tmp) = TT;
    }
    else if(name(TT) == INTT)
      ival(Tmp) = ival(TT);
    else
      son(Tmp) = copy_block(son(TT),arity(name(TT)),B);
    next_br(T);
    next_br(Tmp);
  }
  return Result;
}
```

3.2 Der Ableitungsmechanismus von HU-Prolog

Nachdem die Unifizierung als wesentliche Grundfunktion eines Prologsystems beschrieben wurde, soll jetzt näher auf den eigentlichen Ableitungsmechanismus eingegangen werden.

Wie schon erwähnt, nutzt HU-Prolog 3 Stacks. Da wäre zum einen der Term-Stack zu nennen. Er enthält sowohl die lokalen Variablen der sich in Abarbeitung befindlichen Klauseln, als auch Terme, die während der Prologarbeitung sonst entstehen. In anderen Prologsystemen ist dieser Stack in der Regel zweigeteilt. Zum einen gibt es den sogenannten lokalen-Stack, welcher die lokalen Variablen der Klauseln aufnimmt, zum anderen den globalen-Stack oder auch copy-Stack. Dieser enthält die während der Abarbeitung entstehenden Terme.

Dadurch das in HU-Prolog diese beiden Stacks zu einem zusammengefaßt wurden, ergibt sich eine effizientere Darstellung der Datenstrukturen und der Algorithmen. Ein Nachteil dieser Vorgehensweise ist, daß bekannte Optimierungen (z.B LCO und Tail-Rekursion) nicht mehr zu implementieren sind.

Der Interpreterkern

Des weiteren benutzt HU-Prolog einen Environment-Stack. In diesem werden die zur Abarbeitung des Programmes nötigen Verwaltungsinformationen abgelegt. Dies sind unter anderem:

- der aktuelle Call und sein Environment
- die Klausel, deren Abarbeitung durch diesen Aufruf ausgelöst worden ist
- ein Verweis in den Term-Stack, der den Anfang der lokalen Variablen zur zu bearbeitenden Klausel liefert
- der Stand der anderen Stacks zum Zeitpunkt des Aufrufs

Der dritte wesentliche in HU-Prolog verwendete Stack ist der Trail-Stack. In ihm werden alle Variablenbindungen vermerkt, die im Laufe des Backtrackings wieder rückgängig gemacht werden müssen.

Der eigentliche Abarbeitungsmechanismus wird in HU-Prolog von der Funktion *EXECUTE()* realisiert. Durch diese Funktion werden die Stacks entsprechend manipuliert. *EXECUTE()* benutzt die Funktion *UNIFY()*, um den jeweils aktuellen Aufruf mit dem Kopf einer entsprechenden Klausel zu unifizieren. Durch die Funktion *CallEvalPred()* werden die Build-In Prädikate behandelt.

Die Funktion *EXECUTE()* ist einer der komplizierteren Funktionen des Interpreters. Deswegen soll sie hier in 3 Schritten beschrieben werden.

3.2.1 Das Grundgerüst von EXECUTE()

Als erstes soll eine stark abgerüstete Version von *EXECUTE()* beschrieben werden. Sie erlaubt lediglich die Abarbeitung von nutzerdefinierten Prologprädikaten und von nichtbacktrackbaren Build-In Prädikaten.

EXECUTE() wird mit zwei Parametern, *CALLP* und *CALLENV*, aufgerufen. Diese werden auch innerhalb der Funktion als lokale Variablen genutzt. *CALLP* verweist dabei immer auf den aktuellen Aufruf darstellenden Term, *CALLENV* ist das zugehörige Environment. Die globale Variable *CHOICEPOINT* verweist immer auf das Environment, bei dem im Falle des Backtrackings fortgesetzt werden soll. Es ist dafür zu sorgen, daß diese Variable vor dem Aufruf von *EXECUTE()* einen definierten und sinnvollen Wert hat. Ansonsten gibt es in *EXECUTE()* noch drei lokale Variablen.

Der Interpreterkern

Dies sind:

```
CLAUSE CP;
ENV ENVP;
ENV BASEENV;
```

Dabei bezeichnet *CP* die nächste zu versuchende Klausel und *ENVP* eine temporäre Hilfsvariable. *BASEENV* bezeichnet den Stand des Environmentstacks zum Zeitpunkt des Aufrufes von *EXECUTE()*.

```
GLOBAL boolean EXECUTE (CALLP,CALLENV)
    TERM CALLP;
    ENV CALLENV;
{
    CLAUSE CP;
    ENV BASEENV,ENVP;
    ATOM A;

    BASEENV = mark_env_stack();

CALLQ:
    TRACE("call:",CALLP,CALLENV);
```

Diese Marke kennzeichnet den Aufruf eines Prädikates. Beim Sprung an diese Marke muß *CALLP* den aktuellen Aufruf und *CALLENV* das zugehörige Environment beinhalten. Der Aufruf der Funktion *TRACE()* erzeugt eventuell die Traceausschriften des Interpreters. Sie ist in diesen ersten Entwurf aufgenommen worden, um den mit HU-Prolog vertrauten Leser die Orientierung zu erleichtern. Es folgt ein Sprungverteiler in Abhängigkeit vom Typ des Aufrufes.

```
switch(class(A = name(CALLP)))
{
```

Wenn es sich um ein normales Prologprädikat handelt und es Klauseln für dieses Prädikat gibt dann wird *CP* auf die erste zu versuchende Klausel für dieses Prädikat gesetzt, und an der Marke *PROCQ* fortgesetzt. Wenn es keine Klausel gibt, wird Backtracking ausgelöst.

```
    case NORMP:
        if ((CP= clause(A)) != NIL_CLAUSE)
            goto PROCQ;
        else
            goto FAILQ;
```

Wenn es sich um ein Build-In Prädikat. handelt, wird die Funktion *CallEvalPred()* aufgerufen. Anhand ihres Resultates

Der Interpreterkern

wird entschieden, ob der Aufruf erfolgreich war (Fortsetzung bei *RETURNQ*) oder ob Backtracking ausgelöst werden muß.

```
case EVALP:  
    if(CALLEVALPRED(CALLP,CALLENV));  
        goto RETURNQ;  
    else  
        goto FAILQ;
```

Andere Fälle sind in dieser Version nicht implementiert.

```
default:  
    ; /* empty */  
}
```

PROCQ:

Wenn diese Marke angesprungen wird, enthält *CALLP* den aktuellen Call, *CALLENV* das aktuelle Environment und *CP* die nächste zu versuchende Klausel. Jetzt muß eine Klausel gefunden werden, deren Kopf mit *CALLP* unifizierbar ist. Für diese Klausel muß ein neues Environment angelegt werden. Anschließend wird das erste Goal im Klauselkörper das aktuelle Abarbeitungsziel.

```
/*Anlegen und Initialisieren des Environments*/  
ENVP = newenv();  
call(ENVP)          = CALLP;  
env(ENVP)           = CALLENV;  
choice(ENVP)        = CHOICEPOINT;  
trail(ENVP)         = mark_trail_stack();  
termtop(ENVP)       = mark_term_stack();  
atomtop(ENVP)       = mark_atom_stack();  
stringtop(ENVP)     = mark_string_stack();
```

Jetzt muß eine Klausel gesucht werden, deren Kopf mit *CALLP* unifizierbar ist. Wenn noch weitere Klauseln existieren muß dieses Environment neuer *CHOICEPOINT* werden.

```
for(;;)  
{  
    if (nextcl(CP) != NIL_CLAUSE)  
        CHOICEPOINT= ENVP;  
    /*lokale Variablen fuer CP anlegen*/  
    base(ENVP) = stackvar(nvars(CP));  
  
    /* Versuch der Unifizierung */  
    if(UNIFY(1,CALLP,head(CP),base(CALLENV),base(ENVP)))  
        goto unified;
```

Wenn die Unifizierung nicht erfolgreich war, wird

Der Interpreterkern

CHOICEPOINT wieder auf seinen alten Stand gesetzt, und der für die lokalen Variablen reservierte Speicherplatz wieder freigegeben. Wenn keine weiter Klausel mehr existiert, wird Backtracking ausgelöst, ansonsten wird die nächste Klausel versucht.

```
CHOICEPOINT = choice(ENVP);
rel_term_stack(base(ENVP));

if((C P = nextcl(CL)) == NIL_CLAUSE)
    goto FAILQ;
}
```

Wenn die Unifizierung erfolgreich war, wird noch das Environment vervollständigt. Anschließend wird der Klauselkörper der gefundenen Klausel neues Abarbeitungsziel. Wenn dieser leer ist, gilt die Klausel als erfolgreich.

```
unified:
rule(ENVP)=CP;
if(name(body(CP)) != NIL_ATOM)
{
    CALLENV= ENVP;
    CALLP=body(CP);
    goto CALLQ;
}
/* goto RETURNQ; */

RETURNQ:
TRACE("proved:",CALLP,CALLENV);
```

Beim Erreichen der Marke *RETURNQ* verweist *CALLP* auf einen erfolgreich abgearbeiteten Aufruf und *CALLENV* auf das zugehörige Environment. Wenn *CALLENV* älter als *BASEENV* ist, ist damit der als Argument von *EXECUTE()* angegebene Aufruf erfolgreich abgearbeitet. In jedem anderen Fall wird die Abarbeitung mit *brother(CALLP)* fortgesetzt. (Dies ist der nächste Aufruf in der Klausel) Wenn das Klauselende erreicht ist, wird der die Abarbeitung dieser Klausel auslösende Aufruf erfolgreich.

```
if( is_older_env(CALLENV,BASEENV) )
    return true;
if(name(next_br(CALLP)) != NIL_ATOM)
    goto CALLQ;
CALLP = call(CALLENV);
CALLENV = env(CALLENV);
goto RETURNQ;
```

Der Interpreterkern

FAILQ:

```
TRACE( "fail:", CALLP, CALLENV);
```

Diese Marke wird beim Backtracking erreicht. *CALLP* enthält den fehlgeschlagenen Aufruf und *CHOICEPOINT* enthält das Environment bei dem eine alternative Lösung zu suchen ist. Wenn *CHOICEPOINT* älter als *BASEENV* ist, wurde er nicht in diesem Aufruf der Funktion *EXECUTE()* angelegt. Es gibt also keine Möglichkeit des Backtrackings. *EXECUTE()* endet mit *false*. Sonst werden die im *CHOICEPOINT* abgespeicherten Werte restauriert. Beim Restaurieren des Trail-Stacks werden dabei die im Trail vermerkten Variablenbindungen gelöst

```
if( is_older(CHOICEPOINT,BASEENV) )
    return false;

CALLP = call(CHOICEPOINT);
CALLENV = env(CHOICEPOINT);
CP = rule(CHOICEPOINT);
rel_trail_stack(trail(CHOICEPOINT));
rel_atom_stack(atomtop(CHOICEPOINT));
rel_string_stack(stringtop(CHOICEPOINT));
rel_term_stack(termtop(CHOICEPOINT));
rel_env_stack(CHOICEPOINT);

CHOICEPOINT= choice(CH);
```

Jetzt muß eine alternative Klausel gesucht werden. Wenn es solche nicht mehr gibt, wird wiederum Backtracking ausgelöst.

```
if((CP = nextcl(CP)) == NIL_CLAUSE)
    goto FAILQ;

TRACE( "redo", CALLP, CALLENV);

goto PROCQ;
}
```

3.2.2 Eine vollständige Version

Nachdem nun die grundlegende Arbeitsweise von *EXECUTE()* dargestellt wurde, soll eine erweiterte Version beschrieben werden. Zur ersten Version gibt es folgende Ergänzungen:

- Es ist ein vollständiger Sprungverteiler für den Zustand *CALLQ* implementiert.
- Daraus ergibt sich auch eine Erweiterung des Zustandes *FAILQ* für den Fall von backtrackbaren Build-In-Prädikaten.

Der Interpreterkern

Ehe nun die Funktion selbst beschrieben wird, noch einige Bemerkungen zu den backtrackbaren Build-In Prädikaten. Ein backtrackbares Build-In wird in HU-Prolog wie ein normales Build-In durch eine C-Funktion realisiert. Damit diese beim Backtracking erneut aufgerufen wird, wird für jeden Aufruf eines *BTEVALP* ein Choicepoint angelegt. Das einzige Problem ist noch, daß einige Backtrackbare Build-In Prädikate die Information brauchen, ob sie zum ersten Mal aufgerufen werden, oder im Backtracking erreicht werden; andere Prädikate (wie z.B. *clause/2*) müssen sich einen Pointer in die Datenbasis merken - es muß also Information über das Ende der C-Funktion hinaus gerettet werden. Dazu wurde in HU-Prolog ein BacktrackControlTerm eingeführt.

```
GLOBAL TERM BCT;
```

Dieser Variable kann von der das *BTEVALP* realisierenden C-Funktion ausgewertet werden. Beim ersten Aufruf hat diese Variable den Wert *NIL_TERM*. Die C-Funktion kann diesen Wert ändern. Es ist garantiert, daß die ein *BTEVAL* realisierende C-Funktion im Backtrackfall genau denselben Wert der Variablen *BCT* vorfindet, der beim letztmaligen Verlassen dieser Funktion vorlag. Nun zur Funktion selbst.

```
GLOBAL boolean EXECUTE (CALLP,CALLENV)
    TERM CALLP;
    ENV CALLENV;
{
    CLAUSE CP;
    ENV BASEENV,ENVP;
    ATOM A;

    BASEENV = mark_env_stack();

CALLQ:
    TRACE("call:",CALLP,CALLENV);

    switch(class(A = name(CALLP)))
    {
        case NORMP:
            if ((CP= clause(A)) != NIL_CLAUSE)
                goto PROCQ;
            else
                goto FAILQ;

        case EVALP:
            if(CALLEVALPRED(CALLP,CALLENV) != false);
                goto RETURNQ;
            else
                goto FAILQ;
```

Der Interpreterkern

```
case BTEVALP:  
    BCT = NIL_TERM;
```

Ein backtrackbares Build-In Prädikat. Beim ersten Aufruf wird der *BCT* initialisiert. Für die backtrackbaren Build-In's muß ein Choicepoint angelegt werden. Danach wird ganz normal die Funktion *CallEvalPred()* aufgerufen. Endet sie erfolgreich und ist *BCT* != *NIL_TERM* so möchte das Prädikat im Backtracking nochmal gerufen werden. Der Wert von *BCT* wird im Environment vermerkt, um beim Backtracking restauriert werden zu können. Wenn *BCT* == *NIL_TERM* möchte das Prädikat im Backtracking nich mehr gerufen werden und der Choicepoint kann auf den alten Wert gesetzt werden.

```
REDOEVALQ:  
    ENVP = newenv();  
    call(ENVP)           = CALLP;  
    env(ENVP)            = CALLENV;  
    choice(ENVP)         = CHOICEPOINT;  
    termtop(ENVP)        = mark_term_stack();  
    trail(ENVP)          = mark_trail_stack();  
    atomtop(ENVP)        = mark_atom_stack();  
    stringtop(ENVP)      = mark_string_stack();  
    CHOICEPOINT          = ENVP;  
  
    if(CALLEVALPRED(CALLP,CALLENV))  
    {  
        if(BCT != NIL_TERM)  
            bct(ENVP) = BCT;  
        else  
            CHOICEPOINT = choice(ENVP);  
        goto RETURNQ;  
    }  
  
    CHOICEPOINT = choice(ENVP);  
    goto FAILQ;  
  
case GOTOP:
```

Hier wird, wie im Kapitel 2.4 beschrieben, die normale Klauselstruktur durchbrochen und bei *son(CALLP)* fortgesetzt. Dieser Mechanismus wird im wesentlichen zum temporären Einfügen von Code zur Realisierung der Interruptbehandlung und des Prädikates *unknown/1* (vgl. Kapitel 3.2.3) sowie des Prädikates *:=/2* (vgl. Kapitel 4.3.5) benutzt.

```
CALLP = son(CALLP);  
if(CALLP != NIL_TERM && name(CALLP) != NIL_ATOM)  
    goto CALLQ;  
goto RETURNQ;
```

Der Interpreterkern

case CUTP:

Sonderbehandlung des Prädikate `!/0`. Der Choicepoint wird eventuell zurückgesetzt. Dabei ist zu beachten, daß die Klauseln für `,/2` und `;/2` transparent bezüglich `!/0` sind.

```
ENVP = CALLENV;
CP = rule(ENVP);
while(!is_older_env(ENVP, BASEENV) &&
      ( CP == ANDG || CP == OR1G ||
        CP == OR2G || CP == NIL_CLAUSE))
{
    ENVP = env(ENVP);
    CP = rule(ENVP);
}
CHOICEPOINT = choice(ENVP);
goto RETURNQ;
```

case ARITHP:

Sonderbehandlung für von der funktionalen Arithmetik benutzten Prädikate. Sie werden immer erfolgreich, generieren aber eventuell neue Aufrufe; ändern also die Variable `CALLP`. (Näheres dazu im Kapitel 4.3.5)

```
CALLP = DOEVAL(CALLP, CALLENV);
goto RETURNQ;
```

case VARP:

Behandlung von Variablen als Aufruf. Die Variablen werden dereferenziert. Wenn es dann immer noch kein vernünftiger Aufruf ist, wird ein Fehler ausgelöst. Sonst wird `CALLP` entsprechend gändert.

```
TMP = CALLP;
deref_(TMP, base(CALLENV));
if(! normatom(name(CALLP)))
    ABORT(CALLE);
if(name(next_brother(TMP)) != NIL_ATOM)
    CALLP = mk2sons(name(CALLP), son(CALLP),
                     GOTOT, TMP);
else
    CALLP = mk2sons(name(CALLP), son(CALLP),
                     NIL_ATOM, NIL_TERM);
goto CALLQ;
```

Der Interpreterkern

```
NUMBP:
    ABORT(CALLE);
default:
    /* Andere Varianten duerfen nicht auftreten */
    SYSTEMERROR("EXECUTE");
}

PROCQ:
ENVP = newenv();
call(ENVP)          = CALLP;
env(ENVP)           = CALLENV;
choice(ENVP)         = CHOICEPOINT;
termtop(ENVP)        = mark_term_stack();
trail(ENVP)          = mark_trail_stack();
atomtop(ENVP)        = mark_atom_stack();
stringtop(ENVP)       = mark_string_stack();

for(;;)
{
    if(nextcl(CP) != NIL_CLAUSE)
        CHOICEPOINT= ENVP;

    base(ENVP) = stackvars(nvars(CP));

    /* Versuch der Unifizierung */
    if(UNIFY(1,CALLP,head(CP),base(CALLENV),base(ENVP)))
        goto unified;

    CHOICEPOINT = choice(ENVP);
    rel_term_stack(termtop(ENVP));

    if((CP = nextcl(CP)) == NIL_CLAUSE)
        goto FAILQ;
}

unified:
rule(ENVP)=CP;
if(name(body(CP)) != NIL_ATOM)
{
    CALLENV= ENVP;
    CALLP=body(CP);
    goto CALLQ;
}
goto RETURNQ;
```

Der Interpreterkern

```
RETURNQ:
    TRACE( "proved:", CALLP, CALLENV );
    if( is_older(CALLENV, BASEENV) )
        return true;
    if(name(next_br(CALLP)) != NIL_ATOM)
        goto CALLQ;
    CALLP = call(CALLENV);
    CALLENV = env(CALLENV);
    goto RETURNQ;

FAILQ:
    TRACE( "fail:", CALLP, CALLENV );
    if( is_older(CHOICEPOINT, BASEENV) )
        return false;
    CALLP = call(CHOICEPOINT);
    CALLENV = env(CHOICEPOINT);
    CP = rule(CHOICEPOINT);
    BCT = bct(CHOICEPOINT);
    rel_trail_stack(trail(CHOICEPOINT));
    rel_atom_stack(atomtop(CHOICEPOINT));
    rel_string_stack(stringtop(CHOICEPOINT));
    rel_term_stack(termtop(CHOICEPOINT));
    rel_env_stack(CHOICEPOINT);
    CHOICEPOINT= choice(CH);

    if(class(name(CALLP)) == BTEVALP)
    {
        TRACE( "redo", CALLP, CALLENV );
        goto REDOEVALQ;
    }

    if((CP = nextcl(CP)) == NIL_CLAUSE)
        goto FAILQ;

    TRACE( "redo", CALLP, CALLENV );
    goto PROCQ;
}
```

3.2.3 Die endgültige Version

Die dritte und vollständige Version entspricht genau der im Interpreter benutzten. Zur zweiten beschriebenen Version gibt es folgende Unterschiede:

- Das HU-Prolog Prädikat *unknown/1* wird realisiert, das heißt, wenn zu einem normalen Prädikat keine Klauseln vorhanden sind, wird stattdessen das Prädikat *unknown/1* aufgerufen. (vgl. HU-Prolog Sprachbeschreibung [1]) Das ist folgendermaßen realisiert:

Der Interpreterkern

```
switch(class(A = name(CALLP)))
{
```

Wenn es sich um ein normales Prologprädikat handelt, und es Klauseln dazu gibt, wird wie bisher verfahren.

```
    case NORMP:
        if((CP = clause(A)) != NIL_CLAUSE)
            goto PROCQ;
```

Wenn es keine Klauseln gibt, aber das Prädikat *unknown/1* vom Nutzer definiert wurde, so wird dieses anstelle des ursprünglichen Aufrufes verwendet. Das Argument von *unknown/1* ist dabei der ursprüngliche Aufruf.

```
    if(clause(UNKNOWN_1) != NIL_CLAUSE)
    {
        TERM T;
        T = mkfreevar();
        (void)UNIFY(1,CALLP,T,
                    base(CALLENV),NIL_TERM);
        next_br(CALLP);
        if(name(CALLP) != NIL_ATOM)
            CALLP = mk2sons(UNKNOWN_1,T,
                            GOTOT,CALLP);
        else
            CALLP = mk2sons(UNKNOWN_1,T,
                            NIL_ATOM,NIL_TERM);
        goto CALLQ;
    }
```

In allen anderen Fällen wird Backtracking ausgelöst.

```
    goto FAILQ;
    /* weiter wie gehabt */
}
```

- Eine analoge Behandlung von *ansyncron* ausgelösten Interrupts (Prädikat *interrupt/0*) sowie von abfangbaren Fehlern (Prädikat *error/2* - vgl. Kapitel 4.2 und 4.3) wird realisiert.
- Es wird gesichert, daß der Ableitungsmechanismus auch merkt, wenn inzwischen Klauseln gelöscht wurden.
- Beim Suchen von Klauseln im Zweig *PROCQ* wird ein Indexing über das erste Argument vorgenommen (d.h. wenn das erste Argument des Aufrufes keine Variable ist, werden nur noch die Klauseln in die engere Wahl gezogen,

Der Interpreterkern

bei denen das erste Argument des Kopfes der Klausel entweder eine Variable ist, oder mit dem ersten Argument des Aufrufes übereinstimmt). Dadurch wird einerseits die Abarbeitungszeit beschleunigt (es muß nicht jedesmal eine Unifizierung versucht werden), andererseits wird eventuell Speicherplatz gespart, da eventuell kein Choicenode angelegt werden braucht.

- Die Funktion *EXECUTE()* wird nochmals optimiert.

Den Quelltext dieser endgültigen Fassung von *EXECUTE()* entnehme man dem File *exec.c*

4. Realisierung der Build-In Prädikate

Ein wesentlicher Teil jeder Prologimplementation sind die zur Verfügung stehenden Build-in Prädikate. Dieses Kapitel ist für die Leser gedacht, die das HU-Prolog System um eigene Build-in Prädikate erweitern wollen. Die dazu notwendigen Informationen und Beispiellösungen werden im folgenden beschrieben.

4.1 Schnittstellengestaltung

Wie man bei der Beschreibung der Funktion *EXECUTE()* gesehen hat, führt ein Aufruf der Build-In Prädikate auf einen Aufruf der Funktion *CalleEvalPred()*. Diese Funktion wertet ihre Argumente aus, bereitet die Argumente des aufgerufenen Build-In Prädikates auf und ruft anschließend eine Funktion, die das entsprechende Build-In realisiert.

Die so aufgerufende Funktion kann sich auf folgende Konventionen verlassen.

- Die globale Variable *ENV E*; enthält das Environment des Aufrufes, die Variable *TERM BE*; den Wert *base(E)*. Dieses Environment erhält einen Bedeutung beim Dereferenzieren von in den Argumenten des Build-In's vorkommenden Variablen.
- Die Argumente des Build-In's werden in den globalen Variablen *TERM A0,A1,A2*; abgelegt. Es ist garantiert, daß diese Argumente bereits dereferenziert sind.
Bemerkung: Es gibt in HU-Prolog maximal dreistellige Build-In Prädikate
- Des weiteren beinhalten die beiden globalen Variablen *TERM aCALLP*; und *ENV aCALLENV*; den Aufruf und das Environment des Build-In's.

4.2 Fehlerbehandlung

Bevor die Funktion *CalleEvalPred()* erläutert wird, einige Bemerkungen zur Fehlerbehandlung. Intern werden von HU-Prolog drei Fehlerarten unterschieden.

- Von Build-In Prädikaten oder Hilfsfunktionen festgestellte Fehler, die Fehler im Sinne von Prolog sind. (also Fehler im Prologprogramm) Beim Erkennen solcher Fehler wird die Funktion
ERROR(Fehlernummer)
aufgerufen.

Realisierung der Build-In Prädikate

Diese Fehler können vom Nutzer mit Hilfe des Prologprädikates `error/2` abgefangen werden.

- Fehler, die aufgrund der Einschränkungen der konkreten Prologimplementation auftreten. Dies sind in der Regel Speicherplatzprobleme des Prologsystems. Beim Erkennen dieser Fehler wird die Funktion

```
ABORT( Fehlernummner )  
aufgerufen.
```

- Die dritte Art von Fehlern, sind Fehler des Prologsystems selbst. Diese sollten eigentlich nie auftreten. Wenn im Interpreter jedoch Inkonsistenzen festgestellt werden, führt dies zum Aufruf der Funktion

```
SYSTEMERROR( Fehlernachricht )
```

Eine Weiterarbeit ist in diesem Fall nicht sinnvoll; das Prologsystem wird nach Information des Nutzers verlassen.

Alle drei Funktionen haben eins gemeinsam. Sie kehren nicht an ihre Aufrufstelle zurück - die Programmabarbeitung wird an anderer Stelle fortgesetzt.

Wie ist dies nun realisiert?

Die C-Bibliothek stellt mit den Funktionen `setjmp()` und `longjmp()` die Möglichkeit von globalen Sprüngen aus Prozeduren auf einen alten Abarbeitungsstand zur Verfügung. Diese werden benutzt, um in Fehlerfall in eine umgebende Funktion zurückzuspringen. Dazu werden zwei globale Variablen definiert, die die Sprungadressen beinhalten. Diese werden von den Funktionen `ERROR()` und `ABORT()` verwaltet.

```
#include <setjmp.h>  
  
GLOBAL jmp_buf Abort_Level;  
GLOBAL jmp_buf Error_Level;  
  
GLOBAL ERROR( N )  
    ERRORT N;  
{  
    ERRORFLAG = N;  
    longjmp(Error_Level,1);  
}
```

Realisierung der Build-In Prädikate

```
GLOBAL ABORT( N )
    ERRORT N;
{
    /* Fehlerausschrift fuer den Nutzer */
    longjmp(Abort_Level,1);
}
```

Die beiden globalen Variablen *Abort_Level* und *Error_Level* werden in der Funktion *main()* (siehe Kapitel 5) initialisiert. Dadurch ist gewährleistet, daß im Fehlerfall standardmäßig nach *main()* zurückgesprungen wird.

Die globale Variable *Error_Level* kann jedoch von anderen Funktionen umdefiniert werden. Dies macht zum Beispiel die in diesem Kapitel beschriebene Funktion *CallEvalPred()*. Dadurch kann der Aufruf von *ERROR()* auf einem bestimmten Niveau abgefangen werden.

4.3 Die Funktion CallEvalPred()

Eine erste Realisierung der Funktion *CallEvalPred()* könnte wie folgt aussehen:

```
boolean CallEvalPred(Callp, Callenv)
    TERM Callp;
    ENV Callenv;
{
```

Wenn nicht anders festgelegt, endet ein Build-In Aufruf erfolgreich:

```
boolean Result = true;
```

Die globalen Variablen werden initialisiert:

```
E      = Callenv;
BE     = base(Callenv);
aCallp = Callp;
aCallenv = Callenv;

switch(arity(name(Callp)))
{
    case 3: A2 = arg3(Callp);
    case 2: A1 = arg2(Callp);
    case 1: A0 = arg1(Callp);
    case 0: break;
    default: SystemError("Callevalpred.1");
}
```

Es folgt ein Sprungverteiler über den Namen des Prädikates. Hier nur zwei Prädikate zur Demonstration:

Realisierung der Build-In Prädikate

```
switch(name(Callp))
{
    /* . . . */
    case GET_0:      Result = DOGET0(); break;
    /* fuer Praedikate die nicht fehlschlagen */
    case TAB_1:      DOTAB(); break; /* Result == true */
    /* usw. */
    default: SystemError("Callevalpred.2");
}
return Result;
```

Diese hier angegebene erste Realisierung unterscheidet sich von der endgültigen Version in zwei wesentlichen Punkten:

Zum einen wird in HU-Prolog nicht jedes Build-In durch eine eigene C-Funktion realisiert. Aus Effektivitätsgründen kann das oben angegebene Aufrufschema geändert werden.

Beispiel:

Das Prädikat *nl/0* führt direkt auf den Aufruf einer I/O-Funktion:

```
case NL_0: ws("\n");break
```

Bei Prädikaten die ähnliche Semantik realisieren wird nur eine Funktion aufgerufen, welche über Parameter die Unterscheidung des abzuarbeitenden Prädikates erhält.

```
case DICT_1 : Result = DODICT(false);break;
case SDICT_1: Result = DODICT(true );break;
```

Weitere Optimierungen dieser Art sind der konkreten Implementation der Funktion *CallEvalPred()* im File *eval.c* zu entnehmen.

Der zweite wesentliche Unterschied betrifft die Behandlung von Fehlern in den Build-In Prädikaten. Wie oben beschrieben, führt der Aufruf der Funktion *ERROR()* zum Auslösen eines *longjmp(Error_Level)*. Diese Variable *Error_Level* wird von der Funktion *CallEvalPred()* temporär umgesetzt. Da das rekursive Aufrufen von *CallEvalPred()* über rekursive Aufrufe von *EXECUTE()* möglich sind, wird der alte Wert der Variablen *Error_Level* in eine lokalen Variable gerettet und vor Verlassen der Funktion wieder zurückgespeichert.

Beim Auftreten eines Fehlers während der Abarbeitung eines Build-In Prädikates kehrt die Steuerung also an die Funktion *CallEvalPred()* zurück. Dies ist wie folgt realisiert:

Realisierung der Build-In Prädikate

```
#include <setjmp.h>

boolean CallevalPred(Callp, Callenv)
    TERM Callp;
    ENV Callenv;
{
    boolean Result = true;
    jmp_buf old_error_level;

    E      = Callenv;
    BE     = base(Callenv);
    aCallp = Callp;
    aCallenv = Callenv;

    switch(arity(name(Callp)))
    {
        case 3: A2 = arg3(Callp);
        case 2: A1 = arg2(Callp);
        case 1: A0 = arg1(Callp);
        case 0: break;
        default: SystemError("Callevalpred.1");
    }
}
```

Die globale Variable *Error_Level* wird lokal gerettet. Da der zugrundeliegende Datentyp ein Array ist, wird dazu die Funktion *memcpy()* verwendet. Anschließend wird *Error_Level* neu gesetzt. Dabei endet *setjmp(Error_Level)* mit dem Funktionswert 0. Wenn im Verlaufe einer Fehlerbehandlung *longjmp(Error_Level)* gerufen wird, so wird die Programmabarbeitung an der Aufrufstelle von *setjmp()* fortgesetzt. In diesem Falle endet *setjmp()* mit einem Funktionswert ungleich 0 und der Sprungverteiler wird umgangen.

```
    memcpy(old_error_level, Error_Level, sizeof(jmp_buf));

    if(setjmp(Error_Level) == 0)
        switch(name(Callp))
        {
            /* Sprungverteiler - siehe oben */
        }
}
```

Vor Beendigung der Funktion muß die globale Variable *Error_Level* wieder restauriert werden.

```
    memcpy(Error_Level, Old_Error_Level, sizeof(jmp_buf));
    return Result;
}
```

Realisierung der Build-In Prädikate

4.4 Beispiele für die Realisierung von Build-In's

In diesem Kapitel soll die Implementation einiger ausgewählter Prädikate beschrieben werden. Dabei wird die Arbeit mit den Datenstrukturen dokumentiert. Des Weiteren soll der Leser in die Lage versetzt werden, anhand dieser Beispiele weitere Build-In's selbst zu implementieren.

4.4.1 Prädikate zur Termklassifizierung

Einfache Prädikate zur Termklassifizierung sind `atom/1`, `integer/1`, `real/1`, `compound/1` und `number/1`. Diese Klassifizierungen können direkt anhand der Komponente `name()` des zu klassifizierenden Terms vorgenommen werden. Diese erfolgt noch innerhalb der Funktion `CallEvalPred()`.

Beispiel:

```
/* Funktion CallEvalPred - switch Anweisung */
case INTEGER_1: Res = is_integer(A0);break;
case COMPOUND_1: Res = norm_atom(A0) && arity(name(A0));break;
/*
Dabei ist zum Beispiel is_integer() ein Makro der Form
#define is_integer(X) (name(X) == INTT || name(X) == LONGT)
*/
```

Weitere Möglichkeiten zur Termklassifizierung bilden die HU-Prolog Prädikate `list/1`, `string/1` und `ground/1`. Diese sind wie folgt implementiert:

```
case GROUND_1: Res = ground(A0);break;
case LIST_1: Res = is_list(A0, false);break;
case STRING_1: Res = is_list(A0, true);break;
```

Zuerst die Implementation von `is_list()`. Es wird geprüft, ob es sich bei dem ersten Argument um eine Liste handelt. Wenn `ascii_list` einen Wert ungleich `false` hat, wird zusätzlich überprüft, ob die Argumente der Liste ASCII Werte sind. Die Länge der Liste wird mitgezählt, um zyklische Liste zu erkennen.

Realisierung der Build-In Prädikate

```
boolean is_list(T,ascii_list)
    TERM T;
    boolean ascii_list;
{
    cardinal list_len = 0;

    while(name(T) == CONS_2)
    {
        if(ascii_list)
        {
            TERM TT ;
            TT = arg1(T);
            if((name(TT) != INTT) ||
               (ival(TT) < 0 ) || (ival(TT) > 255))
                return false;
        }
    }
}
```

Wenn die aktuelle Listenlänge größer ist, als die Anzahl der Terme im System, muß die Liste zyklisch sein. Ebenfalls keine korrekte Liste liegt vor, wenn das letzte Element nicht das Atom []/0 ist.

```
if(++list_len > MAX_TERMS)
    return false;
T = arg2(T); /* T = Restliste von T */
}

if(name(T) != NIL_0)
    return false;
return true;
}
```

Realisierung der Build-In Prädikate

Nun die Implementation von `ground/1`. Es wird hierbei rekursiv geprüft, ob der gegebene Term Variablen enthält.

```
boolean ground(T)
    TERM T;
{
    deref(T);

    if(name(T) == UNBOUNDT)
    {
        /* Variable gefunden - kein Groundterm ! */
        return false;
    }
}
```

Wenn es sich um einen zusammengesetzten Term handelt, werden die einzelnen Argumente überprüft.

```
else if(norm_atom(name(T)))
{
    cardinal ar;
    ar = arity(name(T));
    T = son(T);

    while(ar-->0)
    {
        if(!ground(T))
            return false;
        next_br(T);
    }
    return true;
}
```

Wenn es sich weder um eine Variable, noch um einen zusammengesetzten Term handelt, liegt ein Groundterm vor.

```
else
    return true;
}
```

4.4.2 Prädikate zur Termanalyse und Syntese

4.4.2.1 Das Prädikat `functor/3`

Das Prädikat `functor/3` nutzt die oben beschriebene standardisierte Schnittstelle in `CallEvalPred()`. Es wird die Funktion `DOFUNCTOR()` aufgerufen.

Realisierung der Build-In Prädikate

```
boolean DOFUNCION()
{
    ATOM A;
```

Wenn das erste Argument von `functor/3` ein Funktor ist, wird daraus der Name und die Stellenzahl ermittelt und mit dem zweiten bzw. dritten Argument unifiziert:

```
if(norm_atom(A = name(A0)))
    return UNI(A1,mkatom(LOOKATOM(A,0))) &&
           INTRES(A2,(int)arity(A));
```

Wenn es sich beim ersten Argument um eine Zahl handelt, wird diese wie ein 0-stelliger Funktor interpretiert:

```
else if(is_number(A0))
    return UNI(A0,A1) && INTRES(A2,0);
```

Sonst kann es sich nur um eine ungebundene Variable handeln. Aus dem zweiten und dritten Argument wird Name und Stellenzahl des zu bildenden Funktors entnommen:

```
else /* name(A0) == UNBOUNDT */
{
    cardinal ar;
```

Das dritte Argument muß eine ganze Zahl zwischen 0 und `MAXARITY` sein:

```
if(name(A2) != INTT || (ar = ival(A2)) < 0 ||
                           ar > MAXARITY)
    ARGERROR();
```

Das zweite Argument muß eine Zahl oder ein Atom sein:

```
if(is_number(A1))
{
    /* eine Zahl */
    if(ar != 0)
        ARGERROR();
    return UNI(A0,A1);
}
```

Realisierung der Build-In Prädikate

```
A = name(A1);
if(! norm_atom(A) && arity(A) != 0)
    ARGEROR();
return UNI(A0,mkfunc(LOOKATOM(A,ar),
                      stackvars(ar)));
}
```

4.4.2.2 Das Prädikat =.../2

```
boolean DOUNIV()
{
```

Das erste Argument von =.../2 ist entweder ein Funktor, eine Zahl oder eine ungebundene Variable. Wenn es ein normaler Term ist, wird aus diesem Term entsprechend der Semantik von =.../2 eine Liste generiert. Diese wird anschließend mit A1 unifiziert:

```
if(norm_atom(name(A0)))
{
    TERM Resultlist,T,Argterm;
    cardinal ar;
    ATOM A;

    A = name(A0);
    T = mk2sons(LOOKATOM(A,0),NIL_TERM,NIL_0,NIL_TERM);
    Resultlist = mkfunc(CONS_2,T);
    next_br(T);
```

Resultlist zeigt jetzt auf die ein-elementige Liste, die den Hauptfunktor von A0 enthält. T zeigt auf das Ende der Liste. Für jedes Argument des strukturierten Terms A0 wird die Liste verlängert. Argterm verweist im folgenden immer auf das aktuelle Argument von A0:

```
Argterm = son(A0);
for(ar = arity(A) ; ar > 0 ; --ar,next_br(Argterm))
{
    name(T) = CONS_2;
    son(T) = mk2sons(UNBOUNDT,NIL_TERM,
                     NIL_0,NIL_TERM);
    T = son(T);
    UNI(T,Argterm);
    next_br(T);
}
return UNI(Resultlist,A1);
}
```

Wenn das erste Argument eine Zahl ist, wird eine ein-

Realisierung der Build-In Prädikate

elementige Liste aufgebaut und mit A1 unifiziert:

```

else if(is_number(A0))
{
    TERM Resultlist;
    Resultlist = mkfunc(CONS_2,mk2sons(UNBOUNDT,NIL_TERM,
                                         NIL_0,NIL_TERM));
    UNI(A0,son(Resultlist));
    return UNI(Resultlist,A1);
}

```

Wenn das erste Argument eine ungebundene Variable ist, muß A1 auf eine Liste verweisen. Aus dieser wird dann der entsprechende Term konstruiert:

```

else /* name(A0) == UNBOUNDT */
{
    cardinal len;
    TERM T,Resultterm;
    ATOM A;
    /* Laenge der Liste ermitteln */
    for(len=0, T = A1 ; name(T) == CONS_2, len++<=MAXARITY;
        T = arg2(T));
        if(name(T) != NIL_0 // len--<1 // len > MAXARITY)
            ARGERROR();

    T = arg1(A1); /* erstes Listenelement */
    if(is_number(T))
    {
        if(len != 0)
            ARGERROR();
        return UNI(A0,T);
    }

    if(!norm_atom(A = name(T)) // arity(A) != 0)
        ARGERROR();
    Resultterm = mkfunc(LOOKATOM(A,len),stackterms(len));
}

```

Resultterm ist jetzt Term mit Variablen als Argument. Diese werden an die entsprechenden Listenelemente gebunden.

```

for(T = son(Resultterm),TT = son(arg2(A1)); len-->0;
    next_br(T),TT = son(br(TT)))
    UNI(T,TT);

    return UNI(A0,Resultterm);
}

```

Realisierung der Build-In Prädikate

4.4.3 Termvergleich

Hier soll auf die Klasse der @-Termvergleichsprädikate näher eingegangen werden. Die @-Prädikate münden alle auf die Funktion `compare()`, welche einen Wert kleiner, größer oder gleich 0 zurückgibt, je nachdem, ob das erste Argument kleiner, größer oder gleich dem zweiten Argument ist.

```
/* Ausschnitt aus CallEvalPred() */
case ALT_2: /* @< */ Result = (compare(A0,A1) < 0);
              break;
case ALE_2: /* @<= */ Result = (compare(A0,A1) <= 0);
              break;
/* und so weiter */
```

Die Funktion `compare()` ist folgendermaßen realisiert:

```
int compare(T1,T2)
TERM T1,T2;
{
    card ar1,ar2;
    STRING S1,S2;
    cardinal i;
    deref(T1); deref(T2);
```

Variablen sind kleiner als alle anderen Terme und untereinander gleich.

```
if(name(T1) == UNBOUNDT)
    return ((name(T2) == UNBOUNDT) ? 0 : -1);
if(name(T2) == UNBOUNDT)
    return 1;
```

Jetzt sind sowohl `T1` als auch `T2` keine Variablen. Zahlen sind kleiner als sonstige Terme, und untereinander nach ihrem numerischen Wert geordnet.

```
if(is_number(T1))
    if(is_number(T2))
    {
```

`T1` und `T2` repräsentieren Zahlen. Der Vergleich dieser Zahlen soll hier nicht weiter durchgeführt werden, da es sich bei 3 Sorten von Zahlen (`INTT`, `LONGT` und `REALT`) um eine größere Fallunterscheidung handelt.

```
        return /* -1, 0 oder 1 */ ;
    }
```

Realisierung der Build-In Prädikate

```
    else
        return -1;
    if(is_number(T2))
        return 1;
```

Jetzt handelt es sich bei T_1 und T_2 um normale strukturierte Terme. Es werden der Hauptfunktor und die Argumente verglichen:

```
S1 = ident(name(T1));
S2 = ident(name(T2));
/* Vergleich von S1 und S2 */
for(i = 0; stringchar(S1,i) == stringchar(S2,i);++i)
    if(stringchar(S1,i) == '\0')
        break;;
if(stringchar(S1,i) < stringchar(S2,i))
    return -1;
if(stringchar(S1,i) > stringchar(S2,i))
    return 1;
```

S_1 und S_2 sind gleich. Jetzt werden die Argumente von T_1 und T_2 paarweise verglichen. Beim ersten Unterschied steht das Resultat von `compare()` fest:

```
ar1 = arity(name(T1));
ar2 = arity(name(T2));
T1 = son(T1);
T2 = son(T2);

while(ar1 && ar2)
{
    int res;
    res = compare(T1,T2);
    if(res != 0)
        return res;
    next_br(T1); next_br(T2);
    --ar1; --ar2;
}
```

Mindestens einer der Terme hat keine weiteren Argumente mehr. Der Term der mehr Argumente hat ist als größer zu betrachten:

```
return (ar1 - ar2);
}
```

Realisierung der Build-In Prädikate

4.4.4 Listenverarbeitung

Eine wichtige Problemklasse bei der Benutzung von Prolog ist die Listenverarbeitung. HU-Prolog selbst stellt standardmäßig keine Prädikate zur Listenverarbeitung zur Verfügung. Trotzdem sollen hier mögliche Implementationen solcher Prädikate beschrieben werden, die dann der Nutzer selbst installieren kann. (siehe Kapitel 4.4.) Dabei werden die oben beschriebenen Konventionen für die Parameterübergabe eingehalten.

4.4.4.1 Eine Version von append/3

Hier soll eine Version des Prädikates `append/3` beschrieben werden. Dabei wird vorausgesetzt, daß das erste Argument eine Liste ist. Die zugehörige Implementation in HU-Prolog hat folgende Form:

```
append(List1,List2,Result) :-  
    (list(List1),!,app(List1,List2,Result))  
    ; abort  
    ).  
app([],List,List).  
app([Head : Tail], List , [Head : Resulttail]) :-  
    app(Tail,List,Resulttail).
```

Dies läßt sich in C implementieren, indem das erste Argument von `append/3` kopiert, und das die Liste beendende Atom `NIL_0` durch einen Verweis auf das zweite Argument ersetzt wird.

```
boolean DOAPPEND()  
{  
    TERM Result,Tail;  
  
    Result = Tail = mkatom(NIL_0);  
  
    while(name(A0) == CONS_2)  
    {  
        /* Die Liste wird um ein Element verlängert */  
        name(Tail) = CONS_2;  
        son(Tail) = mk2sons(UNBOUNDT,NIL_TERM,  
                           NIL_0,NIL_TERM);  
        (void)UNI(son(Tail),arg1(A0));  
  
        A0 = arg2(A0);  
        Tail = arg2(Tail);  
    }  
}
```

Wenn `name(A0) != NIL_0` ist, ist das erste Argument von `append()` keine reguläre Liste:

Realisierung der Build-In Prädikate

```
if(name(A0) != NIL_0)
    ERROR(ARGE);
```

Result ist jetzt eine absolute Kopie des ersten Argumentes von *append()*. In dieser Kopie wird jetzt das abschließende *NIL_0* durch das zweite Argument von *append()* ersetzt. Anschließend wird *Result* mit *A2* unifiziert.

```
name(Tail) = UNBOUND;
(void)UNI(Tail, A1);

return UNI(Result,A2);
}
```

In dieser eben beschriebenen Funktion wurde die Resultatsliste nach dem Top-Down-Prinzip aufgebaut. Im nächsten Beispiel erfolgt dieser Aufbau Bottom-Up.

Noch eine Bemerkung zu der oben beschriebenen Funktion. Was passiert, wenn das erste Argument von *append/3* eine zyklische Liste ist. Kann es eventuell passieren, dass in der while-Schleife kein Abbruch erfolgt?

Diese Frage sollte man sich bei allen Schleifen stellen, da das HU-Prolog-System nur an bestimmten Stellen unterbrochen werden kann. In dem oben beschriebenen Beispiel ist das kein Problem, da in jedem Schleifendurchgang Speicherplatz verbraucht wird. Bei einer zyklischen Liste erfolgt also irgendwann einmal ein Abbruch aus Speicherplatzmangel.

4.4.4.2 Umdrehen einer Liste

Hierbei handelt es sich um einen oft benötigten Algorithmus. Die Implementation erfolgt, indem aus den Elementen der als erstes Argument angegebenen Liste die Resultatsliste Bottom-Up aufgebaut wird.

```
boolean DOREVERSE()
{
    TERM Result;

    if(name(A0) == CONS_2)
    {
        /* Die umzudrehende Liste ist nicht leer */
        Result = mk2sons(UNBOUND, NIL_TERM, NIL_0, NIL_TERM);
        (void)UNI(Result, arg1(A0));
        A0 = arg2(A0);
```

Realisierung der Build-In Prädikate

```
while(name(A0) == CONS_2)
{
    Result = mk2sons(UNBOUNDT,NIL_TERM,CONS_2,Result);
    (void)UNI(Result,arg1(A0));
    A0 = arg2(A0);
}

/* Regulaeres Listenende pruefen */
if(name(A0) != NIL_0)
    Error(ARGE);

/* Liste vervollstaendigen */
Result = mfunc(CONS_2,Result);
return UNI(Result,A1);
}

else if(name(A0) == NIL_0)
    return UNI(A0,A1);

else
    Error(ARGE);
/*NOTREACHED*/
}
```

4.4.4.3 Ein Sortierproblem

Zum Abschluß des Kapitels über Listenverarbeitung noch ein in der Praxis häufig auftretenen Problem: das Sortieren einer Liste. Als Ordnungsfunktion sollen die Termvergleichsfunktionen (@</2, @==) herangezogen werden. Diese beruhen auf der im Kapitel 4.3.3 beschriebenen Funktion *compare()*. Das hier benutzte Sortierverfahren ist als "Sortieren durch Einfügen" bekannt.

```
boolean DOSORT()
{
    IMPORT int compare();
    TERM Result;

    Result = mkatom(NIL_0);
```

Jedes Element der in der globalen Variablen *A0* abgespeicherte Liste wird in die Liste *Result* eingesortiert.

Realisierung der Build-In Prädikate

```
while(name(A0) == CONS_2)
{
    TERM ARG1,T;
    ARG1 = arg1(A0); /* Effizienzgruende */

Es wird die bisher aufgebaute Liste Result entlanggegangen.
Dabei wird überprüft, wo eingefügt werden muß.

    T = Result;
    while( name(T)==CONS_2 &&
           compare(arg1(T),ARG1,MAXDEPTH) < 0)
        T = arg2(T);

ARG1 muß jetzt vor T eingefügt werden. Es wird ein neues
Listenelement an T angefügt, welches den bisherigen Wert von
T aufnimmt. Der neue Wert für T ist ARG1.

    son(T) = mk2sons(UNBOUNDT,NIL_TERM,name(T),son(T));
    UNI(son(T), ARG1);
    name(T) = CONS_2;

    A0=arg2(A0);
}

/* Test auf richtiges Listenende */
if(name(A0) != NIL_0)
    Error(ARGE);
/* und fertig */
return UNI(A1,Result);
}
```

4.4.5 Das Prädikat :=/2

In HU-Prolog wurden in Erweiterung der Prolog-typischen relationalen Programmierung auch Komponenten der funktionalen und prozeduralen Programmierung aufgenommen. (vgl. [3]) Diese werden durch das Prädikat :=/2 realisiert. Hier soll nur ein Ausschnitt der Möglichkeiten von :=/2 beschrieben und Varianten der Implementation aufgezeigt werden.

Die Abarbeitung von *Result := Expression* zerfällt logisch in zwei Phasen. Zuerst wird der Ausdruck *Expression* ausgewertet. Das Resultat der Auswertung wird anschließend *Result* zugewiesen. Die Implementation von :=/2 erfolgte deshalb in Prolog.

Realisierung der Build-In Prädikate

```
Result := Expression :-  
    $evaluate(Temp, Expression),  
    '$!', % eine andere Form des !/0 Prädikates  
    $dass(Result, Temp).
```

Das Prädikat `$dass/2` ist dabei ein ganz gewöhnliches Build-In Prädikat welches die Zuweisung erledigt und soll hier nicht näher betrachtet werden. Interessanter ist das Prädikat `$evaluate/2`. Während der Auswertung des Ausdrucks müssen eventuell globale Variablen oder Funktionsaufrufe ausgewertet werden. Dazu müssen Prolog-Aufrufe abgearbeitet werden. Der Interpreter (die Funktion `EXECUTE()`) muß also rekursiv gerufen werden. Dies ist zum Beispiel auch bei der Realisierung des Prädikates `consult/1` der Fall. Im Gegensatz zu `consult/2` kann die Rekursion bei `:=/2` sehr tief verschachtelt sein. Da dies sowohl zeitaufwendig ist, als auch eine größere Belastung für den C-Stack darstellt, ist der rekursive Aufruf von `EXECUTE()` hier keine günstige Lösung. Deshalb wurde ein anderer Weg gewählt.

Wenn die das Prädikat `$evaluate/2` realisierende Funktion `DOEVAL()` zur Auswertung des Ausdrucks weitere Prologaufrufe absetzen muß, modifiziert sie die lokale Variable `CALLP` der Funktion `EXECUTE()` - legt also neue Abarbeitungsziele fest. Damit `DOEVAL()` das tun kann, wird das Prädikat `$evaluate/2` in `EXECUTE()` gesondert behandelt. Diese Prädikate erhalten nicht das Attribut `EVALP` wie andere Build-In Prädikate, sondern das Attribut `ARITHP`. Ein Blick zur Implementation von `EXECUTE()` zeigt den Aufruf:

```
CALLP = DOEVAL(CALLP, CALLENV);
```

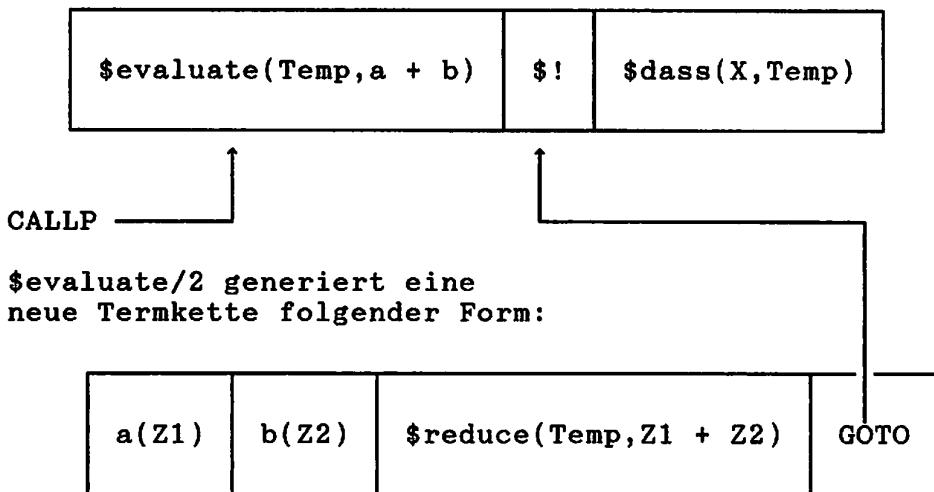
Diese Änderung der weiteren Aufrufziele erfolgt ohne Modifikation der Datenbasis. Ein Beispiel:

Der aktuelle Aufruf sei:

```
?- X := a + b.
```

Realisierung der Build-In Prädikate

Ich greife jetzt den Zustand der Variablen CALLP während der Abarbeitung von `:=/2` heraus.



und liefert diese als Resultat.

4.5 Das File user.c

In diesem Abschnitt wird der Frage nachgegangen, was der Nutzer tun muß, um neue Build-In Prädikate in das HU-Prolog System zu integrieren. Im Prinzip reichen die bisher gegebene Beschreibung der Datenstrukturen und der Build-In Prädikate aus, um neue Build-In's einzufügen. Dafür müßten jedoch in mehreren Files Änderungen vorgenommen werden. (mindestens in `atoms.h`, `atoms.c` und `eval.c`) Um dieses zu vermeiden, wurde das File `user.c` eingeführt in dem alle notwendigen Änderungen lokal durchgeführt werden können. (Es ist ein Werkzeug in Arbeit, das den Nutzer bei der Erstellung von `user.c` unterstützt. Leider gibt es davon im Moment keine lauffähige Version)

Im folgenden soll der prinzipielle Aufbau des Files `user.c` erläutert werden. Dabei werden zwei neue Prädikate definiert.

Als erstes werden zwei Makros vereinbart.

```
#define LENGTH_2           useratom(0)
#define LAST_2              useratom(1)
```

Die folgende Funktion `InitUser()` wird beim Starten des HU-Prolog-Systems genau einmal aufgerufen. Hier könne diverse Initialisierungen vorgenommen werden. Speziell können hier dem System auch neue Atome bekanntgemacht werden.

Realisierung der Build-In Prädikate

```
GLOBAL void InitUser()
{
    InitUAtom(LENGTH_2, "my_length", NONO, 2, EVALP, true);
    InitUAtom(LAST_2, "my_last", NONO, 2, EVALP, true);
    /* ... */
}
```

Dabei hat ein Aufruf von `InitUAtom()` folgende Bedeutung:

Das erste Argument definiert das neue Atom. Dieses Atom erhält die in den folgenden Argumenten beschriebenen Eigenschaften und wird in die Atometabelle eigekehrt.

Das zweite Argument beschreibt die Zeichenkette, unter der das neue Atom bekannt werden soll.

Wenn das dritte Argument `NONO` ist, steht das vierte für die Stellenzahl des Atoms. Wenn das dritte Argument einen anderen Wert des Aufzählungstyps `OCLASS` beinhaltet, wird das Atom als Operator deklariert. Die Stellenzahl, der Vorrang und die Assoziativität des Operators lassen sich dann aus dem zweiten und dritten Argument herleiten. Das fünfte und sechste Argument von `InitUAtom()` beschreibt den Inhalt der Attribute `class()` und `system()` des neuen Atoms. (siehe auch Kapitel 2.2)

Analog zur Funktion `InitUser()` gibt es eine Funktion `ExitUser()`, die vor dem Verlassen des Prolog-Systems aufgerufen wird.

```
GLOBAL void ExitUser()
{
    /* empty */
}
```

Als nächstes folgen Funktionen, die der Nutzer zur Realisierung der Build-In Prädikate zur Verfügung stellen muß. Die hier angegebenen Funktionen wurden bereits im Kapitel über Listenverarbeitung dokumentiert.

Realisierung der Build-In Prädikate

```
LOCAL boolean DOLEN()
{
    cardinal list_len;

    if(!islist(A0, false))
        Error(ARGE);

    for(list_len=0;
        name(A0) == CONS_2 ;
        A0 = arg2(A0), ++list_len);

    return IntResult(A1, list_len);
}

LOCAL boolean DOLAST()
{
    if(!is_list(A0, false))
        Error(ARGE);

    while(name(T = arg2(A0)) == CONS_2)
        A0 = T;

    return UNI(A0, A1);
}
```

Abschließend der Sprungverteiler für die neuen Prädikate.
Dieser funktioniert analog zu CallEvalPred() und wird von
dort aufgerufen.

```
GLOBAL boolean CallUserPred(A)
    ATOM A;
{
    switch(A)
    {
        case LENGTH_2:   return DOLEN();
        case LAST_2:    return DOLAST();
        default: SystemError("CallUserPred");
    }
}
```

Eine weitere mögliche Version dieses Files findet der
interessierte Leser in den beiliegenden Quellen von HU-
Prolog.

5. Globale Steuerung, Toplevel

Nach der Beschreibung des eigentlichen Interpreterkernes nun zu den ihn umgebenden Funktionen. Diese sollen hier nur auszugsweise und in ihrer grundlegenden Struktur behandelt werden.

Beim Aufruf eines C-Programmes wird die Funktion `main()` aufgerufen. Diese hat in HU-Prolog folgende Form:

```
main(argc,argv)
    int argc;
    char *argv[];
{
    boolean first_time = true;

    /* Behandlung der Kommandozeilenargumente */
    Init.Arguments(argc,argv);
```

Anschließend werden alle Komponenten des HU-Prolog Systems initialisiert. Dafür wird für jede Komponente eine spezielle Initialisierungsfunktion aufgerufe. Durch diese werden zum Beispiel die Atomtabelle mit den vordefinierten Atomen gefüllt, einige vordefinierte Prologprädikate in die Datenbasis geschrieben, usw.

```
Init_Atoms();
Init_Database();
/* usw. */
```

Als nächstes werden die Wiedereintrittspunkte für den Fehlerfall initialisiert. (vergleiche Kapitel 4.1) Die Standardbehandlung für die Funktion `ERROR()` wird durch die Funktion `ABORT()` realisiert.

```
setjmp(Abort_Level);
ERRORFLAG = NOERROR;
setjmp(Error_Level);
if(ERRORFLAG != NOERROR)
    ABORT( ERRORFLAG );
```

Beim Start des Interpreters können sowohl ein 'Systemfile' als auch ein Programm eingeladen werden. Dazu wird dieselbe Funktion wie beim Standardprädikat `consult/1` benutzt.

Globale Steuerung und Toplevel

```
if( first_time )
{
    first_time = false;

    if( *SYS_FILE != '\0' )
    {
        TERM T;
        aSYSMODE = true;
        T=mkatom(LOOKUP(SYS_FILE,0,false,true));
        DOCONSULT(T,false);
        aSYSMODE = false;
    }

    if( *PROG_FILE != '\0' )
    {
        TERM T;
        T=mkatom(LOOKUP(PROG_FILE,0,false,true));
        DOCONSULT(T,false);
    }
}
```

Nach diesen Initialisierungen wird der interaktive Toplevel gestartet, der die gesamte folgende Arbeit übernimmt.

```
TOLEVEL();
exit(0);
}
```

Die Funktion **TOLEVEL()** realisiert eine unendliche Schleife die nur durch spezielle Prädikate abbrechbar ist. Diese Prädikate setzen die globale Variable **HALTFLAG** auf **true**.

```
void TOLEVEL()
{
    while( ! HALTFLAG )
    {

        /* Stacks initialisieren */
        init_string_stack();
        init_atom_stack();
        init_term_stack();
        init_trail_stack();
        init_env_stack();
```

Wenn Klauseln zum Prädikat **toplevel/0** vorhanden sind, so werden diese aufgerufen. Damit kann sich der HU-Prolog Nutzer seinen eigenen Toplevel definieren.

```
if(clause(TOP_0))
    EXECUTE(mkatom(TOP_0),NIL_ENV);
```

Globale Steuerung und Toplevel

Wenn es keinen nutzerdefinierten Toplevel gibt, wird der Standard-Toplevel realisiert. Dazu wird nach Ausgabe eines Promptzeichens ein Term eingelesen, und anschließend folgendermaßen modifiziert:

```
aus: a(X)
      wird:
a(X),write('X = '),write(X),not ask(59).
```

Dieser Term wird dann abgearbeitet.

```
else
{
    TERM T;
    ws("?- ");
    T = READIN();
    T = mktoplevel(T);

    if(EXECUTE(T,NIL_ENV))
        ws("yes\n");
    else
        ws("no\n");
}
}
```

6. Schlußbemerkungen

In den vorangegangenen Kapiteln ist die prinzipielle Arbeitsweise von HU-Prolog erläutert worden. Dabei konnte, schon allein aus Platzgründen, keine vollständige Implementations-Dokumentation gegeben werden. Nicht beschriebenen Teile des HU-Prologsystems sollten sich aus dem bisher bekanntem und den beiliegenden Quelltexten leicht erschließen lassen.

Obwohl HU-Prolog ein recht effizientes System ist, kann man die Effizienz natürlich noch verbessern. Dabei gibt es prinzipiell zwei Wege.

Zum einen kann man die Semantik des Interpreterkerns ändern, also zum Beispiel eine vollständige LCO (last call optimization) einbauen. Das setzt allerdings größere Änderungen im Gesamtkonzept des Interpreters voraus. Für eine LCO ist zum Beispiel die Teilung des Term-Stacks notwendig. Dabei besteht immer die Gefahr, daß man durch Optimierung einer Komponente die Effizienz einer anderen Komponente verringert.

Der zweite Weg besteht darin, bei gleichbleibenden Grundalgorithmen die Implementation (den C-Quelltext) zu optimieren. Hauptaugenmerk ist dabei auf die Unifizierung zu richten, da sie die meiste Zeit beansprucht. Die kommerziell vertriebene Version von HU-Prolog geht diesen Weg und erreicht damit eine Verdoppelung der Geschwindigkeit und eine Reduzierung des Speicherplatzbedarfes. Das wird allerdings auf Kosten unübersichtlicherer Quelltexte und der Durchbrechung des Prinzips der abstrakten Datentypen durchgesetzt.

Es scheint, daß die in dieser Arbeit beschriebene Implementation einen guten Kompromiß zwischen Effizienz des Programmes und Übersichtlichkeit der Implementation darstellt.

Portabilität und Wartungsfreundlichkeit von HU-Prolog beruhen auf der ausschließlichen Nutzung einer höheren Programmiersprache sowie der konsequenten Nutzung abstrakter Datentypen. Hinter dem Konzept der abstrakten Datentypen muß aber auch eine effektive Implementation der Schnittstelle zu diesen Datentypen stehen.

Unter den zu Beginn der Arbeit an HU-Prolog verbreiteteren Programmiersprachen bot C zusammen mit dem zur Sprache gehörenden Präprozessor die besten Möglichkeiten, dies zu realisieren.

Der Nachteil von C, insbesondere bei intensiver Nutzung des Präprozessors, ist die mangelnde Kontrollmöglichkeit des Compilers über die Einhaltung der Schnittstellen. In HU-Prolog wird dieser Nachteil durch eine über den Präprozessor

Schlußbemerkungen

schaltbare zweite "saubere" Implementation der abstrakten Datentypen umgangen. Hierbei wird kein Wert mehr auf Effizienz gelegt, dem (ANSI) C-Compiler aber maximale Möglichkeiten zur Schnittstellenprüfung eingeräumt.

Heute zeichnet sich mit C++ eine alternative Implementationssprache ab. Der Vorteil wäre eine mögliche Verlagerung der Schnittstellen zu den Datentypen aus dem Präprozessor in die eigentliche Programmiersprache, welches dem Compiler erheblich mehr Möglichkeiten zur Typprüfung einräumen würde, ohne der Effizienz zu schaden (vgl. [8] Seite 380). Des Weiteren würde ein objektorientierter Ansatz die Struktur der Quelltexte noch einmal vereinfachen. Hier fehlt es aber noch an weitergehenden Erfahrungen des Autors im Umgang mit C++.

7. Literaturverzeichnis

- [1] Ch.Horn, M.Dziadzka, M.Horn: Sprachbeschreibung HU-Prolog, edv-aspekte, Berlin 8 (1989) 4, 2-31
- [2] M.Dziadzka: Datenrepräsentation in HU-Prolog, iir, Inform., Inf. Rep., Berlin 6 (1990) 5, 52-66
- [3] M.Horn: Die funktionale und prozedurale Komponente von HU-Prolog, iir, Inform., Inf. Rep., Berlin 6 (1990) 5, 33-51
- [4] H.J.Goltz, H.Herre: Mathematische Grundlagen der logischen Programmierung, iir, Inform., Inf. Rep., Berlin 4 (1988) 14
- [5] H.J.Goltz: Unifikationstheorie, in: J.Grabowski u.a.: Grundlagen der Künstlichen Intelligenz, Akademie Verlag Berlin 1989
- [6] W.F.Clocksin, C.S.Mellish: Programming in Prolog, Springer Verlag Berlin Heidelberg 1981, 1984
- [7] Prolog ISO 6373, Draft for Working Draft 2.0, International Organization for Standardization, July 1989
- [8] M.A.Ellis, B.Stroustrup: The Annotated C++ Reference Manual, ANSI Base Document, Addison-Wesley Publishing Company, 1990