Diplomarbeit

Implementierung eines Prolog-Interpreters für die Vienna Abstract Machine mit einem und zwei Befehlszeigern

ausgeführt am Institut für
Computersprachen
an der Technischen Universität Wien

unter Anleitung von

Univ.Doz. Dipl.-Ing. Dr. Andreas Krall

und

Univ.Ass. Dipl.-Ing. Dr. Anton Ertl

als verantwortlich mitwirkenden Universitätsassistenten

von

Fernando Dorner
Paulusgasse 6/13
A-1030 Wien

Mat.Nr.: 9025728

Wien, im Jänner 1997

Kurzfassung:

Die vorliegende Diplomarbeit beschreibt die Implementierung eines kombinierten Prologcompilers und Prologinterpreters, der $VAM_{2\&1P,}$ auf der Basis der Vienna Abstract Machine (VAM).

Verschiedene Varianten dieser abstrakten Maschinen wurden für unterschiedliche Zwecke entworfen. Die VAM_{2P} eignet sich gut für einen schnellen Interpreter für kompakten Zwischencode, die VAM_{1P} unterstützt eine noch schnellere Ausführung der Prologprogramme auf Maschinencodebasis. Die $VAM_{2\&1P}$ nützt die Vorteile der VAM_{2P} und der VAM_{1P}, indem sie einen ähnlich kompakten Zwischencode wie des Interpreters VAM_{2P} verwendet und für selbstrekursive *Klauseln* einen optimierten Zwischencode ausführt.

Der Hauptteil dieser Diplomarbeit beschreibt die Implementierungen der VAM_{2P}, VAM_{1P} und $VAM_{2\&1P}$. Anhand einiger Prologbeispiele wird die Übersetzung in Zwischencode, das Speichermodell und die Arbeitsweise des (Zwischencode-)Interpreters dargestellt.

Abstract:

This thesis describes the implementation of a mixed compiler/interpreter system ($VAM_{2\&1P}$) for the programming language Prolog. The implementation is based on the Vienna Abstract Machine (VAM), which has been developed at the TU Wien.

Different versions of the VAM are used for different purposes. The VAM_{2P} is well suited for fast interpretation, the VAM_{1P} has been designed for native code compilation. Some advantages of the VAM_{2P} (compact intermediate code size) and the VAM_{1P} (fast execution of recursive calls of the last subgoal) have been combined in the $VAM_{2\&1P}$.

This thesis presents different models of the VAM (VAM_{1P}, VAM_{2P}, $VAM_{2\&1P}$), describes the compilation to the $VAM_{2\&1P}$-code, the purpose of the memory model, and gives the details of the interpreter.

Inhaltsverzeichnis

1 Einleitung

1.1 Historische Entwicklungen

Die Programmiersprache Prolog wurde zu Beginn der siebziger Jahre von Alain Colmerauer und Philippe Roussel entwickelt [Colm93]. Der erste Prolog-Interpreter wurde zunächst in Algol, spätere Versionen in Fortran und Pascal implementiert. Diese ersten Versionen waren aber noch nicht sehr effizient.
Mit diesen Erfahrungen stellten G. Battani, H. Meloni und R. Bazzoli ein effizienteres Prolog-System zusammen. Es handelte sich dabei um einen "structure-sharing"-Interpreter. Dieser hatte im Wesentlichen bereits dieselben builtin-*Prädikate* der heutigen modernen Prolog-Systeme.
Warren und Pereira stellten 1977 ein Prologsystem fertig, das aus Interpreter und Compiler (DEC-10 Prolog) bestand [Warr77]. 1983 wurde von Warren eine weitere abstrakte Maschine entwickelt, die heute als Warren Abstract Machine (WAM) bekannt ist [Warr83]. Eine abstrakte Maschine arbeitet ein (Prolog-)Programm wie ein Prozessor ab, allerdings mit dem Unterschied, daß diese Abarbeitung implizit durch ein Maschinencodeprogramm und nicht durch den Prozessor selbst ausgeführt wird. Dieses für die Programmabarbeitung notwendige Maschinencodeprogramm wird natürlich auf einem Prozessor direkt ausgeführt. Der Instruktionssatz, den das Maschinencodeprogramm abarbeiten kann mit dem dazugehörigen Speichermodell wird als abstrakte Maschine bezeichnet. Die WAM stellte auch die Grundlage der meisten darauffolgenden Prolog-Implementierungen dar.

Die WAM teilt die *Unifikation* der *Argumente* (siehe Kapitel 1.2.1) einer *Klausel* in Schreib- und Lesezugriffe auf Register auf. Die Vienna Abstract Machine (VAM) eliminiert diese Registerzugriffe indem die *Unifikation* der *Argumente* in einem Schritt durchgeführt wird. Die erste Variante der Vienna Abstract Machine, die VAM_{2P}, basiert auf den beschriebenen Grundlagen in den Arbeiten von Andreas Krall [Kral86], [Kral87], [Kral88], [Kral90]). Sie übersetzt ein Prologquellprogramm in Zwischencodebefehle, die dann interpretiert und der Reihe nach abgearbeitet werden.
Die VAM_{1P} ist eine abstrakte Maschine, die ein Prologprogramm in ein fertiges Maschinencodeprogramm übersetzt, das dann von einem realen Prozessor (Maschine) ausgeführt werden kann [Berg91].
Auf der Basis der VAM_{2P} wurde die VAM_{AI}, eine abstrakte Maschine für abstrakte Interpretation, entwickelt [Chen94], [KrBe95_1]. Bei der abstrakten Interpretation werden über das Prologquellprogramm globale Analysen erstellt. Die daraus zusätzlich erhaltenen Informationen gegenüber der VAM_{2P} werden direkt in den Zwischencode der VAM_{AI} abgebildet und für Optimierungen bei der Codeerzeugung der VAM_{1P} [KrBe95_2] verwendet. Weitere Informationen über abstrakte Interpretation können in [Chen94] nachgelesen werden.

1.2 Abarbeitung eines Prolog-Programmes

Jedes Prologprogramm wird durch elementare Grundbausteine aufgebaut, den *Termen*. Ein *Term* ist entweder eine *Variable*, eine Konstante, oder ein zusammengesetzter *Term*, dessen *Argumente* wieder *Terme* sind. Konstante sind entweder ganze Zahlen oder Atome (das sind beliebige Zeichenketten).

Konstanten:	42	otto	'demo'
Variablen:	X	Liste	

Zusammengesetzte *Terme* werden auch als Strukturen bezeichnet und bestehen aus einem *Funktor* (dieser ist ein Atom) und n *Argumenten*, die wieder beliebige *Terme* sind. Ein Prologprogramm besteht aus *Klauseln* und *Fakten*, die aus Strukturen zusammengesetzt werden. Ein *Faktum* besteht nur aus einer Struktur, die durch einen Punkt abgeschlossen wird.

Beispiel: Prologprogramm

Klausel: `a(A,B) :- b(A), c(B).`
Fakten: `b(1).` `c(1).`

Dieses Prologprogramm besteht aus der *Klausel* `a` und den beiden *Fakten* `b` und `c`. Die beiden *Fakten* `b` und `c` treten zusätzlich in der *Klausel* `a` als *Unterziele* `b` und `c` auf, wobei man den Term `a(A,B)` als *Kopf*, und die Terme `b(A)` und `c(A)` als *Rumpf* der zu beweisenden *Prologklausel* bezeichnet. `A` und `B` stellen sogenannte *Variablen* dar, die während der *Unifikation* an die Konstante `1` gebunden werden. Die bei den Termen `a,b` und `c` in Klammern angegebenen Parameter (`A,B,1`) werden auch als *Argumente* des Terms bezeichnet. Als *Prädikat* oder Prozedur bezeichnet man eine Gruppe von *Klauseln* (auch *Fakten*), die dieselbe *Kopf*-Struktur (Name und Argumentanzahl) aufweisen.

Die Abarbeitung von Prolog-Programmen kann in zwei grundlegende Schritte eingeteilt werden: *Resolution* und *Unifikation*:

Bei der *Resolution* in Prolog wird versucht, zu jedem *Unterziel* (U_i) einer zu beweisenden *Klausel* eine *Klausel* zu finden, deren *Kopf* denselben *Funktor* (Name und Argumentanzahl der *Klausel*) besitzt wie U_i und alle *Argumente* unifizierbar sind (siehe *Unifikation*). Gibt es für die neue *Klausel* ebenfalls *Unterziele*, so werden auch diese auf dieselbe Art bewiesen - erst dann ist U_i bewiesen. Prologprogramme werden nach der depth-first-Strategie abgearbeitet (U_i wird gänzlich aufgelöst, bevor eine Lösung für U_{i+1} gesucht wird). Diese Form der *Resolution* wird auch als *SLD-Resolution* bezeichnet [Lloy87].

Unifikation bezeichnet den Vorgang, bei dem zwei *Terme* durch geeignete *Instanzierung* (Substitution der *Variablen* durch *Terme*) zur Übereinstimmung gebracht werden. *Unifikationen* kommen beim Aufruf einer *Klausel* zwischen den *Argumenten* des aufrufenden *Ziels* und den *Argumenten* des *Kopfes* der aufgerufenen *Klausel* vor. Es wird dabei ein "most general unifier" zweier *Terme* gesucht. Das ist die kleinste Menge von Substitutionen, die angewendet werden müssen, um die beiden *Terme* in Übereinstimmung zu bringen.

1.2.1 *Unifikationsalgorithmus*

Zwei *Terme* sind unifizierbar, falls

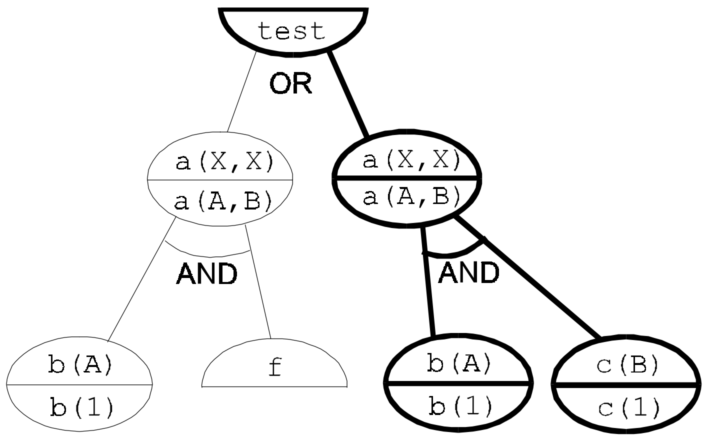
- beide *Terme* Konstanten sind und diese gleich sind
- beide *Terme* Strukturen sind und diese den gleichen *Funktor* (Name und Arität) und die gleiche Argumentzahl haben und alle *Argumente* unifizierbar sind
- beide *Terme Variablen* sind (diese werden bei der *Unifikation* aneinander gebunden)
- ein *Term* eine freie *Variable* und der andere eine Konstante oder Struktur ist (der zweite *Term* wird der freien *Variablen* zugewiesen)

1.2.2 Aufbau der Ablaufsteuerung

Der Ablauf eines Prologprogrammes entspricht dem Beweisen einer prädikatenlogischen Aussage, wobei das Programm von oben nach unten nach einer passenden *Klausel* durchsucht wird und die *Unterziele* einer *Klausel* von links nach rechts bewiesen werden. Die Abarbeitung von Programmen kann durch sogenannte Beweisbäume dargestellt werden [Bruy82], deren Knoten den einzelnen *Unterzielen* des eine Stufe höher stehenden *Ziels* entsprechen.

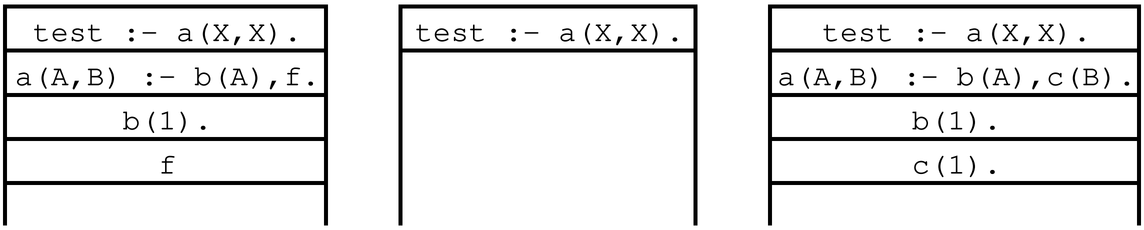
Als Beispiel ist für folgendes Prologprogramm der Beweisbaum gegeben:

```
test :- a(X,X).
a(A,B) :- b(A), f.
a(A,B) :- b(A), c(B).
b(1).
c(1).
```



Wenn ein *Ziel* nicht bewiesen werden kann, wird versucht, ab der letzten existierenden Alternative einen neuen Beweisbaum aufzubauen. Dieser Schritt wird auch als *Backtracking* bezeichnet. Die dünnen Linien beim Beweisbaum stellen den Beweisbaum bis zum Scheitern von f dar. Danach wird *Backtracking* eingeleitet. Die dicken Linien kennzeichnen die Lösung des Prologprogramms. Die "AND"-Pfade im Beweisbaum enthalten die einzelnen *Unterziele* einer *Klausel*. Die "OR"-Pfade stellen die verschiedenen Lösungswege eines Prologprogrammes dar.

Um den Beweisbaum in linearisierter Form speichern zu können und das *Backtracking* einfach darstellen zu können, eignet sich am besten ein Stack. Zu beachten ist, daß sich immer nur ein Lösungsweg ("OR"-Pfad) des Beweisbaumes am Stack befindet. Beim *Backtracking* wird der "überflüssige" Teil des Beweisbaumes vom Stack entfernt.



liefert FALSE   *Backtracking*   liefert TRUE

Auf diesem Stack wird für jede zu beweisende *Klausel* ein sogenanntes *stack frame* angelegt. Beim *Backtracking* werden alle *stack frames* bis einschließlich der Alternativen gelöscht. Der Beweisvorgang samt Anlegen neuer *stack frames* wird bei der nächsten alternativen *Klausel* fortgesetzt. Dieser Vorgang wird fortgesetzt bis das Prologprogramm bewiesen ist, oder endgültig scheitert. In welcher Form ein Prologprogramm dargestellt wird, und wie es mit Hilfe einiger Stacks abgearbeitet wird, zeigen die folgenden Kapitel.

Die Implementierung eines Prolog-Interpreters oder -Compilers ist die Implementierung eines automatischen Beweisers für prädikatenlogische Aussagen. Der Interpreter besteht aus Programmteilen für die *Unifikation*, die *Resolution* und die vordefinierten *Prädikate*.

2 Datenstrukturen und Speichermodell

2.1 Datenstrukturen

Weil in Prolog die Datentypen zur Laufzeit geprüft werden, kann der Typ und der Wert einer *Variablen* erst zur Laufzeit bestimmt werden. Deshalb muß auch die Typinformation in den *Variablen* gespeichert werden. Um eine effiziente Verarbeitung zu ermöglichen, besitzen die Speicherzellen immer dieselbe Größe (hier ein Maschinenwort MW → 32 Bits), einen Bereich für den Wert und einen für den *tag* (Schild, Typ). Die Anzahl der Bits die für den *tag* verwendet werden ist variabel. Die *Terme* des Prologprogrammes werden für die *Unifikation* in unten angeführter Form gespeichert.

Objekt	Wert (Bits 31-3)	*tag* (Bits 2-0)
Atom (Zeichenkette)	Nummer des Atoms	100 "Atom"
Integer	ganze Zahl (Zweierkomplementdarstellung)	000 "Integer"

Tabelle 1

Beispiele:
Das *Faktum* `männlich(otto)` beinhaltet das Atom `otto`, das durch die Nummer des Atoms (z.B.: 1) und dem *tag* "100" klassifiziert wird.
Das *Faktum* `schaltjahr(1996)` beinhaltet die ganze Zahl (Integer) `1996`. Die ganze Zahl `1996` wird in Zweierkomplementdarstellung abgespeichert. Der Typ wird anhand des *tags* bestimmt.

Solange eine *Variable* noch mit keinem Wert initialisiert worden ist, wird sie als "ungebunden" bezeichnet. Ungebundene *Variablen* werden durch eine sogenannte Selbstreferenz ("Zeiger auf sich selbst") dargestellt. Die Selbstreferenz wird ebenfalls durch eine Speicherzelle mit *tag* dargestellt, wobei der Wert die Adresse der Speicherzelle ("Zeiger auf sich selbst") ist (siehe Tabelle 2):

Objekt	Wert (Bits 31-2)	*tag* (Bits 1-0)
Referenz	Zeiger auf die referenzierte *Variable*	01 "Referenz"

Tabelle 2

Erhält diese *Variable* während der *Unifikation* einen Wert, so wird sie als "gebunden" bezeichnet. Die Speicherzelle für die ungebundene *Variable* wird dann durch eine Speicherzelle mit *tag*, die den Wert und den Typ des Objekts (Atom, Integer, Liste, Struktur) kennzeichnet, ersetzt.

In Prologprogrammen können noch weitere Datenstrukturen auftreten: Strukturen

Eine Struktur wird ebenfalls mit Hilfe einer Speicherzelle mit einem speziellen *tag* dargestellt, die auf einen Speicherbereich zeigt mit n+1 Speicherzellen (n Elemente der Struktur und ein *Funktor*). Ein *Funktor* besteht aus dem Namen und der Arität (Anzahl der Elemente der Struktur) einer Struktur (siehe Abb. 1).

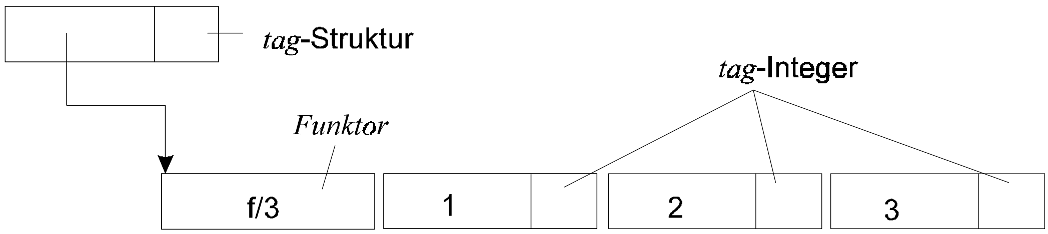


Abb. 1: Darstellung der Struktur f(1, 2, 3)

Eine Liste stellt den Spezialfall einer zweielmentigen Struktur dar. Für diesen Datentyp wird weder Name noch die Anzahl der *Argumente* (2) der Struktur benötigt. Eine Liste wird mit Hilfe einer Speicherzelle mit einem speziellen *tag* für Listen dargestellt, die auf einen Speicherbereich mit zwei weiteren Speicherzellen mit *tags* zeigt. Die erste Speicherzelle beinhaltet dann das erste Element der Liste, die zweite den Rest der Liste (siehe Abb. 2). Beide Elemente können wieder Listen sein. Eine leere Liste wird durch ein spezielles Atom dargestellt, wobei die Nummer des Atoms eindeutig ist und von keinem anderen Atom verwendet wird.

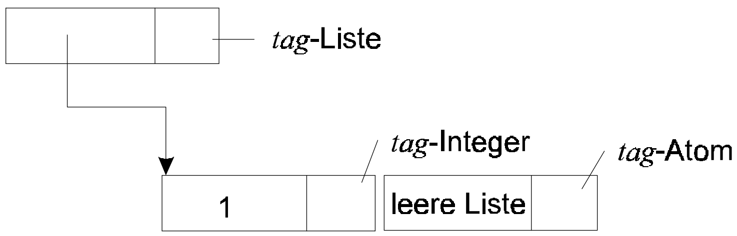


Abb. 2: Darstellung der Liste [1]

In Tabelle 3 und 4 werden die Speicherzellen für Strukturen, Listen und leere Listen dargestellt:

Objekt	Wert (Bits 31-2)	*tag* (Bits 1-0)
Liste	Zeiger auf das erste Element der Liste	10
Struktur	Zeiger auf den Beginn des Strukturblocks (*Funktor*)	11

Tabelle 3

Objekt	Wert (Bits 31-3)	*tag* (Bits 2-0)
nil (leere Liste)	1..1	100

Tabelle 4

2.2 Speichermodell

2.2.1 Einführung

Während der Ausführung eines Prologprogrammes werden bei der VAM drei Stacks benötigt (siehe Abb. 3), um die Aufrufhierarchie der Prologterme zu verwalten und die *Variablen* zu speichern. Die aus dem Prologquellprogramm (*Terme*) generierten Zwischencodebefehle werden im vierten Speicherbereich, dem sogenannten *heap*, abgespeichert. Die Erzeugung und die Abarbeitung dieser Zwischencodebefehle wird in den Kapiteln 3, 4 und 5 anhand der abstrakten Maschinen VAM_{2P}, VAM_{1P} und $VAM_{2\&1P}$ beschrieben. Im *heap* befinden sich auch alle Atome eines Prologprogrammes, die mittels einer Hashtabelle verwaltet werden.

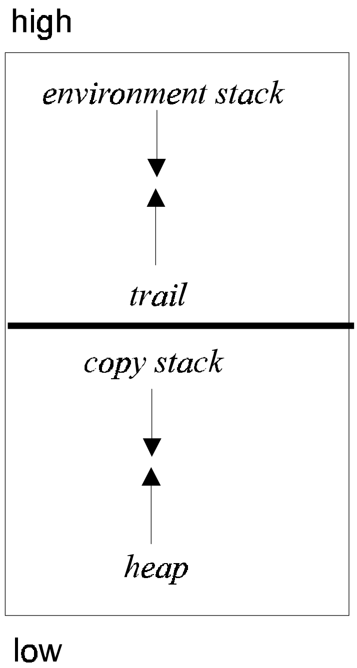


Abb. 3 : Speichermodell bei der VAM

Der *environment stack* speichert die *Variablen* und Rücksprungadressen der aufeinanderfolgenden *Unterziele*. *Variablen* von der Größe eines Maschinenwortes können direkt in diesem Stack abgelegt sein, alle anderen sind jedoch als Verweis in den *copy stack* abgelegt weil die Größe der zusammengesetzten Terme erst zur Laufzeit bestimmt werden kann. Listen und Strukturen werden deshalb am *copy stack* gespeichert. Um Variablenreferenzen innerhalb des *environment stacks* zu vermeiden, wird eine ungebundene *Variable* am *copy stack* angelegt und von der *Variablen* im *environment stack* referenziert. Auf dem *trail* werden die Adressen der *Variablen* markiert, die während des Programmablaufes gebunden werden und beim *Backtracking* wieder zurückgesetzt werden müssen. Der *trail* enthält nur Referenzen auf den *copy stack*. Im *trail* werden die Referenzen direkt durch die Adresse des Maschinenwortes ohne *tag* dargestellt.

2.2.2 Programmabarbeitung

Für jeden Aufruf eines *Unterziels* und für den Aufruf der ersten *Klausel* wird ein *stack frame* am *environment stack* angelegt, das die *Variablen* einer *Klausel* und die für den Beweisvorgang nötigen Kontrollinformationen enthält. Bei *stack frames* wird zwischen indeterministischen und deterministischen *stack frames* unterschieden. Ein *stack frame* wird als deterministisch bezeichnet, falls für eine *Klausel* keine alternativen *Klauseln* existieren, ansonsten als indeterministisch (*choice point*).

Bei einem *stack frame* werden die *Variablen*, der Zeiger auf das *stack frame* der aufrufenden *Klausel* (Vorgängerframe) und der Zeiger auf den Fortsetzungscode (Rücksprungadresse) nach dem aufrufenden *Unterziel* gespeichert. Der Zeiger auf das Vorgängerframe (`goalframptr`) und der Zeiger auf den Fortsetzungszwischencode (`goalptr`) werden auch als *continuation* bezeichnet (siehe Abb. 4).

`goalframeptr`	Zeiger auf das Vorgängerframe
`goalptr`	Zeiger auf Adresse des Fortsetzungscodes
Variablen des *Unterziels*	*Variablen*

Abb. 4: *stack frame (*oder *activation record)*

Wenn mehrere Alternativen zu einer *Klausel* existieren, wird zusätzlich ein *choice point* am *environment stack* angelegt. Bei einem *choice point* werden Zeiger auf das *stack frame* der aufrufenden *Klausel* (Vorgängerframe), das aufrufende *Unterziel* (`goalptr`), die noch existierenden alternativen *Klauseln* (`alternatives`), den letzten *choice point* (`choicepointptr`), die Spitze des *copy stacks* (`copyptr`) und die Spitze des *trails* (`trailptr`) gespeichert (siehe Abb. 5).

`goalframeptr`	Zeiger auf das Vorgängerframe
`goalptr`	Zeiger auf das aufrufende *Unterziel*
`alternatives`	Zeiger auf alternative *Klauseln*
`trailptr`	Zeiger auf die Spitze des *trails*
`copyptr`	Zeiger auf Spitze des *copy stacks*
`choicepointptr`	Zeiger auf letzten *choice point*

Abb. 5: *choice point*

Hat das Programm die Bearbeitung einer *Klausel* erfolgreich abgeschlossen (es existieren keine weiteren *Unterziele*), so wird zunächst der `goalframeptr` wiederhergestellt und dann auf die Fortsetzungsadresse (`goalptr`) gesprungen.
Scheitert ein Programmzweig, dann wird der letzte *choice point* gesucht. Die auf dem *trail* markierten *Variablen* werden am *copy stack* in den "ungebundenen" Zustand zurückgesetzt. Die *stack frames* bis zur letzten Alternativen werden vom Stack entfernt. Der Programmablauf wird bei der ersten alternativen *Klausel* fortgesetzt (*Backtracking*).

Für die Abarbeitung der Zwischencodebefehle verwendet die VAM folgende Register:

`stackptr`	Zeiger auf die Spitze des *environment stack*
`copyptr`	Zeiger auf die Spitze des *copy stack*
`trailptr`	Zeiger auf die Spitze des *trail*
`choicepointptr`	Zeiger auf den letzten *choice point*
`goalframeptr`	Zeiger auf das *stack frame* der aufrufenden Prozedur
`goalptr`	Zeiger auf den Code der *Argumente* des Aufrufers
`headframeptr`	Zeiger auf das *stack frame* der aufgerufenen Prozedur
`headptr`	Zeiger auf den Code der *Argumente* des *Kopfes* der aufgerufenen *Klausel*

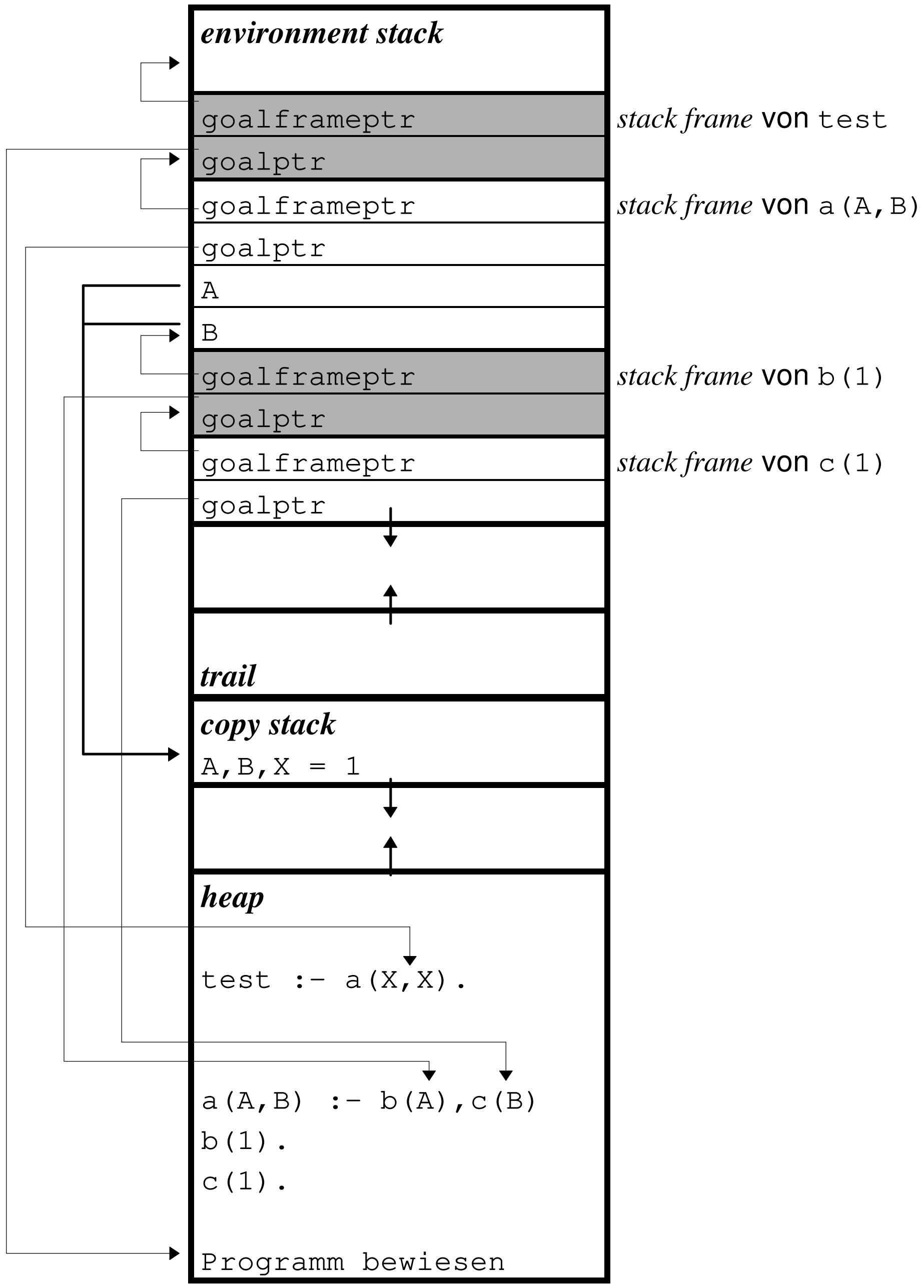
Der `headptr` und `goalptr` stellen die zwei Instruktionszeiger des Interpreters auf den Zwischencode im *heap* dar. Dabei zeigt der `headptr` auf den nächsten Zwischencodebefehl der aufgerufenen *Klausel*, `goalptr` auf den der aufrufenden Prozedur. Der `goalframeptr` beziehungsweise `headframeptr` zeigt jeweils auf das *stack frame* (aufgerufene oder aufrufende Prozedur) in dem sich die *Variablen* für die beiden Prozeduren und die nötigen Kontrollinformationen für den Beweisvorgang befinden. Existiert ein *choice point*, dann wird über den aktuellen `choicepointptr` auf die benötigten Informationen zugegriffen. Der `trailptr` zeigt auf die Spitze des Speicherbereichs indem sich die Variablenreferenzen auf gebundene *Variablen* befinden. `stackptr` und `copyptr` zeigen auf die anderen Spitzen der Stacks, den *environment stack* und den *copy stack*.

Die Aufgabe der verschiedenen Stacks wir anhand folgenden Beispiels veranschaulicht:

```
test :- a(X,X).
a(A,B) :- b(A), c(B).
b(1).
c(1).
```

Der in der *continuation* enthaltene Zeiger auf den Fortsetzungszwischencode (`goalptr`) zeigt auf den Zwischencode im *heap*, auf den gesprungen wird, wenn alle *Unterziele* der *Klausel* abgearbeitet worden sind. Die einzelnen *stack frames* werden für die Abarbeitung der *Resolution* mit einem Zeiger auf das Vorgängerframe (`goalframeptr`) "dynamisch verkettet".
Für `test` wird ein *stack frame* angelegt ohne *Variablen*. Die *Variable* `X` wird in einem eigenen Speicherplatzbereich gespeichert, weil es sich dabei um eine temporäre *Variable* handelt (siehe Kapitel 3). Beim Aufruf von `a(X,X)` wird für die *Klausel* `a(A,B)` ein *stack frame* angelegt mit dem Speicherplatz für zwei *Variablen* `A` und `B`. Weil es sich bei `X` und `A` um *Variablen* handelt, die jeweils das erste Mal in einer *Klausel* auftreten wird durch den *Unifikationsalgorithmus* des Prolog-Interpreters oder -Compilers eine ungebundene Speicherstelle (Selbstreferenz) am *copy stack* angelegt. `B` verweist dann, so wie `A` und `X` (Referenzen), auf die Speicherstelle im *copy stack*. Beim Aufruf von `b(A)` wird für `b(1)` ein *stack frame* ohne *Variablen* angelegt. Gleichzeitig erfolgt bei der *Unifikation* von `b(A)` mit `b(1)` die Bindung der *Variablen* `A` an den Wert `1` in die Speicherstelle im *copy stack*. `B` wird dadurch automatisch an `1` gebunden und wird dann beim Aufruf von `c(B)` (erzeugt ein *stack frame* für `c(1)`, der für die Programmabarbeitung benötigt wird) mit `1` von `c(1)` unifiziert, was ein TRUE-Resultat zurückliefert. Die *Variable* `X` wird durch die Referenz in den *copy stack* mit `1` unifiziert, und das Prologprogramm ist bewiesen.

Die Speicheraufteilung für das Prologprogramm auf Seite 13 sieht daher folgendermaßen aus: (Die in der *continuation* enthaltenen Zeiger sind grafisch dargestellt)



Stack-Überprüfungen

Um zu verhindern, daß der Prolog-Interpreter/Compiler wegen Speichermangels abstürzt und bereits vorher in ungeschützte Speicherbereiche schreibt, wird vor jedem *Unterzielaufruf* überprüft, ob noch genügend Speicherplatz vorhanden ist. Ist zu wenig Speicher vorhanden (*environment stack*, *copy stack*, *trail*, *heap*) so kann eine *garbage collection* Routine [Gran96] aufgerufen werden, die unnötige Variablenreferenzen aus diesen Speicherbereichen zu entfernen versucht. Wenn auch nach dieser Speicherkompaktierung noch immer zu wenig Speicher für die nächsten *Unifikations*- und *Resolutionsschritte* vorhanden ist, so bricht der Prolog-Interpreter/Compiler mit einer "zu wenig Speicher"-Fehlermeldung ab.

3 VAM$_{2P}$

Der VAM$_{2P}$-Übersetzer übersetzt das Prologquellprogramm in Zwischencodebefehle, die im *heap* gespeichert werden. Diese Übersetzung der *Terme* (Prologprogramm) in Zwischencodebefehle erfolgt in zwei Durchläufen. Der erste Durchlauf durchsucht die *Terme* nach *Variablen* und speichert die Informationen in einer Variablentabelle. Der zweite Durchlauf generiert aus den *Termen* die Zwischencodebefehle der VAM$_{2P}$ (siehe Abb. 6). Der VAM$_{2P}$-Übersetzer bestimmt zwischen diesen beiden Durchläufen die Klassen (void, temporär, lokal) und die `Offsets` der *Variablen*. Die Darstellungen der Datenstrukturen entspricht den in Kapitel 2 beschriebenen. Aus Speicherplatz- und Portabilitätsgründen werden die *Prologklauseln* nicht direkt auf Maschinencode übersetzt, sondern es wird ein linearisierter Zwischencode generiert. Für die Abarbeitung der Zwischencodebefehle verwendet der Interpreter drei Stacks und einen *heap* (siehe Kapitel 2). Der Zwischencodeinterpreter VAM$_{2P}$ führt dann die einzelnen Zwischencodebefehle aus, was dem Beweisen des Prologprogrammes entspricht.

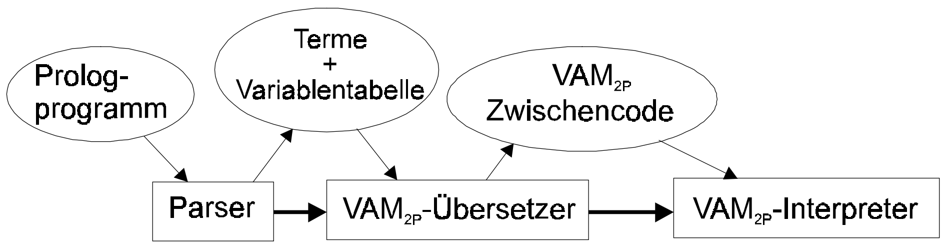


Abb. 6: Übersetzung eines Prologprogrammes in Zwischencodebefehle (VAM$_{2P}$)

3.1 Zwischencode:

Damit der Interpreter das Prologprogramm abarbeiten kann, wird das Prologprogramm in Zwischencodebefehle übersetzt. Diese steuern die *Unifikation* und die *Resolution* des Prologprogramms (siehe Kapitel "*Unifikation* & *Resolution*" auf der nächsten Seite). Für Optimierungen werden die *Variablen* eines Prologprogramms noch genauer spezifiziert:

Variablenklassifizierung:

Nach ihrer Lebensdauer unterscheidet der Prolog-Interpreter oder -Compiler drei Klassen von *Variablen*:

- void *Variablen*
- temporäre *Variablen*
- lokale *Variablen*

Tritt eine *Variable* nur einmal in einer *Klausel* auf und transportiert daher keine Information, so wird diese *Variable* als void-*Variable* bezeichnet. Temporäre *Variablen* werden nur während der *Unifikation* eines *Prädikats* benötigt, wobei sie nur als *Argumente* des *Kopfes* oder *Unterziels* auftreten. Alle anderen *Variablen* sind lokale *Variablen* (treten sowohl im *Kopf* als auch im *Rumpf* einer *Klausel* oder in zwei verschiedenen *Unterzielen* im *Rumpf* einer *Klausel* auf), sie transportieren die Informationen zwischen den *Unterzielen* und dem *Kopf* einer *Klausel*.

Beispiel:

```
for (X, X).
for (Y, 0).
for (M, N) :- N>0, N is (N-1), for (M, N).
```

`X` bezeichnet eine temporäre *Variable*. `Y` bezeichnet eine void-*Variable*. `0` wird als Integer abgespeichert. `M` und `N` bezeichnen lokale *Variablen*.

Die aus den *Variablen* gewonnenen Informationen werden in Zwischencodebefehle umgesetzt und für Optimierungen in der Abarbeitung eines Prologprogrammes verwendet (siehe Kapitel 3 bis 5).

Weil eine void-*Variable* keinen Informationsfluß in einer *Prologklausel* verursacht, braucht für diese *Variable* kein Speicherplatz reserviert werden. Solange eine temporäre oder lokale *Variablen* noch mit keinem Wert initialisiert worden ist, wird sie als "ungebunden" bezeichnet. Alle lokalen *Variablen* werden im *environment stack* in den *stack frames* gespeichert (siehe Kapitel 2). Temporäre *Variablen* werden statt im *stack frame* in einem kleinen Speicherbereich, der speziell für diese Variablenklasse reserviert ist, abgespeichert. Bei den lokalen und temporären *Variablen* wird noch zwischen ihrem ersten und allen weiteren Vorkommnissen in einer *Klausel* unterschieden. Eine *Variable*, die das erste Mal in einer *Klausel* auftritt, erhält normalerweise bei der *Unifikation* mit einer anderen *Variablen* einen Wert, außer bei der *Unifikation* mit einer *Variablen*, die ebenfalls das erste Mal in einer *Klausel* auftritt. In diesem Fall wird eine ungebundene *Variable* (Selbstreferenz) am *copy stack* angelegt. Der *environment stack* enthält dann jeweils nur einen Zeiger (Referenz) auf diese ungebundene *Variable* im *copy stack*. Bei temporären *Variablen* enthält nicht der *environment stack* sondern der für temporäre *Variablen* reservierte kleine Speicherbereich eine Referenz in den *copy stack*. Dadurch entfallen die Initialisierungen der *Variablen* beim Anlegen im *stack frame*, Einträge auf dem *trail* und Überprüfungen, ob ein Eintrag am *trail* nötig ist.

Unifikation & *Resolution*:

Ein Prologsystem führt bei der *Unifikation* entweder den Vergleich von zwei Werten, die Zuweisung eines Wertes an eine *Variable* oder das Aufbauen einer Referenz von einer *Variablen* auf eine andere durch. Bei der *Unifikation* wird jeweils ein *Term* des aufrufenden *Unterziels* und ein *Term* der aufgerufenen *Klausel* (*Kopf*) auf Gleichheit überprüft. Bei dem ersten Auftreten eines *Terms* in einer *Klausel* wird dieser mit einem Wert oder als ungebunden initialisiert und am *environment stack* im *stack frame* abgespeichert. Bei der *Unifikation* zweier lokaler *Variablen*, die beide ihr erstes Auftreten im Programm haben, wird eine ungebundene Speicherzelle am *copy stack* angelegt und von beiden *Variablen* im *environment stack* referenziert. Bei dem weiteren Auftreten desselben *Terms* wird die Speicherzelle im *copy stack* mit dem aktuellen Wert des *Terms* initialisiert, sofern die Speicherzelle zuvor ungebunden war. Hatte die Speicherzelle bereits einen Wert erhalten, dann wird sie mit dem aktuellen Wert des *Terms* im Prologprogramm verglichen. Dieser Vergleich liefert einen Wahrheitswert. Der Beweisvorgang des Prologprogrammes scheitert bei einem FALSE-Resultat. Falls eine *Unifikation* nicht erfolgreich ist, wird *Backtracking* eingeleitet. Bei einem TRUE-Resultat wiederholt sich dieser Vorgang für alle *Terme* eines *Prädikats* und für alle *Klauseln* eines Prologprogrammes (*Resolution*).

Die VAM_{2P}-Befehle bestehen aus drei Zwischencode-Gruppen, den *Unifikations*-, den Kontroll(*Resolutions*)- und den Abschluß-Befehlen.

Unifikations-Befehle

Befehl	Operand	Folgebefehle	Bedeutung
h_/g_const	Konstante C	kein	Konstante (ganze Zahl oder Atom)
h_/g_nil	keine	kein	leere Liste
h_/g_list	keine	erstes Element und Rest der Liste	Liste
h_/g_struct	keine	*Funktor* und Elemente der Struktur	Struktur
h_/g_void	keine	kein	void-*Variable*
h_/g_fsttmp	Offset	kein	erstes Auftreten einer temporären *Variablen*
h_/g_nxttmp	Offset	kein	weiteres Auftreten einer temporären *Variablen*
h_/g_fstvar	Offset	kein	erstes Auftreten einer lokalen *Variablen*
h_/g_nxtvar	Offset	kein	weiteres Auftreten einer lokalen *Variablen*

Tabelle 5

Kontroll-Befehle

Befehl	Operand	Folgebefehle	Bedeutung
goal	[N,] P (siehe Text)	*Argumente* und call/lastcall-Befehl	*Unterziel*
nogoal	keine	kein	Ende eines *Faktums*
cut	keine	kein	cut-Operator
builtin I	Index des *Prädikats*	*Argumente* und call/lastcall-Befehl	builtin-*Prädikate*

Tabelle 6

Abschluß-Befehle

Befehl	Operand	Bedeutung
call	keine	Ende eines *Unterziels* (nicht das letzte)
lastcall	keine	Ende des letzten *Unterziels*

Tabelle 7

Die *Unifikations*-Befehle (Tabelle 5) behandeln die *Unifikation* eines *Terms* vom aufrufenden *Unterziel* (z.B.: `g_const`) mit dem zugehörigen *Term* der aufgerufenen *Klausel* (z.B.: `h_fstvar`). Zu beachten ist, daß die *Unifikations*-Befehle für das aufrufende *Unterziel* ("goal"-Befehle → `g_`) anders codiert sind als die *Unifikations*-Befehle der aufgerufenen *Klausel* ("head"-Befehle → `h_`). Jeder *Term* vom aufrufenden *Unterziel* und von der aufgerufenen *Klausel* wird dem Datentyp (siehe Kapitel 2) entsprechend in die *Unifikationsbefehle* umgesetzt. Der Wert des *Terms* wird im Operanden beziehungsweise bei den Folgebefehlen nach dem *Unifikationsbefehl* abgespeichert. Die lokalen und temporären *Variablen* einer *Klausel* werden entsprechend ihrem ersten und weiteren Auftreten in die Befehle `fstvar/nxtvar` und `fsttmp/nxttmp` übersetzt. Eine lokale *Variable* wird dann an der Stelle `Offset` im *stack frame*, eine temporäre *Variable* im dafür vorgesehenen speziellen Speicherbereich an der Stelle `Offset` gespeichert. Bei *Prologklauseln* mit bis zu vier lokalen *Variablen* wird ein spezieller Zwischencode erzeugt, der den Interpreter veranlaßt, spezielle vordefinierte Speicherbereiche zu verwenden (`Offset` 0-3 im *stack frame*). Die VAM_{2P} braucht dann in diesem Fall nicht mehr den `Offset` aus dem Operanden auszulesen, sondern hat die Information implizit beim Zwischencodebefehldekodieren. Listen/Strukturen werden mit einem list/struct-Befehl eingeleitet. Die Elemente der Liste/Struktur werden anschließend mit den *Unifikationsbefehlen* laut Tabelle 5 angegeben.

Die Kontrollbefehle (Tabelle 6) steuern den Programmfluß und die *Resolution* eines Prologprogrammes und kennzeichnen den Typ des *Unterziels* (*Unterziel*, *Faktum*, cut, builtin-*Prädikat*). Der Zwischencodeinterpreter VAM_{2P} verknüpft diese Befehle mit den Abschlußbefehlen (Tabelle 7), die jeweils das Ende eines oder des letzten *Unterziels* angeben.
Ein *Unterziel* beginnt demnach mit einem `goal`-Befehl und dem Namen des *Unterziels* (*Prädikat* `P`). Danach werden die einzelnen *Terme* beziehungsweise *Argumente* des *Unterziels* in *Unifikationsbefehle* laut Tabelle 5 übersetzt. Das *Unterziel* endet schließlich mit einem `call`- oder `lastcall`-Abschlußbefehl. Beim ersten *Unterziel* einer *Klausel* wird ein zusätzlicher Operand `N` mitgeführt, der die Anzahl der lokalen *Variablen* im *Kopf* der *Klausel* angibt. Dieser Operand wird für die Lastcall-Optimierung benötigt (siehe Kapitel 3.4.2). Ein *Faktum* oder ein cut wird mit dem Kontrollbefehl `nogoal` laut Tabelle 6 abgeschlossen. Vor einem *Faktum* stehen die *Unifikationsbefehle* der *Argumente* des *Faktums*. Ein builtin-*Prädikat* beginnt mit einem `builtin`-Kontrollbefehl und dem Index `I`, der die Nummer des builtin-*Prädikats* in der builtin-Tabelle angibt. Danach werden die einzelnen *Terme* beziehungsweise *Argumente* des builtin-*Prädikats* in *Unifikationsbefehle* laut Tabelle 5 übersetzt. Das builtin-*Prädikat* endet schließlich mit einem `call`- oder `lastcall`-Abschlußbefehl.

Beispiel:

```
test :- a(X,X).
a(A,B) :- b(A), c(B).
b(1).
c(1).
```

```
test :-          "Startziel ohne Variablen"
a(               goal 0, a
X,               g_fsttmp 0
X                g_nxttmp 0
).               lastcall

a(A,             h_fstvar 0
B) :-            h_fstvar 1
b(               goal 2, b
A                g_nxtvar 0
),               call
c(               goal c
B                g_nxtvar 1
).               lastcall

b(1              h_const 1
).               nogoal

c(1              h_const 1
).               nogoal
```

Das folgende Beispiel zeigt den VAM_{2P}-Zwischencode für eine append-Prozedur:

```
test :- append([1],[],[1]).
append ([], L, L,).
append([H| L1], L2, [H| L3]) :- append(L1, L2, L3).
```

```
test :-            "Startziel ohne Variablen"
append(            goal 3, append
[                  g_list
1                  g_const 1
],                 g_nil
[],                g_nil
[                  g_list
1                  g_const 1
]                  g_nil
).                 lastcall

append([],         h_nil
L,                 h_fsttmp 0
L                  h_nxttmp 0
).                 nogoal

append([           h_list
H|                 h_fsttmp 0
L1],               h_fstvar 0
L2,                h_fstvar 1
[                  h_list
H|                 h_nxttmp 0
L3]) :-            h_fstvar 2
append(            goal 3, append
L1,                g_nxtvar 0
L2,                g_nxtvar 1
L3                 g_nxtvar 2
).                 lastcall
```

3.2 Der Zwischencodeinterpreter VAM_{2P}:

Die Hauptaufgabe des Zwischencodeinterpreters VAM_{2P} ist die Unifizierung der *Argumente* eines *Unterziels* mit den *Argumenten* im *Kopf* einer *Klausel* und die Durchführung der *Resolution*. *Unifikationsbefehle* werden mit *Unifikationsbefehlen* kombiniert, um die *Unifikation* zweier *Terme* durchzuführen. Kontrollbefehle werden mit Abschlußbefehlen kombiniert, was der Abarbeitung der *Resolution* entspricht. Um eine schnelle Entschlüsselung der Befehlskombinationen zu ermöglichen, werden die einzelnen Kombinationen durch eine Addition beider Befehle bestimmt. Dafür ist eine eindeutige Summe von jeder Befehlskombination notwendig. Die Codierung der Zwischencodebefehle der VAM_{2P} ist so gelöst, daß jede Befehlskombination (*Unifikationsbefehl*-"head" mit *Unifikationsbefehl*-"goal"; Abschlußbefehl mit Kontrollbefehl) eindeutig einer Integer-Zahl zugeordnet wird (dies trifft auch auf die VAM_{1P}- und $VAM_{2\&1P}$-Befehle zu !). Während der *Unifikation* liest der Interpreter die Befehle für die *Argumente* des *Unterziels*, dekodiert sie, liest die Befehle für die *Argumente* des *Kopfs* einer *Klausel*, dekodiert sie und unifiziert die *Argumente* beider Befehle miteinander.

Das Dekodieren der Zwischencodebefehle für den *Kopf* und den *Rumpf* erfolgt in einem Schritt, in dem der Code für das *Argument* im *Kopf* der *Klausel* und der Code für das *Unterziel* im *Rumpf* addiert werden und die Dekodierung über die Summe der beiden Werte erfolgt. In der Programmiersprache C erfolgt die Dekodierung unter Verwendung einer switch-Anweisung. Für die Dekodierung existieren zwei Zeiger auf den Zwischencode. Der Zeiger `goalptr` zeigt auf den Zwischencode des *Unterziels*, der Zeiger `headptr` zeigt auf den Zwischencode des *Kopfes* der passenden *Klausel*. Das Lesen und Dekodieren der Befehle wird in C folgendermaßen gelöst:

```
switch (*goalptr++ + *headptr++)
```

Die *Unifikation* wird dann mit Hilfe der Stacks *environment stack*, *copy stack* und *trail* durchgeführt. Dabei werden die Datenstrukturen an den Stacks angelegt und während der Abarbeitung der Zwischencodebefehle mit Werten initialisiert und bei der *Unifikation* verglichen. Für den Spezialfall, daß ein *Unterziel* nur eine temporäre *Variable* hat, wird auf diese *Variable* direkt über ein Register zugegriffen.

Bei einem *Unterziel*-Aufruf wird eine dazupassende *Klausel* gesucht. Damit diese "passenden" *Klauseln* möglichst schnell gefunden werden, wird ein sogenanntes *Klauselindizierungs*-Verfahren (engl. "clause indexing") eingeführt. Dazu werden in einem binären Suchbaum, dem sogenannten *Funktorbaum*, der Name und die Arität der einzelnen Prozeduren abgespeichert. Als Prozedur bezeichnet man eine Gruppe von *Klauseln*, die denselben *Funktor* und dieselbe Argumentanzahl beziehungsweise Arität besitzen. Beim *Funktor* befindet sich ein Verweis auf die Liste der dazugehörenden *Klauseln*. Zusätzlich zur sequentiellen Verkettung sind die *Klauseln* für die *Klauselindizierung* nach dem Wert ihres ersten *Arguments* in einem AVL-Baum verkettet. (siehe Kapitel 3.4.3)

Die erste *Klausel* des Programmes erzeugt der Interpreter selbst, sodaß sich auf jeden Fall mindestens ein *choice point* am *environment stack* befindet, damit auch in dem Fall, daß das gesamte Prologprogramm scheitert, ein sauberer Ausstieg ins Betriebssystem gewährleistet ist.

Anhand der beiden Beispiele am Anfang dieses Kapitels wird die Abarbeitung der Prologprogramme durch die Zwischencodebefehlskombinationen beschrieben. Eine detaillierte Beschreibung des gesamten VAM$_{2P}$-Instruktionssatzes findet sich im Anhang. Die einzelnen Befehlskombinationen werden noch mit *Klauselindizierungsbefehlen* verbunden. Diese werden hier aber wegen der besseren Lesbarkeit weggelassen.

Beispiel 1:

```
test :- a(X,X).
a(A,B) :- b(A), c(B).
b(1).
c(1).
```

Die vom VAM$_{2P}$-Übersetzer generierten Zwischencodebefehle werden folgendermaßen kombiniert:

```
<Befehl des Aufrufers (goal)>-<Befehl des Aufgerufenen (head)>

<call>-<goal 0,a>
<g_fsttmp 0>-<h_fstvar 0>
<g_nxttmp 0>-<h_fstvar 1>
<lastcall>-<goal 2, b>
<g_nxtvar 0>-<h_const 1>
<call>-<nogoal>
<call>-<goal c>
<g_nxtvar 1>-<h_const 1>
<lastcall>-<nogoal>
```

Bei der Abarbeitung der Befehlskombinationen legt die VAM$_{2P}$ (<call>-<goal 0,a>) für die *Klausel* test ein *stack frame* am *environment stack* an. Für den Aufruf von a(A,B) wird ein *stack frame* am *environment stack* angelegt mit Speicherplatz für zwei lokale *Variablen*. Die Information über den Speicherplatz für lokale *Variablen* wird der Variablentabelle, die durch den VAM$_{2P}$-Übersetzer generiert worden ist, entnommen. Die temporäre *Variable* X wird in einem speziellen Speicherbereich abgespeichert. Dem Aufruf von a(X,X) folgt die *Unifikation* der *Argumente* X mit A und X mit B aus der *Klausel* a(A,B) :- b(A), c(B) durch die Zwischencodebefehle <g_fsttmp 0>-<h_fstvar 0> und <g_nxttmp 0>-<h_fstvar 1>. Weil diese *Variablen* (A, B und X) noch keinen Wert haben, wird am *copy stack* eine ungebundene *Variable* angelegt und von den Speicherzellen im *environment stack* beziehungsweise im Speicherbereich für temporäre *Variablen* referenziert. Der *Resolutionsalgorithmus* ruft durch den Zwischencodebefehl <lastcall>-<goal 2, b> das nächste *Unterziel* b(A) auf. Die ungebundene *Variable* im *copy stack* wird mit dem Wert 1 aus dem Zwischencodebefehl <g_nxtvar 0>-<h_const 1> initialisiert. Der Zwischencodebefehl <call>-<nogoal> zeigt an, daß für die *Klausel* b(1) keine weiteren *Unterziele* mehr existieren. Die letzten drei Zwischencodebefehle behandeln den Aufruf des *Unterziels* c(B) und die *Unifikation* der *Variablen* B mit 1. Die Abarbeitung der Zwischencodebefehle liefert ein TRUE-Resultat, bei dem X automatisch an den Wert 1 gebunden wird. Die genaue Darstellung der Stackinhalte wurde bereits in Kapitel 2 beschrieben.

Das zweite Beispiel zeigt die Zwischencode-Befehlsabarbeitung anhand eines rekursiven append-Beispiels mit Listen:

```
test :- append([1],[],[1]).
append([], L, L).
append([H| L1], L2, [H| L3]) :- append(L1, L2, L3).
```

Die Zwischencodebefehle werden durch die VAM$_{2P}$ während des Programmablaufs abgearbeitet, indem jeweils ein Befehl des *Unterziels* einer *Klausel* (Aufrufer) mit einem Befehl des aufgerufenen *Kopfes* einer *Klausel* (Aufgerufener) vom *heap* ausgelesen und kombiniert wird.

`append([1],[],[1])` ruft die *Klausel* `append([],L,L).` auf:
`<call>-<append/1(`*Faktum*`)>` → Zwischencodebefehl
Die *Unifikation* von `[1]` mit `[]` scheitert. Daher wird *Backtracking* eingeleitet und die nächste Alternative abgearbeitet:

`append([1],[],[1])` ruft die alternative *Klausel*
`append([H| L1], L2, [H| L3]) :- append(L1, L2, L3)` auf:
Die *Argumente* von `append([1],[],[1])` werden mit den *Argumenten* von `append([H| L1], L2, [H| L3])` unifiziert:

Zwischencodebefehlskombination	Aktionen des VAM$_{2P}$-Interpreters
`<call>-<goal 3, append>`	Die VAM$_{2P}$ ruft die alternative *Klausel* `append([H| L1], L2, [H| L3])` auf und legt ein neues *stack frame* am *environment stack* an.
`<g_list>-<h_list>`	liefert TRUE; die Bearbeitung wird bei der nächsten Befehlskombination fortgesetzt:
`<g_const 1>-<h_fsttmp 0>`	Die temporäre *Variable* `H` wird mit dem Wert `1` initialisiert
`<g_nil>-<h_fstvar 0>`	Die lokale *Variable* `L1` wird im aktuellen *stack frame* angelegt und mit einer leeren Liste initialisiert.
`<g_nil>-<h_fstvar 1>`	Die lokale *Variable* `L2` wird im aktuellen *stack frame* angelegt und mit einer leeren Liste initialisiert.
`<g_list>-<h_list>`	liefert TRUE; die Bearbeitung wird bei der nächsten Befehlskombination fortgesetzt:
`<g_const 1>-<h_nxttmp 0>`	Der Inhalt der temporären *Variablen* `H` wird mit `1` verglichen (unifiziert), was einem TRUE-Resultat entspricht.
`<g_nil>-<h_fstvar 2>`	Die lokale *Variable* `L3` wird im aktuellen *stack frame* angelegt und mit einer leeren Liste initialisiert.
`<lastcall>-<goal 3, append>`	Das *Unterziel* `append(L1, L2, L3)` ruft das *Faktum* `append([], L, L)` auf (durch *Klauselindizierung* - siehe Kapitel 3.4.3). Für die Verarbeitung dieses *Faktums* wird am *environment stack* ein *stack frame* angelegt.
`<g_nxtvar 0>-<h_nil>`	Die lokale *Variable* `L1` wird mit einer leeren Liste (nil) verglichen. → TRUE Resultat
`<g_nxtvar 1>-<h_fsttmp 0>`	Die temporäre *Variable* `H` wird mit der leeren Liste (`L2`) initialisiert.
`<g_nxtvar 2>-<h_nxttmp 0>`	Die lokale *Variable* `L3` wird mit der temporären *Variablen* `H` verglichen. → TRUE Resultat
`<lastcall>-<nogoal>`	Die Bearbeitung des Prologprogramms ist mit der Bearbeitung des *Faktums* `append` beendet und liefert ein TRUE-Resultat zurück.

3.2.1 *Unifikation* von Listen und Strukturen:

Bei der *Unifikation* von Listen oder Strukturen gibt es mehrere Möglichkeiten:

Bei der *Unifikation* einer void-*Variablen* mit einer Liste/Struktur wird die Struktur überlesen (siehe Tabelle 8). Wenn der Zwischencodeinterpreter VAM_{2P} eine `<void>-<list>` oder eine `<void>-<struct>` Befehlskombination einliest wechselt er in einen Modus, in dem nur ein Instruktionszeiger (*Kopf* oder *Unterziel*) erhöht wird, solange bis die Elemente der Liste/Struktur überlesen worden sind. Enthalten diese Elemente wieder Listen oder Strukturen, so wird dieser Überlesevorgang (skip-Modus) erneut rekursiv gestartet. Dieser rekursive Aufruf trifft auch auf die weiter unten beschriebenen write- und read-Modi der VAM_{2P} zu, falls die Listen oder Strukturen wieder Listen oder Strukturen enthalten! Für Listen müssen 2 Befehle (*Argumente*) überlesen werden (erstes Element der Liste und Restliste), bei Strukturen sind es n (Arität der Struktur) Befehle. Danach schaltet der Interpreter wieder in den ursprünglichen Modus, bei dem beide Instruktionszeiger (*Kopf* und *Unterziel*) eingelesen und erhöht werden.

Beispiel:
```
test :- listskip(X).
listskip([1]).
```

Befehle des Aufrufers (*Unterziel*)	Befehle des Aufgerufenen (*Kopf*)	Bemerkung
`listskip(X)`	`listskip([1])`	Sourcecode
g_void	h_list	VAM_{2P} wechselt in den Ein-Instruktionszeiger-Modus (skip-Modus)
	h_const 1	erstes Element der Liste überlesen
	h_nil	zweites Element der Liste überlesen
		Liste überlesen → VAM_{2P} wechselt in den Zwei-Instruktionszeiger-Modus
lastcall	nogoal	Ende der Bearbeitung

Tabelle 8

Bei der *Unifikation* des ersten Auftretens einer *Variablen* mit einer Struktur legt die VAM_{2P} die Struktur am *copy stack* (siehe Kapitel 2) an. Dies ist bei allen `<fstvar/fsttmp>-<list/struct>` Befehlskombinationen der Fall. Dabei arbeitet die VAM_{2P} ebenfalls nur mit einem Instruktionszeiger (write- oder copy-Modus), legt aber die Elemente der Liste/Struktur entsprechend den eingelesenen Befehlen am *copy stack* an (siehe Tabelle 9). Die im *stack frame* reservierte Speicherzelle für diese *Variable* erhält eine Referenz auf die Liste/Struktur im *copy stack*. Nach dem Anlegen der Liste/Struktur am *copy stack* wechselt der Interpreter wieder in den Modus mit zwei Instruktionszeigern.

Beispiel:
```
test :- listcreate(X), listunify(X).
listcreate([1]).
listunify([1]).
```

Befehle des Aufrufers (*Unterziel*)	Befehle des Aufgerufenen (*Kopf*)	Bemerkung
`listcreate(X)`	`listcreate([1])`	Sourcecode
g_fstvar X	h_list	VAM_{2P} wechselt in den Ein-Instruktionszeiger-Modus (write-Modus)
	h_const 1	erstes Element der Liste auf den *copy stack* schreiben
	h_nil	Restliste auf den *copy stack* anlegen
		Liste am *copy stack* angelegt; der Speicherzelle X wird eine Referenz auf die Liste im *copy stack* zugewiesen. → VAM_{2P} wechselt in den Zwei-Instruktionszeiger-Modus
`listunify(X)`	`listunify([1])`	siehe Tabelle 10

Tabelle 9

Die dritte Möglichkeit betrifft die *Unifikation* einer bereits aufgetretenen *Variablen* mit einer Liste/Struktur. Damit sind alle `<nxtvar/nxttmp>-<list/struct>` Befehlskombinationen gemeint. Die VAM_{2P} wechselt dann wieder in einen Ein-Instruktionszeiger-Modus (auch unify- oder read-Modus), liest die Zwischencodebefehle des *Kopfes* oder *Unterziels* der Reihe nach ein und vergleicht sie mit den Einträgen der zu unifizierenden Liste/Struktur am *copy stack*. Nach erfolgreicher *Unifikation* wechselt der Interpreter wieder den ursprünglichen Zwei-Instruktionszeiger-Modus und setzt die Abarbeitung bei der nächsten Zwischencodekombination fort (siehe Tabelle 10).

Beispiel:
```
test :- listcreate(X), listunify(X).
listcreate([1]).
listunify([1]).
```

Befehle des Aufrufers (*Unterziel*)	Befehle des Aufgerufenen (*Kopf*)	Bemerkung
`listcreate(X)`	`listcreate([1])`	siehe Tabelle 9
`listunify(X)`	`listunify([1])`	Sourcecode
g_nxtvar X	h_list	VAM_{2P} wechselt in den Ein-Instruktionszeiger-Modus (read-Modus)
	h_const 1	erstes Element der Liste vom *copy stack* auslesen und mit `1` unifizieren
	h_nil	Restliste aus dem *copy stack* auslesen und mit der leeren Liste unifizieren
		Liste unifiziert → VAM_{2P} wechselt in den Zwei-Instruktionszeiger-Modus
lastcall	nogoal	Ende der Bearbeitung

Tabelle 10

3.3 Erweiterungen

3.3.1 Garbage Collection

Bei der VAM_{2P} werden teilweise am *copy stack* und am *trail* Daten angelegt, die nach einiger Zeit nicht mehr benötigt werden. Um diese Daten aufzuspüren und die Stacks von diesem Datenmüll (garbage) zu befreien, wird momentan an einem *garbage collector* gearbeitet [Gran96].

3.3.2 Metacall

Um den Aufruf (Meta-call) von vordefinierten *Prologprädikaten* (sogenannter builtin-*Prädikate*) zu implementieren werden zur Übersetzungszeit Informationen über die Startadressen der Prozeduren gesammelt. Für einen Metacall wird eine spezieller Befehl anstatt des `goal`-Befehls generiert mit einem Parameter, der den Variablenoffset der Metacall-*Variablen* im *stack frame* enthält. Dieser Metacall wird durch einen `call`- beziehungsweise `lastcall`-Befehl abgeschlossen. Zur Laufzeit enthält diese Metacall-*Variable* ein Atom oder eine Struktur, wobei der *Funktor* der Struktur auf den Zwischencode der entsprechenden Prozedur verweist. Der `headptr` wird dann auf diese Startadresse der Prozedur "gesetzt" und der Interpreter setzt dort mit seiner Bearbeitung fort (read-Modus - siehe Kapitel 3.2.1). Der Strukturzeiger zeigt dann auf die *Argumente* der entsprechenden Prozedur. Nach Bearbeitung dieses Metacalls (Erreichen des `lastcall`-Befehls) schaltet der Interpreter wieder in den Zwei-Instruktionszeiger-Modus und bearbeitet die weiteren Zwischencodebefehle.

3.4 Optimierungen

3.4.1 Variablenklassifizierung

Die wichtigste Optimierung des VAM_{2P}-Übersetzers ist die Variablenklassifizierung (void, temporär, lokal), wie sie in Kapitel 3.1 beschrieben worden ist. Dabei handelt es sich um Speicherplatzeinsparungen. Eine void-*Variable* darf nur ein einziges Mal innerhalb einer *Klausel* vorkommen. Sie erhält nur einen Wert zugewiesen, der nie abgefragt wird. Daher kann für eine void-*Variable* die Zuweisung und die Reservierung von Speicherplatz entfallen. Die temporären *Variablen* werden in die erste temporäre *Variable* und die restlichen temporären *Variablen* eingeteilt. Die erste temporäre *Variable* wird von der VAM_{2P} in einem Register abgelegt und beschleunigt dadurch den Datenzugriff auf diese *Variable*. Weil temporäre *Variablen* nur innerhalb eines *Unterziels* oder des *Kopfes* der zu unifizierenden *Klausel* vorkommen, benötigen sie den Speicherplatz nur für diese *Unifikation*. Bei *Variablen* wird noch zwischen ihrem ersten und allen weiteren Vorkommnissen unterschieden. Die *Variable* benötigt daher nur einmal pro *Klausel* einen Speicherplatz und nicht für jedes Vorkommen.

3.4.2 Lastcall-Postoptimierung

Ein deterministisches *stack frame* kann nach dem Aufruf des letzten *Unterziels* verworfen werden. Dadurch verbraucht der *environment stack* wesentlich weniger Speicher. Ein deterministisches *stack frame* kann aber auch schon vor dem Aufruf des letzten *Unterziels* gelöscht werden, indem man nach der *Variablenunifikation* das *stack frame* der aufgerufenen *Klausel* über das *stack frame* der aufrufenden *Klausel* kopiert (siehe Abb. 8). Für diesen Zweck wird der Operand `N` vom `goal`-Befehl des ersten *Unterziels* verwendet, der die Anzahl der zu kopierenden *Variablen* angibt.

Weil diese Optimierung erst nach der vollständigen *Unifikation* erfolgen kann und erst zur Laufzeit entschieden wird, nennen wir sie Lastcall-Postoptimierung. Diese Lastcall-Optimierung ist bei der VAM_{2P}, bei der VAM_{1P} und bei der $VAM_{2\&1P}$ identisch. Bei der VAM_{1P} und bei der $VAM_{2\&1P}$ wird zusätzlich ein zweiter Lastcall-Optimierungs-Algorithmus eingeführt (siehe Kapitel 4 und 5).
Diese Optimierung ist eine reine Speicherplatzoptimierung und kostet etwas an Rechenzeit.

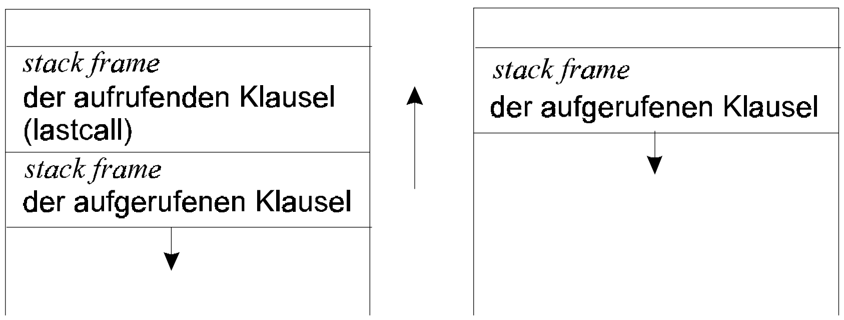


Abb. 8: Lastcall-Postoptimierung

3.4.3 *Klauselindizierung*

Das *Ziel* der Indizierung von *Prologklauseln* ist, die Anzahl der *Klauseln* die bei einem *Unterziel*-Aufruf auf "Übereinstimmung" geprüft werden zu minimieren. Das wirkt sich vor allem auf die Ausführungszeiten des Prologprogrammes aus. Der erste Schritt ist, nur *Klauseln* einer Prozedur und nicht alle *Klauseln* bei einem *Unterziel*-Aufruf auszuprobieren. Der nächste Schritt ist, nur *Klauseln* zu einem *Unterziel* aufzurufen, die mit dem ersten *Argument* unifizierbar sind.

Die VAM_{2P} speichert die Indizierungs-Informationen in geeigneten Datenstrukturen. Weil die VAM_{2P} die *Unterziel-Argumente* von links nach rechts abgearbeitet, wird nur eine "erstes-*Argument*"-Indizierung unterstützt. Für die *Klausel*-Auswahl wird ein ausbalancierter Binärbaum zu Hilfe genommen. Die Indizierungswerte sind entweder Integers, Atome oder die *Funktoren* von Strukturen. Jede *Klausel* erhält einen zusätzlichen "*header*", der einen Zeiger auf *Klauseln* mit niedrigeren Indizes, einen Zeiger auf *Klauseln* mit größeren Indizes, einen Zeiger auf *Klauseln* mit demselben Index und einen Zeiger für Linearverbindungen der *Klauseln* enthält (siehe Abb. 7).

Für jede Prozedur existiert ein "*header*", der einen Zeiger auf *Klauseln* mit nil als erstes *Argument*, einen Zeiger auf *Klauseln* mit einer Liste als erstes *Argument*, einen Zeiger auf die Wurzel des binären Suchbaumes und einen Zeiger auf die erste *Klausel* enthält.

Der ausbalancierte Suchbaum kann durch eine Hashtabelle ersetzt werden. Weil aber die Anzahl der *Klauseln* in einer Prozedur im allgemeinen eher klein ist, ist der Suchbaum schneller.

Dieser *Klauselindizierungsalgorithmus* ist auch bei der VAM_{1P} und bei der $VAM_{2\&1P}$ identisch.

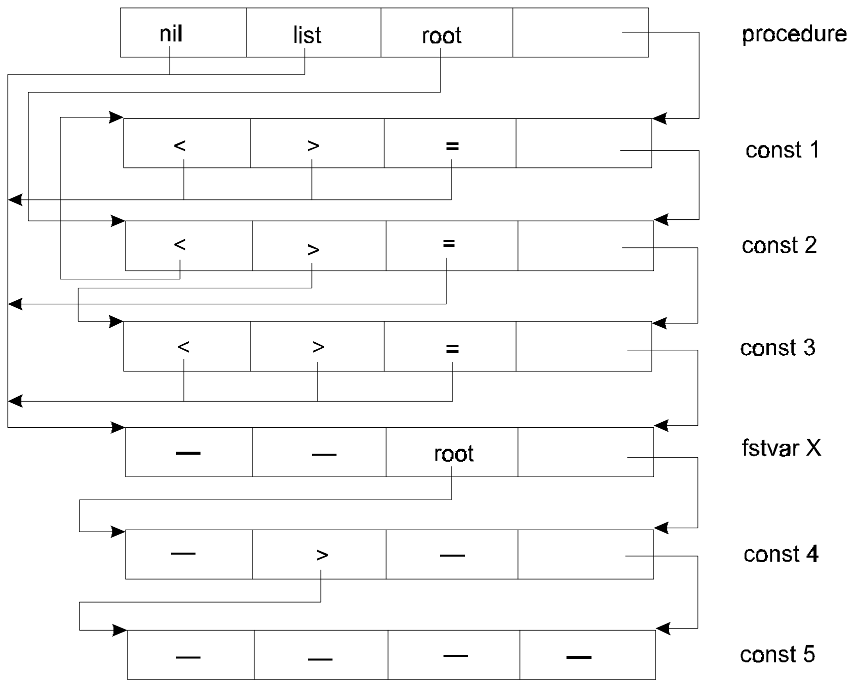


Abb. 7: *Klauselindizierung*

Um den Implementierungsaufwand für die *Klauselindizierung* im Interpreter zu reduzieren wird anstatt des `goal`-Befehls ein `xgoal`-Befehl generiert. Dieser Befehl wird nur generiert falls das erste *Argument* eine *Variable* ist. Der `xgoal`-Befehl hat dieselben Operanden und Folgebefehle wie der `goal`-Befehl. Zusätzlich wird ein Operand `V` mitgeführt, der den `Offset` der *Variablen* (erstes *Argument* der *Klausel*) im *stack frame* angibt.

3.5 Vor- und Nachteile:

Die Vorteile der VAM_{2P} liegen in ihrem kompakten Zwischencode, sowie in der Portabilität des Zwischencodes auf die verschiedensten realen Maschinen (Prozessoren). Die VAM_{2P} hat den Nachteil, daß die Zwischencodebefehle erst zur Laufzeit kombiniert und interpretiert werden, was sich in der Ausführungszeit des Prologprogrammes negativ auswirkt. Deshalb wurde ein Prolog-Compiler entwickelt, die VAM_{1P} (siehe nächstes Kapitel), der die Ausführungszeit eines Prologprogrammes gegenüber der VAM_{2P} wesentlich verkürzt.

4 VAM_{1P}

Die VAM_{1P} wurde als abstrakte Maschine entwickelt für Maschinencodeerzeugung. Die VAM_{1P} stellt einen Prolog-Compiler dar, der die Prologquellprogramme zuerst in *Terme* umwandelt und dann durch den VAM_{2P}-Übersetzer in VAM_{2P}-Zwischencodebefehle übersetzt (siehe Kapitel 3). Der VAM_{1P}-Übersetzer generiert aus diesen Zwischencodebefehlen Maschinencode eines Prozessors (reale Maschine) (siehe Abb. 9). Bei der Ausführung des Maschinencodes auf einem Prozessor können sich gegenüber der Interpretation des Prologprogrammes bei der VAM_{2P} wesentlich kürzere Laufzeiten ergeben.

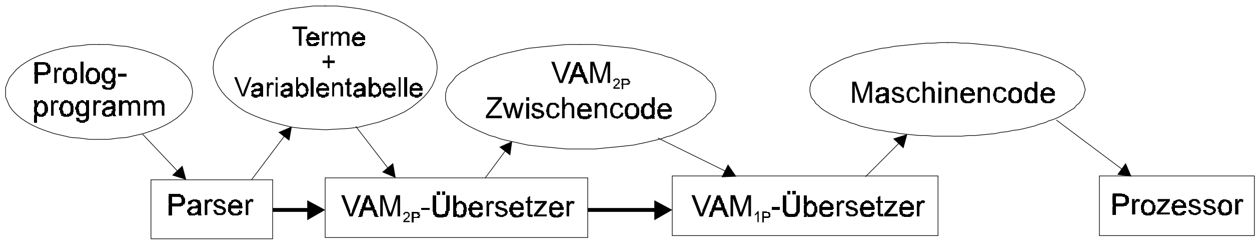


Abb. 9: VAM_{1P}-Übersetzer

Die VAM_{1P} ist in der Darstellung der Datenobjekte und in der Speicheraufteilung an die VAM_{2P} angelehnt. Der wesentliche Unterschied dieses Prologcompilers zu dem Prologinterpreter VAM_{2P} liegt darin, daß der Übersetzer die Zwischencode-Kombinationen während der Übersetzungszeit anstatt zur Laufzeit generiert und deshalb zur Laufzeit mit einem Instruktionszeiger (`codeptr`) auskommt. Der Speicherbedarf des *choice points* wird dadurch auch um einen Instruktionszeiger vermindert. Der Zeiger auf die alternativen *Klauseln* zeigt direkt auf den Codebereich der restlichen alternativen *Klauseln*. Da die Argumentübergabe und -übernahme beim Aufruf einer *Klausel* bei der VAM_{1P} gleichzeitig erfolgt, wird für jede mögliche Kombination eines *Unterziels* mit dem *Kopf* einer *Klausel* ein eigener Code erzeugt. Dies wirkt sich auf die Größe des erzeugten Codes vor allem dann sehr stark aus, wenn ein *Unterziel* sehr oft aufgerufen wird und sehr viele Alternativen für das aufgerufene *Unterziel* existieren. Auch die in Kapitel 4.1 beschriebenen Optimierungen vergrößern den erzeugten Code sehr.
Die VAM_{1P} generiert für die *Unifikation* von Listen oder Strukturen Maschinencode, der die Listen oder Strukturen zur Laufzeit am *copy stack* anlegen und unifizieren soll. Der bei der VAM_{2P} beschriebene read- und write-Modus wird bei der VAM_{1P} in Maschinencode auscompiliert. Dieser Vorgang verursacht gegenüber der VAM_{2P} eine Vergrößerung des erzeugten Codes. Zur Laufzeit werden dann die entsprechenden Maschinencodebefehle, die die Strukturen am *copy stack* anlegen und unifizieren sollen, ausgeführt.

4.1 Optimierungen

Die VAM_{1P} führt folgende Optimierungen wie sie bei der VAM_{2P} beschrieben worden sind durch: Variablenklassifizierung, *Klauselindizierung* (hier während der Übersetzungszeit) und die Lastcall-(Post-)Optimierung
Bei der VAM_{1P} werden aber noch folgende weitere Optimierungen bezüglich des Speicherbedarfs des Compilers und der Ausführungsgeschwindigkeit der übersetzten Prologprogramme eingeführt:

Weil die Befehle des Zwischencodes während der Übersetzungszeit kombiniert werden, wird für alle unnötigen Befehlskombinationen kein Code erzeugt. Dies trifft für alle Kombinationen zu, die ein "TRUE"-Resultat erzeugen. Im allgemeinen wird für eine void-*Variable* kein Code erzeugt, außer bei einer Kombination von einer void-*Variablen* mit dem ersten Auftreten einer lokalen *Variablen*.

Beispiel:

```
test1(Z) :- test2(V,Z).
test2(X,Y) :- a(X), b(Y).
a(1).
b(1).
```

<g_void>-<h_fstvar X>;	(test2(V,Z) und test2(X,Y))	
	→ generiere Maschinencode für <g_void>-<h_fstvar X>	
g_nxtvar X;	(bei a(X))	→ keine Codeerzeugung

Temporäre *Variablen* werden gänzlich eliminiert. Der *Unifikations*-Partner mit dem ersten Auftreten einer temporären *Variablen* wird direkt mit dem *Unifikations*-Partner mit dem weiteren Auftreten einer temporären *Variablen* unifiziert. Wenn die temporären *Variablen* in Strukturen vorkommen, können sie nicht so einfach eliminiert werden (siehe Kapitel 4.1.3).

Klauseln, die ein *Argument* enthalten, das zu einem FALSE-Resultat ausgewertet worden ist, werden aus der Liste der alternativen *Klauseln* entfernt und werden dann beim *Backtracking* nicht mehr aufgerufen.

Die im *Kopf* einer *Klausel* vorkommenden *Variablen* werden soweit möglich während der *Unifikation* der *Argumente* in Registern gehalten. Erst kurz vor Aufruf der Prozedur werden die in den Registern enthaltenen Werte im aktuellen *stack frame* abgelegt. Wie weit diese Optimierung ausgenützt wird hängt von der Anzahl der Register der realen Maschine ab. Ein Laufzeitgewinn ergibt sich vor allem in Kombination mit der Lastcall-Optimierung. Dabei müßten die *Variablen* sonst zwischengespeichert und vor dem Aufruf der Prozedur vom *stack frame* der aufrufenden *Klausel* kopiert werden, wodurch pro *Variable* zwei zusätzliche Speicherzugriffe notwendig wären.

4.1.1 Lastcall-Optimierung

Die VAM_{1P} implementiert zwei Versionen für die Lastcall-Optimierung. Die erste Variante ist identisch zu der in der VAM_{2P} verwendeten (Postoptimierung).

Die zweite Variante stellt die sogenannte Lastcall-Präoptimierung dar.

Präoptimierung:

Wenn für eine aufrufende *Klausel* keine Alternativen existieren (bzw. *Klausel* enthält ein Cut) und das letzte *Unterziel* der aufrufenden *Klausel* nur mehr mit genau einem *Kopf* unifiziert werden kann, dann können die *Variablen* der aufgerufenen *Klausel* (`headframe`) direkt im *stack frame* der aufrufenden *Klausel* (`goalframe`) abgelegt werden, sofern sie so umgeordnet werden, daß keine Konflikte zwischen Lese- und Schreiboperationen entstehen. Diese Umordnung ist während der Übersetzungszeit möglich.
Unifikationen, die *Variablen* mit demselben `Offset` (Stelle im *environment stack*) betreffen, können entfallen, da sie ja an derselben Stelle im Speicher abgelegt werden und im *Rumpf* der aufgerufenen *Klausel* richtig angesprochen werden.
Diese Optimierung kann bereits zur Übersetzungszeit durchgeführt werden, da als einzige Alternative das Scheitern des Aufrufs bleibt. Daher nennt man diese Optimierung Lastcall-Präoptimierung (siehe Abb. 10).

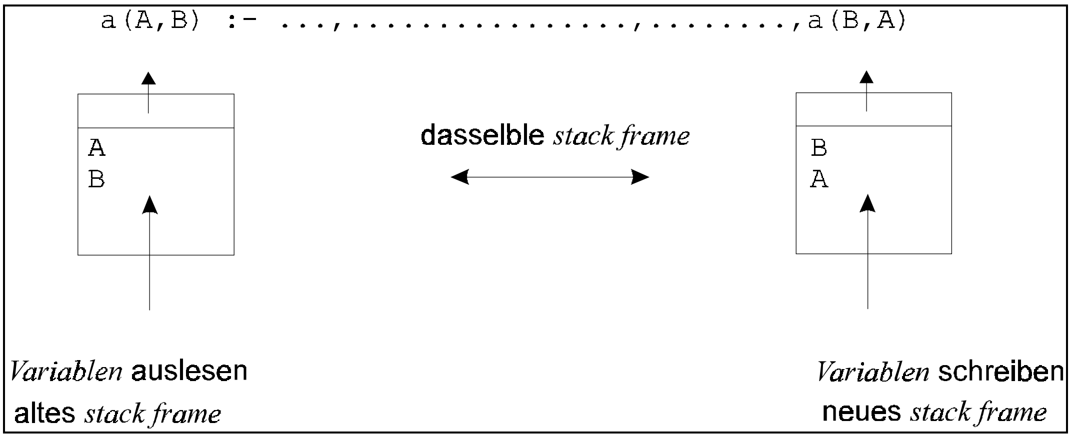


Abb. 10: Lastcall-Optimierung

4.1.2 Stacktrimming

Falls für die aktuelle *Klausel* kein *choice point* vorhanden ist, wird bei geeigneter Vergabe des `Offsets` der *Variablen* im *stack frame* jeweils vor Aufruf des nächsten *Unterziels* der Speicherplatz für die nicht mehr benötigten lokalen *Variablen* (das sind jene, die im Rest der *Klausel* nicht mehr auftreten) freigegeben. Dabei werden die `Offsets` nach dem jeweils letzten Auftreten einer lokalen *Variablen* vergeben. Die zuletzt auftretende *Variable* hat den geringsten `Offset`.

Beispiel:

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

Variable	X	Y	Z	Q
`Offset`	3	0	2	1

4.1.3 temporäre *Variablen* in Strukturen:

Temporäre *Variablen*, die nicht zur Übersetzungszeit aufgelöst werden können, da sie nur als Elemente in Listen oder Strukturen vorkommen, und diese Strukturen mit *Variablen* unifiziert werden sollen, benötigen eine gesonderte Behandlung. Der Inhalt dieser temporären *Variablen* ist zur Übersetzungszeit noch nicht bekannt. Damit kein Zwischenspeichern dieser *Variablen* zur Laufzeit nötig ist, ordnet die VAM_{1P} die *Argumente* um.

Beispiel:

Bei der *Unifikation* der *Argumente* von

`foo(...,A,...,B,...)` mit `foo(...,[X|L],...,foo1(X,_),...)`

mit `X` als temporärer *Variable* werden die *Argumente* so umgeordnet, daß `A` und `B`, beziehungsweise `[X|L]` und `foo1(X,_)`, zusammengezogen werden.

Durch eigene Befehle, die zwei Zellen am *copy stack* miteinander unifizieren, können die temporären *Variablen* direkt am *copy stack* abgelegt werden. Dadurch entfällt ein unnötiges Zwischenspeichern dieser *Variablen*.

4.1.4 Rekursionsoptimierung:

Häufig werden Daten in Prolog in Listen oder Strukturen gesammelt. Die Prozeduren zur Verarbeitung von Listen haben dabei meistens folgendes Aussehen: Mindestens eine *Klausel* enthält einen rekursiven Aufruf der Prozedur, wobei die Liste oder Struktur bei diesem Aufruf kleiner wird. Mindestens eine *Klausel* enthält eine Abbruchbedingung der Rekursion. Im Falle einer Listenverarbeitung ist diese Abbruchbedingung, daß die Liste beim rekursiven Aufruf leer ist. Prozeduren, die schrittweise Elemente von Listen oder Strukturen abspalten und verarbeiten, wobei die Größe der Struktur in jedem Schritt abnimmt, bezeichnen wir als Strukturrekursion.

Beispiel:

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

Die Prozedur `split/3` spaltet eine Liste mit einer geraden Anzahl von Elementen in eine Liste, die alle Elemente mit gerader Ordnung und eine, die alle Elemente mit ungerader Ordnung enthält. Die Abbruchbedingung ist erfüllt, sobald alle Listenelemente abgearbeitet sind. Der direkte Rekursionsschritt verkürzt jeweils alle drei Listen um ein beziehungsweise zwei Elemente.

Üblicherweise werden bei der Übersetzung der Prozedur `split/3` sowohl *Unifikations*- als auch Kontrollbefehle erzeugt. In jedem Schritt werden die drei *Variablen* im *stack frame* abgelegt, obwohl sie kurz danach, ohne dazwischen verändert zu werden, wieder geladen und dereferenziert werden. Da beim direkt rekursiven Aufruf ohnedies solange dasselbe Programmstück durchlaufen wird, bis die Liste leer ist, kann auf das Speichern der *Variablen* im *stack frame* verzichtet werden und die *Variablen* können während der gesamten Listenverarbeitung in Registern gehalten werden. Nur beim ersten Aufruf der Prozedur müssen die *Variablen* über das *stack frame* übergeben werden. Diese Optimierung bezeichnet man mit Rekursionsoptimierung.

<u>Einschränkungen:</u>

Die Rekursionsoptimierung ist auf jene Fälle beschränkt, in denen mindestens ein *Argument* eine beim Aufruf bereits gebundenen Liste oder Struktur enthält. Wird `split/3` zum Beispiel mit drei ungebundenen *Variablen* aufgerufen, dann kann die Rekursionsoptimierung nicht sinnvoll eingesetzt werden, weil nur jeweils ein Rekursionsschritt ausgeführt und die Prozedur sofort danach wieder verlassen wird.

Die Rekursionsoptimierung ist außerdem auf direkt rekursive Aufrufe beschränkt, wobei der *Rumpf* der rekursiven *Klausel* nur diesen direkt rekursiven Aufruf enthalten darf. Enthielte der *Rumpf* mehrere *Unterziele*, so könnte die in der Rekursion kleiner werdende Struktur in einem dieser *Unterziele* verwendet werden, wodurch eine Speicherung der *Variable*, die diese Struktur enthält, im *stack frame* notwendig wäre. Auch die Bedingung, daß diese *Variable* nur im *Kopf* und im rekursiven Aufruf vorkommt, ist zu schwach, da sie durch Aliasing (namentlich verschiedene *Variable* enthalten Zeiger auf dieselbe Speicherstelle; entsteht durch temporäre *Variablen* im aufrufenden *Unterziel*) beim Aufruf unter einem anderen Namen in einem der anderen *Unterziele* vorkommen könnte.

4.2 VAM_{1P}-Befehlssatz:

Der Zwischencode des ersten Durchlaufes des VAM_{1P}-Übersetzers entspricht dem der VAM_{2P}. Der VAM_{1P}-Compiler übersetzt den Zwischencode in Zwischencodebefehlskombinationen, die den VAM_{1P}-Befehlssatz darstellen.

Der Befehlssatz wird aufgeteilt in *Unifikations*-, *Strukturunifikations*-, Strukturerzeugungs-, Kontroll-, *Klauselindizierungs*- und Optimierungs-Befehle.

Es werden hier nur die wesentlichen Unterschiede des VAM_{1P} Befehlssatzes (Zwischencode-Befehls-Kombinationen) zur VAM_{2P} beschrieben. Eine detaillierte Beschreibung findet man im Anhang. Die *Unifikationsbefehle* der VAM_{1P} sind nahezu identisch mit denen der VAM_{2P}. Die restlichen Befehle steuern explizit die *Strukturunifikation*, die Strukturerzeugung, die *Klauselindizierung*, die einzelnen Optimierungen und die *Resolution* (Kontrollbefehle) des Prologprogrammes durch einen Makroaufruf, der den Zwischencode interpretieren und ausführen kann (Interpreter-Version) oder den Zwischencode in Maschinencode der realen Maschine übersetzt (Compiler-Version). Bei der VAM_{2P} wurden die zuletzt genannten Befehle implizit vom Interpreter beim jeweiligen Zwischencodebefehl ausgeführt.

Alle nicht angeführten Kombinationen scheitern oder können bereits zur Übersetzungszeit aufgelöst werden. Die Zwischencodekombinationen sind symmetrisch, das heißt der Zwischencode der aufgerufenen *Klausel* und der Zwischencode der aufrufenden *Klausel* können bei gleichbleibender Aktion vertauscht werden.

Die *Unifikations*-Befehle (siehe Tabelle 11) behandeln die *Unifikation* eines *Arguments* vom aufrufenden *Unterziel* mit dem zugehörigen *Argument* von der aufgerufenen *Klausel*. Zu beachten ist, daß hier keine unterschiedliche Codierung der "head"- und "goal"-*Unifikationsbefehle* stattfindet wie bei der VAM_{2P}. So werden zum Beispiel beide *Unifikations*-Möglichkeiten einer Konstanten mit einer lokalen Variablen (g_const-h_fstvar bzw. g_fstvar-h_const) jeweils in den VAM_{1P}-*Unfikations*-Befehl `fstvar_const` übersetzt.
Wird eine Struktur mit dem ersten Auftreten einer lokalen *Variablen* kombiniert (fstvar_struct/fstvar_list), so wird die Struktur- oder Listenerzeugung gestartet. Der Operand `Size` gibt die Größe der Struktur (bzw. Liste) samt allen Unterstrukturen an. Der Operand `Reg` bei den VAM_{1P}-Befehlen nxtvar_struct/nxtvar_list und unify_list/struct ist ein Register, welches einen Zeiger auf die Struktur/Liste enthält. Der Operand `Offset` gibt die Speicherstelle relativ zum aktuellen `copyptr` an, an der die Liste beziehungsweise Struktur gespeichert wird/ist (`copyptr` + `Offset`).
Die *Strukturunifikations*-Befehle (siehe Tabelle 12) werden verwendet, um Elemente einer neuen Struktur mit den Elementen einer bereits existierenden Struktur zu unifizieren. Der Operand `CAddr` gibt das Sprungziel zu den Strukturerzeugungs-Befehlen an.
Die Strukturerzeugungs-Befehle (siehe Tabelle 13) werden benutzt, um Strukturelemente beziehungsweise Listen auf den *copy stack* zu kopieren.
Die Kontroll-Befehle (siehe Tabelle 14) beinhalten die Stack-Bearbeitung, steuern den Programmfluß und die notwendigen Änderungen im *stack frame*.
Die *Klauselindizierungs*-Befehle (siehe Tabelle 15) werden für die *Klauselindizierung* benötigt (siehe auch Kapitel 3.4.3).

Die Befehle für die Rekursionsoptimierung bei der VAM_{1P} werden für Schleifenoptimierungen (bei Listen und Strukturen) verwendet (siehe Tabelle 16).
Die Befehle für die Entfernung von temporären *Variablen* in Strukturen sind in Tabelle 17 angeführt.

***Unifikations*-Befehle**
fstvar_const Var, Const
nxtvar_const Var, Const
fstvar_nil Var
nxtvar_nil Var
fstvar_list Var, Size
nxtvar_list Var, Reg, Size, CAddr
fstvar_struct Functor, Var, Size
nxtvar_struct Functor, Var, Reg, Size, CAddr
fstvar_fstvar Var1, Var2
nxtvar_fstvar Var1, Var2
nxtvar_nxtvar Var1, Var2
fstvar_void Var

Tabelle 11

***Strukturunifikations*-Befehle**
unify_const Const, Offset
unify_nil Offset
unify_list Reg, Size, Offset, CAddr
unify_struct Functor, Reg, Size, Offset, CAddr
unify_fstvar Var, Offset
unify_nxtvar Var, Offset

Tabelle 12

Strukturerzeugungs-Befehle
create_const Const, Offset
create_nil Offset
create_list Offset1, Offset2
create_struct Functor, Offset1, Offset2
create_fstvar Var, Offset
create_nxtvar Var, Offset

Tabelle 13

Kontroll-Befehle
alloc_stackframe VarCount
adjust VarCount
store_cont_goto ContAddr, ProcAddr
read_cont_goto
copy_cont_goto ProcAddr
goto Addr
store_var Reg, Offset
copy_var Offset
push_choice_point AltAddr
cut VarCount
call_bip BipId
inline_bip BipId

Tabelle 14

***Klauselindizierungs*-Befehle**
deref Var
if_undef label_of_firstclause
if_const label_of_firstclause
if_nil label_of_firstclause
if_list label_of_firstclause
if_struct label_of_firstclause
comp_const const, address_of_<_funct, address_of_>_funct
comp_struct functor, address_of_<_funct, address_of_>_funct
const_undef const
nil_undef
list_undef Size
struct_undef Size
list_val Reg, Var
struct_val Reg, Var

Tabelle 15

Optimierungs-Befehle(1. Teil)
extract_list Reg, LAddr, ContAddr
construct_list Offset
construct_extract_list Reg, LAddr, UAddr
extract_struct Functor, Reg, Size, SAddr, ContAddr
construct_struct Functor, Size, Offset
construct_extract_struct Functor, Reg, Size, SAddr, UAddr

Tabelle 16

Optimierungs-Befehle(2. Teil)
create_undef Offset
unify_create Offset, Offset
unify_unify Offset1, Offset2
create_ref Offset1, Offset2

Tabelle 17

Als Beispiel für die Übersetzung von Prologquelltext in VAM_{1P}-Befehle wird das Prologprogramm mit der `append`-Prozedur aus Kapitel 3 angegeben [Berger91]:
Dieses Beispiel (siehe Seite 39) ist nicht dafür gedacht, die einzelnen Befehle bis ins kleinste Detail zu verstehen, sondern soll nur veranschaulichen, wie schnell der VAM_{1P}-Code gegenüber dem VAM_{2P}-Code wächst:

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

Zwischencodebefehlskombination	Bemerkungen
append_3_1:	der Code für die *Unifikation* steht bereits im aufrufenden *Unterziel*, daher scheint append([],L,L) hier nicht auf
append_3_2:	:- append(L1,L2,L3)
append_3_2_last_1:	dereferenziere L1 (Variablenindizierung)
deref(goalframeptr[0]	Indizierungsbefehle
if_list(append_3_xgoal_2)	
if_nil(append_3_xgoal_3)	
if_undef(append_3_xgoal_4)	
goto(fail)	
append_3_xgoal_4:	
push_choice_point(append_3_2_last_2)	Wahlpunkt anlegen
nil_undef	L1 = []
append_3_xgoal_3:	
nxtvar_nxtvar(goalframeptr[1], goalframeptr[2]	L2 = L3
read_cont(R1)	
goto R1	
append_3_2_last_2:	2. *Klausel*
list_undef(2)	
goto(append_3_copy_5)	
append_3_xgoal_2:	
adjust(3)	
list_val(R0)	
list_nxtvar(R1, 2, goalframeptr[3], append_3_xgoal7)	
append_3_xgoal_8:	Rekursionsoptimierung Fall: L1 gebunden, L3 gebunden
unify_unify (R1+0, R0+0)	unifiziere die ersten Elemente
extract_list(R0, R0+1, append_3_xgoal_9)	
construct_extract_list(R1, R1+1, append_3_xgoal_10)	
goto(append_3_xgoal_8)	
append_xgoal_9:	
unify_fstvar(goalframeptr[1], R0+1)	
unify_fstvar(goalframeptr[3], R1+1)	
append_3_copy_11:	rekursiver Aufruf
goto(append_3_2)	
append_xgoal_11:	
unify_fstvar(goalframeptr[1],R0+1	
copy_fstvar(goaframeptr[3],1)	
goto (append_3_2)	rekursiver Aufruf
append_3_xgoal_10:	
append_3_xgoal_7:	Rekursionsoptimierung Fall: L1 gebunden, L3 wird erst aufgebaut
unify_copy(R1+0,0)	
extract_list(R0,R0+1, append_3_xgoal_11)	
construct_list(1)	
goto (append_3_xgoal_7)	
append_3_copy_5:	
list_nxtvar(R0, goalframeptr[3], append_3_copy_12)	
unify_fstvar(goalframeptr[3], R0+1)	
unify_copy(R0+0,0)	
copy_fstvar(goalframeptr[1], 1)	
goto(append_3_copy_11)	
append_3_copy_12:	beide Listen erzeugen (L1 und L3)
copy_undef(0)	undefinierte Speicherzelle für H anlegen
copy_fstvar(goalframeptr[3], 1)	
copy_ref(0, 2)	Referenz auf temporäre *Variable* H aus L3
copy_fstvar(goalframeptr[1], 3)	
goto(append_3_copy_11)	

4.3 Realisierung der abstrakten Maschine VAM_{1P}:

Es gibt zwei Möglichkeiten, die abstrakte Maschine VAM_{1P} auf einer realen Maschine zu implementieren:

- Übersetzung in Maschinencode
- einen Zwischencodeinterpreter

Die VAM_{1P} wurde für die Erzeugung von Maschinencode entworfen (Variante 1). Der Maschinencode wird in einem weiteren Übersetzungsschritt mittels eines Makroprozessors auf Assemblercode expandiert, dieser wird dann noch um eine Laufzeitumgebung erweitert, die Speicher für den Stack anfordert, die Register der abstrakten Maschine initialisiert und die Hashtabelle für die Atome aufbaut. Bei der zweiten Variante wird anstatt des Maschinencodes Zwischencode erzeugt, der von einem Interpreter zur Laufzeit abgearbeitet werden kann (so wie bei der VAM_{2P}). Das Aufspalten der Strukturbearbeitung in Strukturerzeugungs- und *Strukturunifikations*-Befehle ist dann nicht mehr notwendig, weil diese Bearbeitung der Interpreter zur Laufzeit macht. Durch den im Verhältnis zur Maschinencodevariante höheren Verwaltungsaufwand, vor allem im Hinblick auf die Auswertung der Befehle und deren Parameter zur Laufzeit, ist diese Methode langsamer als der Maschinencode. Die Laufzeitersparnis dieser Methode gegenüber dem VAM_{2P}-Interpreter ist sehr gering (siehe Kapitel 6).

4.4 Vor- und Nachteile:

Der Vorteil des VAM_{1P}-Compilers liegt in der hohen Ausführungsgeschwindigkeit der übersetzten Prologprogramme. Ein wesentlicher Nachteil liegt in der Größe des erzeugten Zwischencodes und im dadurch erhöhten Speicherplatzbedarf (*heap*) des Compilers. Ein weiterer Nachteil ist, daß die vom Compiler erzeugten Maschinen-Befehle nur auf einen bestimmten Prozessortyp zugeschnitten sind. Das heißt es muß für jeden Prozessortyp ein spezieller Maschinencode generiert werden. Der Code unterstützt dann eventuell bei einem Nachfolger eines Prozessors nicht die neuen Verbesserungen des Prozessors, was ebenfalls eine Überarbeitung des zu erzeugenden Maschinencodes und der VAM_{1P}-Befehle zur Folge hat.

5 $VAM_{2\&1P}$

Die $VAM_{2\&1P}$ wurde nach folgenden Überlegungen entworfen:

Der Vorteil der VAM_{2P} liegt in ihrer kompakten Zwischencodegröße. Der Vorteil der VAM_{1P} liegt in ihrer hohen Ausführungsgeschwindigkeit der Prolog-Progamme. Die $VAM_{2\&1P}$ stellt eine Kombination dieser beiden abstrakten Maschinen dar. Die Idee ist die, die VAM_{1P} nur für selbstrekursive Aufrufe des letzten *Unterziels* einer *Klausel* zu verwenden und sonst die VAM_{2P}. Der modifizierte VAM_{1P}-Code wird dann durch den Zwischencodeinterpreter $VAM_{2\&1P}$ ausgeführt. Diese Vorgehensweise beschleunigt die Ausführung der oft benötigten Schleifen und verwendet die kompakte Zwischencode-Representation für die anderen Teile eines Prolog-Programmes. Die Datenstrukturen der $VAM_{2\&1P}$ entsprechen den beschriebenen in Kapitel 2. Die Arbeitsweise des Zwischencodeinterpreters $VAM_{2\&1P}$ entspricht der VAM_{2P}.

5.1 Zwischencode-Erzeugung

Die Übersetzung der *Terme* (Prologprogramm) in Zwischencodebefehle erfolgt in zwei Durchläufen, wie bei der VAM_{2P}. Der erste Durchlauf durchsucht die *Terme* nach *Variablen* und speichert die Informationen in einer Variablentabelle. Der zweite Durchlauf generiert aus den *Termen* die Zwischencodebefehle der VAM_{2P}. Der VAM_{2P}-Übersetzer bestimmt zwischen diesen beiden Durchläufen die Klassen (void, temporär, lokal) und die `Offsets` der *Variablen*. Handelt es sich bei dem eingelesenen *Term* um ein deterministisches, selbstrekursives *Unterziel* und ist es das letzte, dann erzeugt der VAM_{1P}-Übersetzer für dieses *Unterziel* zusätzlich VAM_{1P}-Zwischencode (siehe Abb. 11). Für die Erzeugung des VAM_{1P}-Zwischencodes kombiniert der VAM_{1P}-Übersetzer den bereits erzeugten VAM_{2P}-Zwischencode vom selbstrekursiven *Unterziel* mit dem *Kopf* der *Klausel*. Zur Laufzeit führt der $VAM_{2\&1P}$-Interpreter VAM_{2P}-Code aus. Nur bei der Ausführung des letzten *Unterziels* einer *Klausel* prüft der $VAM_{2\&1P}$-Interpreter ob dieses *Unterziel* selbstrekursiv und deterministisch ist. Wenn das der Fall ist, führt der $VAM_{2\&1P}$-Interpreter den erzeugten VAM_{1P}-Code (nur für das letzte *Unterziel* !) aus, sonst den VAM_{2P}-Code.

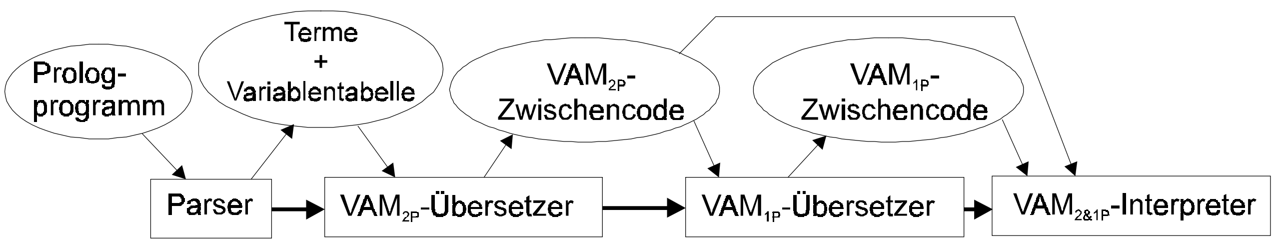


Abb. 11: $VAM_{2\&1P}$

5.2 Befehlssatz der $VAM_{2\&1P}$

Die Zwischencodebefehle enthalten den VAM_{2P}-Befehlssatz und sind um folgende VAM_{1P}-Befehle erweitert:

Die Kontroll- und Abschlußbefehle der VAM_{2P} werden um die Befehle `v1_xgoal` und `v1_lastcall` erweitert (siehe Tabelle 18).

Befehl	**Operand(en)**	**Bedeutung**
v1_xgoal	[headvarcount] procptr, headptr, v1_codeptr varoffset	letztes und selbstrekrusives, deterministisches *Unterziel* einer *Klausel*; (VAM_{2P}-Befehl) bewirkt zur Laufzeit das Umschalten in den Ein- oder Zwei-Instruktionszeiger-Modus des Interpreters (siehe Kapitel 5.3)
v1_lastcall	goalptr	Ende des letzten *Unterziels* (VAM_{1P}-Befehl)

Tabelle 18: Kontrollbefehle

Falls es sich beim eingelesenen *Term* um ein selbstrekursives *Unterziel* handelt, und es das letzte in der *Klausel* ist, dann wird zusätzlich zum VAM_{2P}-Code ein VAM_{1P}-Code erzeugt. Diese VAM_{1P}-Befehle entsprechen im Wesentlichen den zur Übersetzungszeit kombinierten VAM_{2P}-Befehlskombinationen.
Die *Unifikation* der einzelnen *Argumente* in der Selbstrekursion wird mit einem `v1_xgoal`-Befehl eingeleitet und endet mit einem `v1_lastcall`-Befehl (siehe Tabelle 18). Der optionale Operand `headvarcount` beim `v1_xgoal`-Befehl wird wie bei der VAM_{2P} für die Lastcall-Optimierung benötigt. Er gibt die Anzahl der zu kopierenden *Variablen* im *stack frame* an. Der `v1_codeptr` zeigt auf die VAM_{1P}-Befehle für diese Selbstrekursion im *heap*. Ob diese Selbstrekursion deterministisch ist, muß zur Laufzeit vom $VAM_{2\&1P}$-Interpreter überprüft werden. Der Operand `varoffset` bezeichnet den Variablenoffset des im `xgoal`-Befehl inkludierten `nxtvar`-Kommandos (*Kauselindizierung*). Um den VAM_{2P}-Code im nicht deterministischen Fall ausführen zu können, wird beim `v1_xgoal` Befehl auch der Zeiger auf die nächste *Unterziel*-Instruktion (`procptr`) in einem weiteren Operanden gespeichert (wird auch für den Aufruf der *Klauselindizierung* benötigt). Der Operand `headptr` wird zur Feststellung der Selbstrekursion benötigt. Die VAM_{1P}-Code-Sequenz endet mit einem `v1_lastcall` Befehl, dessen Operand `goalptr` den Zeiger auf die nächste *Unterziel*-Instruktion angibt.

Die *Unifikations*-Instruktionen (hier nur für Selbstrekursionen) entsprechen dem VAM_{1P}-Code so wie er in Kapitel 4 beschrieben worden ist mit folgenden Ausnahmen:

Weil Listen und Strukturen rekursiv aufgebaut sind, müssen zu ihrer Abarbeitung rekursive Prozeduren verwendet werden. Da aber auf Grund der Struktur des Interpreters (Registervariablen) keine echten rekursiven Prozeduren verwendet werden können. wird die Rekursion zur Laufzeit aufgelöst und der Stack explizit behandelt (Push und Pop). Die Listen und Strukturen werden zur Laufzeit vom $VAM_{2\&1P}$-Interpreter am *copy stack* angelegt. Nach der erfolgreichen *Unifikation* werden sie dann wieder vom *copy stack* entfernt. Damit der $VAM_{2\&1P}$-Interpreter das tun kann, wird vom VAM_{1P}-Übersetzer für jede Unterstruktur in einer Liste/Struktur ein `pop`-Befehl generiert, der die Unterstruktur vom *copy stack* entfernt. Für die optimierte Listen- und Strukturbearbeitung gibt es noch die `unify_last_list` und `unify_last_struct` Befehle, die in Kapitel 5.4.2 beschrieben werden. Bei den *Strukturunifikationsbefehlen* wird der *Funktor* der Struktur als Operand mitgeführt (siehe Tabelle 19 und 20). Im Gegensatz zur VAM_{1P} werden hier bei der $VAM_{2\&1P}$ die `Offsets` der zu unifizierenden Listen-/Strukturelemente nicht explizit relativ zum *copy stack* angegeben, sondern werden intern vom $VAM_{2\&1P}$-Interpreter zur Laufzeit berechnet (relativ zum aktuellen *stack frame*).

Die void-*Variablen* können zur Übersetzungszeit aufgelöst werden, außer bei der Kombination mit dem ersten Auftreten einer temporären oder lokalen *Variablen* in der *Klausel*. Die void-*Variablen* können in Strukturen nicht zur Übersetzungszeit, sondern erst zur Laufzeit wegoptimiert werden. Dafür muß der VAM_{1P}-Übersetzer die entsprechenden `void_fstvar`-Befehle generieren (siehe Tabelle 20).
Für Befehlskombinationen mit temporären *Variablen* wird ein Code (siehe Tabelle 20) generiert, im Gegensatz zur VAM_{1P}. Bei der Strukturrekursion werden die temporären *Variablen* nicht wegoptimiert.
Die Kontroll-, *Klauselindizierungs*- und Optimierungs-Befehle, wie sie bei der VAM_{1P} beschrieben worden sind, werden hier nicht generiert, weil der $VAM_{2\&1P}$-Interpreter diese Funktionen implizit bei der Zwischencodeinterpretation ausführt.

Der Operand `Offset`, `Offset1` oder `Offset2` gibt jeweils die Speicherstelle im aktuellen *stack frame* für lokale *Variablen* an. Bei temporären *Variablen* bezeichnet der Offset die Speicherstelle in dem für diese Variablenklasse vorgesehenen Speicherbereich.

Eine genaue Beschreibung der $VAM_{2\&1P}$-Befehle findet sich im Anhang.

Befehl	**Operand**	**Bedeutung** *Unifikation* des aktuellen Listen-/Strukturelements am *copy stack* mit ...
unify_const	Konstante C	... einer Konstanten C
unify_nil	keine	... der leeren Liste
unify_void	keine	Überlesen des Listen-/Strukturelements am *copy stack*
unify_fstvar	Offset	... einer lokalen *Variablen*
unify_nxtvar	Offset	... einer lokalen *Variablen*
unify_fsttmp	Offset	... einer temporären *Variablen*
unify_nxttmp	Offset	... einer temporären *Variablen*
unify_list	keine	... einer Unterliste in einer Liste
unify_struct	functor	... einer Unterstruktur in einer Struktur
unify_last_list	keine	... einer Unterliste in einer Liste (Optimierung)
unify_last_struct	functor	... einer Unterstruktur in einer Struktur (Optimierung)
pop	keine	Unterliste/Struktur vom *copy stack* entfernen (nur bei unify_list und unify_struct)

Tabelle 19: Listen-/Struktur-*Unifikationsbefehle*

Befehl	**Operand(en)**	**Bedeutung:** *Unifikation* einer temporären/lokalen *Variablen* mit einer...
const_fsttmp/fstvar	Konstante C, Offset	...Konstanten
const_nxttmp/nxtvar	Konstante C, Offset	...Konstanten
nil_fsttmp/fstvar	Offset	...leeren Liste
nil_nxttmp/nxtvar	Offset	...leeren Liste
void_fsttmp/fstvar	Offset	...void-*Variablen*
list_fsttmp/fstvar	Offset	...Liste
list_nxttmp/nxtvar	Offset	...Liste
fsttmp/fstvar_struct	Offset, functor	...Struktur
nxttmp/nxtvar_struct	Offset, functor	...Struktur
fstvar_fstvar/nxtvar	Offset1, Offset2	...lokalen *Variablen*
fsttmp_fstvar/nxtvar	Offset1, Offset2	...lokalen *Variablen*
nxtvar_nxtvar	Offset1, Offset2	...lokalen *Variablen*
nxttmp_fstvar/nxtvar	Offset1, Offset2	...lokalen *Variablen*

Tabelle 20: *Unifikationsbefehle* für temporäre/lokale *Variablen*

Beispiel: VAM$_{1P}$- und VAM$_{2P}$-Zwischencode für append-Prozedur (nur für Selbstrekursion)

```
test :- append([1,2],[],[1,2]).
append([], L, L).
```
`append([H| L1], L2, [H| L3]) :- append(L1, L2, L3).`

VAM$_{2P}$-Zwischencode:

```
append([          h_list
H|                h_fsttmp 0
L1],              h_fstvar 0
L2,               h_fstvar 1
[                 h_list
H|                h_nxttmp 0
L3]) :-           h_fstvar 2
append(           v1_xgoal 3, procptr,headptr,v1_codeptr, varoffset
L1,               g_nxtvar 0
L2,               g_nxtvar 1
L3                g_nxtvar 2
).                lastcall
```

VAM$_{1P}$-Befehle:

```
<Befehl>          <Operand>
nxtvar_list       0
unify_fsttmp      0
unify_fstvar      0
nxtvar_list       2
unify_fsttmp      0
unify_fstvar      2
v1_lastcall       goalptr
```

Zu beachten ist, daß der VAM$_{1P}$-Übersetzer *Variablenunifikationsbefehle*, die dasselbe `Offset` betreffen wegoptimiert, weil ja ohnehin schon der richtige Wert im aktuellen *stack frame* steht.

5.3 Zwischencodeinterpreter/Compiler $VAM_{2\&1P}$:

Der $VAM_{2\&1P}$-Interpreter entscheidet beim Zwischencodebefehl `<lastcall>-<v1_xgoal>` ob es sich beim letzten Aufruf des *Unterziels* um ein selbstrekursives und deterministisches *Unterziel* einer *Klausel* handelt. Ist das nicht der Fall, werden weiterhin die VAM_{2P}-Befehlskombinationen mit zwei Instruktionszeigern aus dem *heap* ausgelesen und interpretiert. Bei einem selbstrekursiven *Unterziel* (deterministisch) liest der $VAM_{2\&1P}$-Interpreter die VAM_{1P}-Befehlskombinationen aus dem *heap*, indem auf die Befehle mit nur einem Instruktionszeiger zugegriffen wird. Dabei wird (nur) das letzte *Unterziel* mit einem Instruktionszeiger schneller abgearbeitet als mit zwei Instruktionszeigern (siehe Ergebnisse Kapitel 6). Nach dem `v1-lastcall`-Kommando wechselt der $VAM_{2\&1P}$-Interpreter wieder in den Zwei-Instruktionszeiger-Modus. Beim nächsten Schleifendurchlauf wird wieder geprüft, ob das letzte *Unterziel* selbstrekursiv und deterministisch ist, und die oben genannten Schritte wiederholen sich.

Beispiel:

```
test :- append([1,2],[],[1,2]).
append([], L, L).
append([H| L1], L2, [H| L3]) :- append(L1, L2, L3).
```

Zu Beginn wird die temporäre *Variable* `H` mit dem Wert `1`, die lokale *Variablen* `L1` und `L3` werden mit der Liste `[2]`und `L2` wird mit der leeren Liste initialisiert. Die *Klauselindizierung* "erkennt", daß nur die zweite append-*Klausel* "passen" kann. Es wird versucht die *Argumente* des selbstrekursiven, deterministischen und letzten *Unterziels* `append(L1,L2,L3)` mit den *Argumenten* des *Kopfes* derselben append-*Klausel* zu unifizieren. Zur Laufzeit erkennt der Zwischencodeinterpreter, daß der selbstrekursive Aufruf des letzten *Unterziels* dieser *Klausel* deterministisch ist und schaltet daher in einen Ein-Instruktionszeiger-Modus bei dem die VAM_{1P}-Instruktionen abgearbeitet werden. Nach Abarbeitung der VAM_{1P}-Befehle in demselben *stack frame*, initialisiert der $VAM_{2\&1P}$-Interpreter beim `v1-lastcall`-Befehl die zwei Instruktionszeiger für den nächsten zu interpretierenden VAM_{2P}-Befehl und schaltet in den Zwei-Instruktionszeiger-Modus zurück. Der nächste Versuch die *Argumente* vom *Unterziel* `append(L1,L2,L3)` mit den *Argumenten* des *Faktums* `append([],L,L)`zu unifizieren liefert ein TRUE-Ergebnis und das Prolog-Programm ist bewiesen. Im folgenden Absatz werden die Unterschiede der VAM_{2P}-Befehle zu den VAM_{1P}-Befehlen für die selbstrekursive append-*Klausel* genauer dargestellt:

VAM_{2P}-Befehle (für Selbstrekursion):

<Aufrufer>	<Aufgerufener>	<Operand(en)>	<Interpreter-Modus>
g_nxtvar	h_list	0	combine
	h_fsttmp	0	read
	h_fstvar	0	read
g_nxtvar	h_fstvar	1, 1	combine
g_nxtvar	h_list	2	combine
	h_nxttmp	0	write
	h_fstvar	2	write
lastcall	goal ...		combine

VAM_{1P}-Befehle (für Selbstrekursion):

`<Befehl>`	`<Operand>`	<Interpreter-Modus>
`nxtvar_list`	`0`	VAM_{1P}
`unify_fsttmp`	`0`	read
`unify_fstvar`	`0`	read
`nxtvar_list`	`2`	VAM_{1P}
`unify_fsttmp`	`0`	write
`unify_fstvar`	`2`	write
`v1_lastcall`	`goalptr`	Umschalten in combine Modus

Werden die VAM_{1P}-Befehle anstatt der VAM_{2P}-Befehle interpretiert, so werden bei dem Beispiel oben nur 6 statt 10 Befehle (ohne Lastcall- und goal-Befehl) pro Schleifendurchlauf vom *heap* ausgelesen und interpretiert. Die *Klauselindizierung* führt implizit bereits die VAM_{1P}-Befehle `nxtvar_list` (mit Hilfe des Operanden `varoffset` beim `v1_xgoal`-Befehl) aus und "überspringt" sie (siehe auch Kapitel 3.4.3). Durch die Prä-Lastcall-Optimierung (siehe Kapitel 5.4.1) kann bei der $VAM_{2\&1P}$ gegenüber der VAM_{2P} auf das Anlegen neuer *stack frames* bei jedem Schleifendurchlauf verzichtet werden, indem immer im aktuellen *stack frame* die *continuation* und die Variableneinträge aktualisiert werden, was Speicherplatz am *environment stack* einspart und etwas an Laufzeitgewinn bringt.

5.3.1 Verschiedene Betriebsarten des $VAM_{2\&1P}$-Interpreters:

Beim combine-Modus arbeitet der $VAM_{2\&1P}$-Interpreter in einem Zwei-Instruktionszeiger-Modus, indem er jeweils eine *Unterziel*- und eine *Kopfinstruktion* liest und interpretiert. Bei der *Unifikation* von Listen- und Strukturen mit einer void-*Variablen* schaltet der Interpreter in einen Ein-Instruktionszeiger-Modus und überliest dabei die Liste oder Struktur (skip-Modus, siehe auch Kapitel 3.2.1). Bei der *Unifikation* einer Liste oder Struktur mit einer lokalen oder temporären *Variablen* (erstes Auftreten) wechselt der Interpreter ebenfalls in einen Ein-Instruktionszeiger-Modus (VAM_{1P}-Modus), liest die Elemente der Liste oder Struktur und legt sie am *copy stack* an (write-Modus). Wird eine Liste oder Struktur mit einer bereits aufgetretenen *Variablen* in einer *Klausel* unifiziert, dann schaltet der $VAM_{2\&1P}$-Interpreter wieder in einen Ein-Instruktionszeiger-Modus (read-Modus) und unifiziert dabei die Elemente der Liste/Struktur mit denen am *copy stack*. Nach Behandlung der Liste/Struktur schaltet der Interpreter wieder in den combine-Modus.

Bei einem `v1-xgoal` Befehl sucht der $VAM_{2\&1P}$-Interpreter mit Hilfe des Operanden `procptr` und der darauffolgenden *Klauselindizierung* die nächste passende *Klausel* für das aktuelle *Unterziel*. Ist der lastcall-Aufruf des letzten *Unterziels* der aktuellen *Klausel* deterministisch und selbstrekursiv (Operand `headptr` ist identisch mit dem headptr der gefundenen *Klausel* durch *Klauselindizierung*), dann schaltet der Interpreter in einen Ein-Instruktionszeiger-Modus und bearbeitet die VAM_{1P}-Befehle. Nach Verarbeitung des `v1_lastcall`-Befehls, bei dem die Fortsetzungszeiger in den VAM_{2P}-Code ermittelt werden, schaltet der Interpreter wieder in den combine-Modus.

5.4 Optimierungen:

Die bei der VAM_{2P} beschriebenen Optimierungen funktionieren hier analog. Zusätzlich wurde die VAM_{1P} Lastcall-Präoptimierungen hier realisiert:

5.4.1 Lastcall-Präoptimierung:

Die Lastcall-Präoptimierung wird auch vom $VAM_{2\&1P}$-Zwischencodeinterpreter durchgeführt, weil der $VAM_{2\&1P}$-Übersetzer die Variablenzugriffe so umordnet, daß keine Konflikte zwischen Lese- (`nxtvar`-Befehle) und Schreiboperationen (`fstvar`-Befehle) entstehen. Dieser Vorgang wird mit einer temporären *Variablen* als Zwischenspeicher beim Variablenkopieren (umordnen) durchgeführt. Bei dieser Optimierung wird kein neues *stack frame* am *environment stack* angelegt, sondern es wird das *stack frame* der aufrufenden *Klausel* verwendet (siehe Kapitel 4.1.1).

Beispiel:
`a(B,A) :- ...,...,a(A,B).`

ohne Optimierung	mit Optimierung
`<fstvar B>_<nxtvar A>`	`<fsttmp X>_<nxtvar B>`
`<fstvar A>_<nxtvar B>`	`<fstvar B>_<nxtvar A>`
	`<fstvar A>_<nxttmp X>`
`Falsch!`	**`Richtig!`**

5.4.2 Optimierte Listen-/Strukturverarbeitung (Selbstrekursion):

Falls eine Liste oder Struktur in der selbstrekursiven *Klausel* vorkommt, werden vom VAM_{1P}-Übersetzer für die *Argumente* der Liste/Struktur die in Kapitel 5.2 beschriebenen `unify`-Befehle generiert. Auf unterster Ebene wird dabei das Ende einer Liste/Struktur nicht besonders markiert, es wird einfach dadurch erkannt, daß keine `unify_`Befehle mehr folgen. Enthält nun eine Liste/Struktur weitere Substrukturen (wieder Listen/Strukturen), so wird ein `unify_list`/`unify_struct`-Befehl erzeugt, der bei Ausführung den $VAM_{2\&1P}$-Interpreter veranlaßt, den Strukturzeiger für das nächste *Argument* auf den Stack zu schreiben. Die *Argumente* der Unterstruktur werden wieder als `unify`-Befehle generiert und mit einem `pop`-Befehl abgeschlossen. Als Optimierung gibt es den `unify_last_list`/`unify_last_struct`-Befehl, der verwendet wird, wenn das letzte Element einer Liste/Struktur wieder eine Liste oder Struktur ist. Dann wird zur Laufzeit nichts mehr auf den Stack "gepusht" und es wird am Ende auch kein pop mehr aufgerufen. Das letzte *Argument* einer Liste/Struktur muß wegen der Optimierung durch die Befehle `unify_last_list` und `unify_last_struct` nicht auf den Stack "gepusht" werden. Die Rücksprungadresse wird implizit kodiert. Eine positive Adresse bedeutet read- und eine negative Adresse write-Modus. Beim VAM_{2P}-Code wird für die Behandlung des Stacks kein pop-Befehl generiert, sondern der Zwischencodeinterpreter benutzt intern einen Zähler, der die Anzahl der noch zu bearbeitenden Listen- oder Strukturelemente angibt.

Beispiel: *Unifikation* einer lokalen *Variablen* mit der Liste [1,2].

	VAM_{1P}-Befehle:	Aktionen des $VAM_{2\&1P}$-Interpreters
	`<list_fstvar> Offset`	Speicherzelle im *stack frame* (Position `Offset`) für lokale *Variable* erzeugen und einen Verweis auf die zu erzeugende Liste im *copy stack* generieren.
	`<unify_const 1>`	erstes Listenelement am *copy stack* anlegen
	`<unify_last_list>`	Restliste erzeugen, Listenspeicherzelle generieren (Zeiger auf die Liste)
	`<unify_const 2>`	zweites Listenelement am *copy stack* anlegen
	`<unify_last_list>`	Restliste erzeugen, Listenspeicherzelle generieren (Zeiger auf die Liste)
	`<unify-nil>`	leere Liste am *copy stack* erzeugen

Wird eine Listen-/Struktur-*Unifikation* mit einem `list_nxtvar`-Befehl eingeleitet, dann wird die Liste-/Struktur am *copy stack* nicht angelegt, sondern bei den nachfolgenden *Unifikationsbefehlen* von dort ausgelesen und mit den einzelnen *Argumenten* der in der lokalen *Variablen* gespeicherten Liste/Struktur verglichen.

5.5 Vor- und Nachteile:

Ein Vorteil der $VAM_{2\&1P}$ ist der kompakte Zwischencode der VAM_{2P}. Zusätzlich wurde die Ausführungsgeschwindigkeit der Prologprogramme gegenüber der VAM_{2P} durch Optimierungen, die teilweise an die VAM_{1P} angelehnt sind etwas erhöht. Die Prologprogramme werden bei der $VAM_{2\&1P}$ nicht so schnell ausgeführt wie bei der VAM_{1P}, aber der $VAM_{2\&1P}$-Zwischencode kann auf die verschiedensten realen Maschinen (Prozessoren) ohne Überarbeitung übertragen werden.

6 Vergleiche bezüglich des Speicherverbrauchs und des Laufzeitverhaltens

Bei Vergleichen bezüglich des Speicherverbrauchs muß der Speicherplatz für *stack frames*, *choice points*, der Platz am *copy stack*, am *trail* und für den Zwischencode berücksichtigt werden. Beim *copy stack* und beim *trail* ergeben sich bezüglich der VAM_{2P}, VAM_{1P} und $VAM_{2\&1P}$ keine wesentlichen Unterschiede solange kein *garbage collector* eingesetzt wird.

Die unten angeführten Testprogramme wurden auf einem IBM kompatiblen PC mit Betriebssystem Windows 95, mit einem Intel Pentium 100 MHz Prozessor und 32 MB Hauptspeicher durchgeführt. Es wurden dabei verschiedene Benchmarks verwendet:

deterministisches append: In einer Schleife mit 9000 Durchläufen werden 2 Listen von 30 Elementen zusammengehängt (mittels append/3). Das dritte *Argument* von append/3 ist beim Aufruf ungebunden.

naive reverse: Eine Liste mit 30 Elementen wird umgedreht. Unter Verwendung von naive reverse (Abspalten des ersten Elements und Anhängen an den umgedrehten Rest der Liste) sind dazu 496 logische Inferenzen notwendig.

quicksort: Diese Implementierung des Quicksort-Algorithmus in Prolog besteht aus zwei Prädikaten. Diese sortieren eine Liste von 50 Elementen.

8-queens: Implementierung des bekannten 8-Queens Problem; 8 Königinnen-Schachfiguren müssen auf einem 8x8 Felder großen Schachbrett plaziert werden, sodaß sie sich nicht gegenseitig bedrohen.

differentiate: Dieser Benchmark beinhaltet ein Prädikat mit 10 alternativen selbstrekursiven *Klauseln*.

query: Dieses Programm durchsucht eine geographische Datenbank nach Ländern mit ungefähr gleich hoher Populationsdichte. Es besteht aus zwei komplexen Prädikaten und einer Datenbank von 2x25 Fakten der Arität 2 mit unterschiedlichen Atomen im ersten *Argument*.

bucket: Dieses Prologprogramm beschreibt einen rekursiven Suchalgorithmus in Strukturen. In 2 von 5 Prädikaten treten Selbstrekursionen auf.

Die folgenden Tabellen zeigen die Unterschiede in der Zwischencodegröße (bzw. Maschinencodegröße bei der VAM_{1P}) und der Ausführungszeit der Prologprogramme gewichtet zur VAM_{2P} an.

Testprogramm	VAM_{2P} (Bytes)	VAM_{2P} (gewichtet)	**$VAM_{2\&1P}$ (gewichtet)**	VAM_{1P} (gewichtet)
det. append	592	1	**1.11**	9.96
naive reverse	632	1	**1.10**	11.30
quicksort	920	1	**1.14**	9.95
8-queens	752	1	**1.17**	8.25
differentiate	1412	1	**1.32**	28.40
query	2754	1	**1**	3.13
bucket	1372	1	**1.07**	9.75

Tabelle 21: Zwischencodegröße($VAM_{2\&1P}$) / Maschinencodegröße (VAM_{1P}) relativ zur VAM_{2P}

Testprogramm	VAM_{2P} (ms)	VAM_{2P} (gewichtet)	**$VAM_{2\&1P}$ (gewichtet)**	VAM_{1P} (gewichtet)	SICStus (gewichtet)
det. append	0.25	1	**1.31**	26.10	1.10
naive reverse	4.17	1	**1.26**	20.00	1.06
quicksort	6.00	1	**1.04**	18.10	1.10
8-queens	65.40	1	**1.02**	13.50	1.10
differentiate	1.14	1	**1.01**	8.14	0.99
query	41.70	1	**1**	9.70	0.89
bucket	2470	1	**0.95**	5.24	0.88

Tabelle 22: Ausführungszeit, Beschleunigungsfaktor relativ zur VAM_{2P}

Insgesamt kann man sagen, daß die VAM_{1P} die Prologprogramme am schnellsten ausführt.
Das einzige und somit letzte *Unterziel* der det. append-Schleife ist bei jedem Schleifendurchlauf selbstrekursiv und deterministisch, was sich in dem Geschwindigkeitsgewinn der $VAM_{2\&1P}$ gegenüber der VAM_{2P} bemerkbar macht. Beim Programm naive reverse verhält sich die Schleifenrekursion analog zu der beim det. append. Die Geschwindigkeitssteigerung der $VAM_{2\&1P}$ gegenüber der VAM_{2P} ist aber nicht so hoch wie beim det. append, weil das zweite Prädikat (Umdrehen der Liste) immer nur im Zwei-Instruktionszeiger-Modus ausgeführt werden kann.
Bei den restlichen Benchmarks kann der Ein-Instruktionszeiger-Modus des $VAM_{2\&1P}$-Interpreters bei den Schleifenrekursionen zu selten durchlaufen werden (letzter Unterzielaufruf ist nicht selbstrekursiv und deterministisch). Dadurch kann gegenüber der VAM_{2P} keine wesentlich schnellere Ausführung der Benchmarks erzielt werden. Das Testprogramm query "verhält" sich bei der $VAM_{2\&1P}$ wie bei der VAM_{2P}, weil keine selbstrekursiven Schleifen darin enthalten sind.

Die Laufzeit der Prologprogramme auf der $VAM_{2\&1P}$ gegenüber der VAM_{2P} ist zwar nicht sehr viel geringer, aber doch bemerkenswert. Die $VAM_{2\&1P}$ kann sicherlich noch in einigen Punkten verbessert werden, was sich ebenfalls in einer Laufzeitverkürzung der Prologprogramme auswirken könnte.

7 Anhang

7.1 Darstellung der Datenstrukturen bei der VAM:

Folgende Arten von Objekten werden durch ein *tag* im niedrigstwertigen Teil der Speicherzelle unterschieden:

Objekte mit 3 Bits als *tag*

Objekt	Wert (Bits 31-3)	*tag* (Bits 2-0)
Atom (Zeichenkette)	Nummer des Atoms	100
nil (leere Liste)	1..1	100
Integer	ganze Zahl (Zweierkomplementdarstellung)	000

Auf Atome wird mit einer Hashtabelle (der Index ist die Nummer des Atoms) auf einen speziellen Speicherbereich zugegriffen.

Objekte mit 2 Bits als *tag*

Objekt	Wert (Bits 31-2)	*tag* (Bits 1-0)
Referenz	Zeiger auf die referenzierte *Variable*	01
Liste	Zeiger auf das erste Element der Liste	10
Struktur	Zeiger auf den Beginn des Strukturblocks (*Funktor*)	11

7.2 VAM2P-Befehlssatz

Die Befehls-Kombinationen (*Unifikationsbefehle* werden mit *Unifikationsbefehlen* und Kontrollbefehle werden mit Abschlußbefehlen kombiniert) werden im folgenden Format angegeben:
<Befehl des Aufgerufenen >-<Befehl des Aufrufers > Operand(en)
h_... - g_...

Die unten angeführten Codekombinationen sind symmetrisch, "head"-Code (z.B.: const) und "goal"-Code (z.B.: fstvar) können bei gleichbleibender Aktion vertauscht werden. Das heißt zum Beispiel, die VAM2P-Befehle `g_const-h_fstvar` und `g_fstvar-h_const` (siehe auch Kapitel 3) werden hier nur einmal als "const_fstvar"-Befehl angeführt.
Befehle, die ein "FALSE"-Resultat bewirken und einen Abbruch des *Unifikations*- und *Resolutionsalgorithmuses* auslösen, werden hier nicht angeführt.

Der bei den Operanden vorkommende Ausdruck `varoffset` bezeichnet immer den Wert des Variablenoffsets relativ zum Beginn des aktuellen Variablenblocks im *stack frame*.

Für *Klauseln* mit bis zu vier lokalen *Variablen* (häufiger Fall) wird ein optimierter Code erzeugt. Der Zwischencodeinterpreter VAM2&1P greift direkt auf die Speicherstellen der ersten vier lokalen *Variablen* mittels des jeweiligen Variablenoffsets (Beginn des Variablenblocks im *stack frame* + `Offset`) zu (`x`fstvar- und `x`nxtvar-Befehle). Die `varoffset`-Information wird in diesem Fall nicht ausgelesen, sondern ist bereits durch die Werte 0-3 vorbestimmt.

Unifikations-Befehle:

const_const const1, const2

Der Interpreter unifiziert die zwei Konstanten `const1` und `const2`.

const_void const

Der Operand `const` wird überlesen.

const_`x`fstvar const, varoffset
fsttemp`<0>`_const tempoffset, const

Die VAM führt die *Unifikation* einer Konstanten `const` mit dem ersten Auftreten der lokalen/temporären *Variablen* an der Stelle `varoffset/tempoffset` durch. Die lokalen/temporären *Variable* erhält den Wert der Konstanten.

const_`x`nxtvar const, varoffset
nxttemp`<0>`_const tempoffset, const

Dieser Befehl führt die *Unifikation* einer Konstanten `const` mit einer bereits aufgetretenen lokalen/temporären *Variablen* durch.

nil_nil

Diese *Unifikation* liefert ein "TRUE"-Resultat.

nil_void

Diese *Unifikation* liefert ein "TRUE"-Resultat.

nil_xfstvar varoffset
fsttemp<0>_nil tempoffset

Hier wird eine *Unifikation* von einem ersten Auftreten einer lokalen/temporären *Variablen* mit einer leeren Liste durchgeführt.

nil_xnxtvar varoffset
nxttemp<0>_nil tempoffset

Hier wird eine *Unifikation* von einem weiteren Auftreten einer lokalen/temporären *Variablen* mit einer leeren Liste durchgeführt.

list_list listhead1, listtail1, listhead2, listtail2

Der Interpreter versucht zwei Listen mit den Elementen `listhead1`, `listtail1` und `listhead2`, `listtail2` zu unifizieren. Er liest diese Listen vom *copy stack* beziehungsweise legt sie dort an.

list_void listhead, listtail

Die beiden Operanden `listhead` und `listtail` werden überlesen.

list_xfstvar listhead, listtail, varoffset
fsttemp<0>_list tempoffset, listhead, listtail

Die VAM unifiziert das erste Auftreten einer lokalen/temporären *Variablen* mit einer Liste (listhead, listtail). Der Interpreter fordert dabei für die Datenstruktur der Liste entsprechend Speicherplatz am *copy stack* an.

list_`x`nxtvar listhead, listtail, varoffset
nxttemp`<0>`_list tempoffset, listhead, listtail

Diese Befehle versuchen, eine Liste mit einer bereits vorgekommenen lokalen/temporären *Variablen* in einer *Klausel* zu unifizieren. Man überprüft, ob diese *Variable* bereits gebunden ist oder nicht. Ist sie ungebunden, so wird eine neue Liste am *copy stack* angelegt und mit den Listenkopier-Befehl fortgesetzt. Ist die *Variable* bereits an eine Liste gebunden, dann wird die Liste `listhead` vom *copy stack* gelesen und mit der *Variablen* unifiziert.

struct_struct Functor1, n1 Elements, Functor2, n2 Elements

Der Interpreter versucht zwei Strukturen mit den *Funktoren* `Functor1` und `Functor2` mit jeweils `n1` und `n2` Elementen zu unifizieren. Er liest diese Strukturen vom *copy stack* beziehungsweise legt sie dort an.

struct_void Functor, n Elements
Die Struktur wird mit dem *Funktor* `Functor` samt ihren `n` Elementen überlesen.

struct_`x`fstvar Functor, n Elements, varoffset
fsttemp`<0>`_struct tempoffset, Functor, n Elements

Das erste Auftreten einer lokalen/temporären *Variablen* wird mit einer Struktur `Functor` mit `n`-Elementen unifiziert. Die VAM fordert den Speicherplatz für diese *Strukturunifikation* am *copy stack* an.

struct_`x`nxtvar Functor, n Elements varoffset
nxttemp`<0>`_struct tempoffset, Functor, n Elements

Diese Befehle versuchen, eine Struktur mit dem *Funktor* `Functor` mit einer bereits vorgekommenen lokalen/temporären *Variablen* zu unifizieren. Der Interpreter überprüft, ob diese *Variable* bereits gebunden ist oder nicht. Ist sie ungebunden, dann wird eine neue Struktur, die der angegebenen entspricht am *copy stack* angelegt. Ist die *Variable* bereits an eine Struktur gebunden, dann wird die Struktur `Functor` vom *copy stack* ausgelesen und mit der *Variablen* unifiziert.

void_void

Diese *Unifikation* liefert ein "TRUE"-Resultat.

void_x**fstvar varoffset**
fsttemp<0>**_void tempoffset**

Der Interpreter legt eine ungebundene Speicherzelle am *copy stack* für die lokale/temporäre *Variable* an. Tritt im selben *Unterziel/Kopf* diese *Variable* ein weiteres Mal auf, so überliest der Interpreter die lokale/temporäre *Variable*.

void_x**nxtvar varoffset**
nxttemp<0>**_void tempoffset**

Der Interpreter überliest die bereits aufgetretene lokale *Variable*.

fstvar_x**fstvar varoffset1, varoffset2**
fsttemp<0>**_**x**fstvar tempoffset, varoffset**

Dieser Befehl kombiniert die ersten Auftreten zweier lokaler (beziehungsweise einer temporären und einer lokalen) *Variablen*. Dabei wird eine freie Speicherzelle am *copy stack* angelegt. Diese Speicheradresse wird den *Variablen* zugewiesen.

fstvar_x**nxtvar varoffset1, varoffset2**
fsttemp_x**nxtvar tempoffset, varoffset**

Der Interpreter unifiziert das erste Auftreten einer lokalen/temporären *Variablen* mit dem weiteren Auftreten einer lokalen *Variablen*. Dabei wird der Inhalt der *Variablen* an der Stelle `varoffset2/varoffset` der *Variablen* an der Stelle `varoffset1/tempoffset` zugewiesen.

nxtvar_x**nxtvar varoffset1, varoffset2**
nxttemp<0>**_**x**nxtvar tempoffset1, varoffset**

Die VAM liest die Variableneinträge vom *stack frame*/temporären Speicherbereich und unifiziert sie.

Kontroll- und Abschluß-Befehle:

Die Befehlskombinationen haben folgendes Format:

<Befehl des Aufgerufenen>_<Befehl des Aufrufers>

xgoal_call

Es wird ein *stack frame* für die aufgerufene *Klausel* und der aufrufenden *Klausel* generiert. Die aktuellen Verzweigungszeiger werden für die *continuation* generiert und gespeichert. Die *Unifikation* wird fortgesetzt.

xgoal_lastcall

Der Interpreter führt die Lastcall_Optimierung durch.
Für die *Klauseln* mit 1-4 *Argumenten* existieren wieder spezielle optimierte Behandlungsroutinen.

nogoal_call
nogoal_lastcall

Der Interpreter liest die *continuation.* Er setzt die Bearbeitung beim Aufrufer (*stack frame* der Aufrufenden *Klausel*) fort.

cut_call
cut_lastcall

Falls ein *choice point* für die aktuelle *Klausel* existiert, wird er gelöscht. Die Größe des aktuellen *stack frames* des Aufrufers wird auf jene *Variablen* beschränkt, die noch für die restlichen *Unterziele* benötigt werden.
Wird ein cut ausgeführt, so werden beim *Backtracking* Alternativen in den *Unterzielen* oder im *Kopf* der *Klausel* vor dem cut nicht berücksichtigt.

builtin_call
builtin_lastcall

Die VAM führt ein builtin-*Prädikat* aus.

7.3 Befehlssatz der VAM_{1P}

Der Befehlssatz wird aufgeteilt in *Unifikations*-, *Strukturunifikations*-, Strukturerzeugungs-, Kontroll-, *Klauselindizierungs*- und Optimierungs-Befehle.

Unifikations-Befehle

Die *Unifikations*-Befehle behandeln die *Unifikation* eines *Arguments* vom aufrufenden *Unterziel* mit dem zugehörigen *Argument* von der aufgerufenen *Klausel*. Wird eine Struktur oder Liste mit dem ersten Auftreten einer lokalen *Variablen* kombiniert (fstvar_struct/fstvar_list), so wird die Struktur- oder Listenerzeugung gestartet. Der Operand `Size` gibt die Größe der Struktur (bzw. Liste) samt allen Unterstrukturen an. Der Operand `Reg` bei dem VAM_{1P}-Befehl nxtvar_struct/nxtvar_list ist ein Register, das einen Zeiger auf die Struktur/Liste enthält.

fstvar_const Var, Const

Hier wird die *Unifikation* einer Konstanten `Const` mit einer lokalen *Variablen* `Var` durchgeführt.

nxtvar_const Var, Const

Dieser Befehl führt die *Unifikation* zwischen einer Konstanten `Const` und einer bereits aufgetretenen lokalen *Variablen* `Var` durch.

fstvar_nil Var

Die lokale *Variable* `Var` wird mit `nil` initialisiert.

nxtvar_nil Var

Eine lokale *Variable* `Var`, die bereits vorher in einer *Klausel* aufgetreten ist, wird mit `nil` unifiziert.

fstvar_list Var, Size

Die lokale *Variable* `Var` wird mit einer Liste initialisiert. Die Datenstruktur einer Liste (siehe Kapitel 2) wird in einer Variablenzelle gespeichert und der `copyptr`-Zeiger wird um `Size` Elemente verringert.

nxtvar_list Var, Reg, Size, CAddr

Eine Liste wird mit einer bereits vorgekommenen lokalen *Variablen* in einer *Klausel* unifiziert. Es wird überprüft, ob diese *Variable* bereits gebunden ist oder nicht. Ist sie ungebunden, so wird eine neue Liste am *copy stack* mit der Größe von `Size` Elementen angelegt und mit den Listenkopier-Befehlen an der Adresse `CAddr` fortgesetzt.
Ist sie bereits an eine Liste gebunden, dann wird die Startadresse von der Liste im Register `Reg` gespeichert und der nächste VAM_{1P}-Befehl ausgeführt.

fstvar_struct Functor, Var, Size

Die VAM unifiziert das erste Auftreten einer lokalen *Variablen* mit einer Struktur. Die Datenstruktur einer Struktur (siehe Kapitel 2) speichert man in einer Variablenzelle und der `copyptr`-Zeiger wird um `Size` Elemente verringert.

nxtvar_struct Functor, Var, Reg, Size, CAddr

Die VAM_{1P} versucht, eine Struktur mit dem *Funktor* `Functor` mit einer bereits vorgekommenen lokalen *Variablen* in einer *Klausel* zu unifizieren. Es wird überprüft, ob diese *Variable* bereits gebunden ist oder nicht. Ist sie ungebunden, so wird eine neue Struktur am *copy stack* mit der Größe von `Size` Elementen (einschließlich dem *Funktor*) angelegt und mit den Strukturkopier-Befehlen an der Adresse `CAddr` fortgesetzt.
Ist sie bereits an eine Struktur gebunden, dann wird die Startadresse von der Struktur im Register `Reg` gespeichert und der nächste VAM_{1P}-Befehl ausgeführt.

fstvar_fstvar Var1, Var2

Dieser Befehl kombiniert die ersten Auftreten zweier lokaler *Variablen*. Eine freie Speicherzelle wird am *copy stack* angelegt. Diese Speicheradresse wird den *Variablen* zugewiesen.

nxtvar_fstvar Var1, Var2

Das erste Auftreten der lokalen *Variablen* `Var2` wird mit dem weiteren Auftreten der lokalen *Variablen* `Var1` unifiziert. Dabei wird der Inhalt der *Variablen* `Var1` `Var2` zugewiesen.

nxtvar_nxtvar Var1, Var2

Zwei bereits einmal aufgetretenen *Variablen* werden unifiziert.

fstvar_void Var

Dieser Befehl legt eine ungebundene Speicherzelle am *copy stack* für die lokale *Variable* an. Ungebundene Speicherzellen werden durch Selbstreferenzen dargestellt. Tritt im selben *Unterziel/Kopf* einer *Klausel* diese lokale *Variable* ein weiteres Mal auf wird kein Code erzeugt. Dieser fstvar_void-Befehl wird auch dann generiert, falls die lokale *Variable* mit einem "unbekannten" Strukturelement übereinstimmt.

Strukturunifikations-Befehle

Diese Befehle werden verwendet, um Elemente einer neuen Struktur mit den Elementen einer bereits existierenden Struktur zu unifizieren. Jeder Befehl beinhaltet einen Operanden namens `Offset`, welches die Adresse des zugehörigen Strukturelements im *copy stack* angibt. (`copyptr` + `Offset`).

Der Operand `CAddr` gibt das Sprungziel zu den Strukturerzeugungs-Befehlen an.

unify_const Const, Offset

Die Konstante `Const` wird mit dem Strukturelement am *copy stack* an der Adresse `Offset` unifiziert.

unify_nil Offset

Eine leere Liste wird mit dem Strukturelement am *copy stack* an der Adresse `Offset` unifiziert.

unify_list Reg, Size, Offset, CAddr

Dieser Befehl unifiziert eine Liste mit einem Strukturelement am *copy stack* (Adresse `Offset`). Ist dieses Strukturelement noch ungebunden, dann wird eine neue Liste generiert, und ihre Adresse wird dem Operanden `Offset` zugewiesen. Die neue Liste besitzt `Size` Elemente. Die anschließenden Kopier-Befehle für die Liste befinden sich ab Adresse `CAddr`. Ist der Operand `Offset` bereits zu einer Liste gebunden, dann wird der Zeiger auf die Liste ins Register `Reg` geschrieben.

unify_struct Functor, Reg, Size, Offset, CAddr

Eine Struktur mit dem *Funktor* `Functor` wird mit dem Strukturelement am *copy stack* an der Adresse `Offset` unifiziert. Ist dieses Element noch ungebunden, dann wird eine neue Struktur am *copy stack* angelegt, und die ihre Adresse wird dem Operanden `Offset` zugewiesen. Die neue Struktur besitzt `Size` Elemente einschließlich der *Funktor*-Speicherzelle. Die nachfolgenden Strukturkopierbefehle beginnen an der Adresse `CAddr`. Ist `Offset` bereits an eine Struktur (mit *Funktor*) gebunden, dann wird der Zeiger auf die Struktur in das Register `Reg` geschrieben.

unify_fstvar Var, Offset

Die VAM_{1P} unifiziert das erste Auftreten einer lokalen *Variablen* mit der Speicherzelle an der Adresse `Offset` im *copy stack*. Der lokalen *Variablen* `Var` wird der Inhalt dieser Speicherzelle zugewiesen.

unify_nxtvar Var, Offset

Eine bereits mehrmals aufgetretene lokale *Variable* wird mit der Speicherzelle an der Adresse `Offset` im *copy stack* unifiziert.

Strukturerzeugungs-Befehle

Diese Befehle werden benutzt, um Strukturelemente beziehungsweise Listen auf den *copy stack* zu kopieren. Jeder Befehl besitzt einen Operanden `Offset`, der die Speicherstelle relativ zum aktuellen `copyptr` angibt, in der die Liste beziehungsweise Struktur gespeichert wird.

create_const Const, Offset

Die Konstante `Const` wird an der Speicherstelle `copyptr` + `Offset` im *copy stack* abgespeichert.

create_nil Offset

Eine leere Liste wird an der Speicherstelle `copyptr` + `Offset` im *copy stack* abgespeichert.

create_list Offset1, Offset2

Eine neue Liste wird an der Stelle `copyptr` + `Offset1` im *copy stack* abgespeichert. Die Liste beginnt an der Stelle `copyptr` + `Offset2`.

create_struct Functor, Offset1, Offset2

Eine neue Struktur wird an der Stelle `copyptr` + `Offset1` im *copy stack* abgespeichert. Die Struktur beginnt an der Stelle `copyptr` + `Offset2`, wobei das erste Element am *copy stack* der *Funktor* `Functor` dieser Struktur wird.

create_fstvar Var, Offset

Dieser Befehl erzeugt eine freie Speicherzelle an der Adresse `copyptr` + `Offset` im *copy stack*. Die lokale *Variable* `Var` verweist dann auf diese Speicherzelle.

create_nxtvar Var, Offset

Der Inhalt der lokalen *Variablen* `Var` wird in der Speicherstelle an der Adresse `copyptr` + `Offset` gespeichert.

Kontroll-Befehle

Diese Befehle beinhalten die Stack-Bearbeitung, steuern den Programmfluß und die notwendigen Änderungen im *stack frame*.

alloc_stackframe VarCount

alloc_stackframe generiert ein neues *stack frame* am *environment stack*. Die Anzahl der lokalen *Variablen*, die auf diesem *stack frame* gespeichert werden können ist `VarCount`.

adjust VarCount

adjust führt das Stacktrimming (siehe Kapitel 4.1.2) durch.
Die Größe des aktuellen *stack frames* wird soweit vermindert, daß nur noch `VarCount` lokale *Variablen* in diesem *stack frame* gespeichert werden können.

store_cont_goto ContAddr, ProcAddr

store_cont_goto speichert die *continuation* im *stack frame* vor einem Prozeduraufruf und springt zu der Prozedur `ProcAddr`.

read_cont_goto

Nach dem erfolgreichen Aufruf einer Prozedur wird mit read_cont_goto die *continuation* vom *stack frame* gelesen und das nächste *Unterziel* ausgeführt (Sprung zur *continuation*-Adresse).

copy_cont_goto ProcAddr

Beim letzten *Unterziel* wird anstatt des Befehls store_cont_goto der Befehl copy_cont_goto generiert.
copy_cont_goto kopiert die *continuation* vom *stack frame* des Aufrufers und springt zu der Prozedur `ProcAddr` der aufgerufenen *Klausel*.

goto Addr

goto Addr springt zum nächsten Befehl, der an der Adresse `Addr` steht.

store_var Reg, Offset

store_var speichert die *Variablen* des *Kopfes* einer *Klausel*, die sich in den Registern befinden, im *stack frame* ab, sofern sie für den *Kopf* der nächsten *Klausel* benötigt werden.

copy_var Offset

Können nicht alle *Variablen* des *Kopfes* einer *Klausel* in Registern gehalten werden, dann werden die *Variablen* des *stack frames* der aufrufenden *Klausel* mit dem Befehl copy_var in das *stack frame* der aufgerufenen *Klausel* kopiert während der Lastcall-Optimierung.

push_choice_point AltAddr

push_choice_point erzeugt einen *choice point* mit dem Zeiger `AltAddr`, der auf den Code der noch übrigbleibenden alternativen *Klauseln* zeigt.

cut VarCount

cut entfernt den *choice point* und führt "Stacktrimming" aus.

call_bip BipId

Dieser Befehl ruft ein builtin-*Prädikat* mit dem Index `BipId` auf.

inline_bip BipId

Dieser Befehl fügt den Code eines "inline-builtin"-*Prädikats* in die Datenbasis ein.

Indizierungs-Befehle

Diese Befehle werden für die *Klauselindizierung* benötigt.

Tritt eine lokale *Variable* als *Argument* im aufrufenden *Unterziel* auf und ist dieselbe *Variable* schon vorher aufgetreten (keine `fstvar`), so wird der Inhalt dieser *Variablen* als Kriterium zur Auswahl der aufgerufenen *Klausel* herangezogen. Dadurch ist ein gezielter Aufruf passender *Klauseln* möglich. Eigene Makros dienen dem Dereferenzieren und der Bestimmung des Typs dieser *Variablen*, sowie dem Einsprung in die den entsprechenden Typ behandelnden Programmzweige. Außerdem werden für jeden Typ eigene *Unifikations*-Makros, die die dereferenzierte *Variable* mit dem *Kopfargument* unifizieren, benötigt.

deref Var

Die *Variable* `Var` wird dereferenziert und ihr Inhalt in einem eigens für die *Klauselindizierung* vorgesehenem Register abgelegt.

if_undef label_of_firstclause

Falls der Inhalt des *Klauselindexregisters* undefiniert ist, wird zur angegebenen Adresse (label_of_firstclause) gesprungen.

if_const label_of_firstclause

Falls der Inhalt des *Klauselindexregisters* eine Konstante ist, wird zur angegebenen Adresse (`label_of_firstclause`) gesprungen.

if_nil label_of_firstclause

Falls der Inhalt des *Klauselindexregisters* eine leere Liste (nil) ist, wird zur angegebenen Adresse (`label_of_firstclause`) gesprungen.

if_list label_of_firstclause

Falls der Inhalt des *Klauselindexregisters* eine Listenreferenz ist, wird zur angegebenen Adresse (`label_of_firstclause`) gesprungen.

if_struct label_of_firstclause

Falls der Inhalt des *Klauselindexregisters* eine Strukturreferenz ist, wird zur angegebenen Adresse (`label_of_firstclause`) gesprungen.

comp_const const, address_of_<_funct, address_of_>_funct

Falls der Inhalt des *Klauselindexregisters* kleiner als die Konstante const ist wird zur angegebenen ersten Adresse gesprungen, falls er größer ist, zur zweiten. Bei Gleichheit werden die diesem Makro folgenden Befehle ausgeführt.

comp_struct Functor, address_of_<_funct, address_of_>_funct

Falls der *Funktor* `Functor` der durch das *Klauselindexregister* referenzierten Struktur kleiner als der im ersten Operand angegebene Strukturname und die dazugehörige Arität sind, wird zur ersten Adresse gesprungen, falls er größer ist, zur zweiten. Bei Gleichheit werden die diesem Makro folgenden Befehle ausgeführt.

const_undef const

Dem *Klauselindexregister* wird die Konstante const zugewiesen.

nil_undef

Dem *Klauselindexregister* wird die leere Liste zugewiesen.

list_undef Size

Dem *Klauselindexregister* wird eine Listenreferenz auf die Spitze des *copy stacks* zugewiesen. Diesem Makro folgen die Kopierbefehle, die die *Argumente* der Liste am *copy stack* anlegen.

struct_undef Size

Dem *Klauselindexregister* wird eine Strukturreferenz auf die Spitze des *copy stacks* zugewiesen. Diesem Makro folgen die Kopierbefehle, die den *Funktor*, sowie die *Argumente* der Struktur am *copy stack* anlegen.

list_val Reg, Var

Für die diesem Makro folgenden *Listenunifikationsbefehle* wird die Anfangsadresse der durch das *Klauselindexregister* referenzierten Liste dem angegebenen Register zugewiesen.

struct_val Reg, Var

Für die diesem Makro folgenden *Strukturunifikationsbefehle* wird die Anfangsadresse der durch das *Klauselindexregister* referenzierten Struktur dem angegebenen Register zugewiesen.

Optimierungs-Befehle

<u>Diese Befehle werden für Schleifenoptimierungen verwendet (bei Listen und Strukturen):</u>

extract_list Reg, LAddr, ContAddr

construct_list Offset

construct_extract_list Reg, LAddr, UAddr

extract_struct Functor, Reg, Size, SAddr, ContAddr

construct_struct Functor, Size, Offset

construct_extract_struct Functor, Reg, Size, SAddr, UAddr

<u>Diese Befehle werden für die Entfernung von temporären *Variablen* in Strukturen verwendet:</u>

create_undef Offset

unify_create Offset, Offset

unify_unify Offset1, Offset2

create_ref Offset1, Offset2

create_undef Offset

create_undef initialisiert eine Speicherzelle im *copy stack* als ungebunden. Die Adresse dieser *Variablen* ergibt sich aus `copyptr` + `Offset`.

unify_create Offset1, Offset2

unify_create kopiert einen Wert von einer Struktur (`Offset1`) zu einer neuen (`copyptr` + `Offset2`).

unify_unify Offset1, Offset2

unify_unify implementiert den gesamten *Unifikations*-Prozeß zwischen zwei *Argumenten* (`Offset1`, `Offset2`) von Strukturen.

create_ref Offset1, Offset2

create_ref erzeugt eine Referenz von einem *Argument* einer Struktur zu einem anderen einer neuen Struktur.

extract_list Reg, LAddr, ContAddr

Wenn die Adresse `LAddr` nicht zu einer Liste gebunden ist, wird das Programm bei `ContAddr` fortgesetzt. Andererseits wird die Speicheradresse der Liste `LAddr` ins Register `Reg` geschrieben.

construct_list Offset

Eine neue Liste wird erzeugt (`copyptr` + `Offset`).

construct_extract_list Reg, LAddr, UAddr

Ist `LAddr` zu einer Liste gebunden, dann wird diese Adresse ins Register `Reg` geschrieben. Beinhaltet `LAddr` eine ungebundene Speicherzelle, wird eine neue Liste mit zwei Elementen am *copy stack* erzeugt und an der Stelle `UAddr` fortgefahren. In allen andern Fällen liefert die Ausführung dieses Befehls ein "FALSE"-Resultat zurück.

extract_struct Functor, Reg, Size, SAddr, ContAddr

Wenn die Speicherzelle bei der Adresse `SAddr` eine Struktur beinhaltet, dann speichert extract_struct den Beginn der Struktur im Register `Reg`. Andererseits wird die Ausführung des Prologprogrammes bei der Adresse `ContAddr` fortgesetzt.

construct_struct Functor, Size, Offset

construct_struct erzeugt eine Struktur.

construct_extract_struct Functor, Reg, Size, SAddr, UAddr

Ist die Kombination der Befehle `construct_struct` und `extract_struct`.

7.4 $VAM_{2\&1P}$-Befehlssatz (für selbstrekursive *Klauseln*)

Die Befehls-Kombinationen werden im folgenden Format angegeben:
<Befehl des Aufgerufenen/Aufrufers>_<Befehl des Aufrufers/Aufgerufenen> Operand(en)

Die $VAM_{2\&1P}$ generiert beim Lastcall eines deterministischen selbstrekursiven *Unterziels* die unten angegebenen Befehle. Die VAM_{1P}-Code-Befehle beginnen mit einem `v1_goal`-Befehl und enden mit einem `v1_lastcall`-Befehl. Der $VAM_{2\&1P}$-Interpreter führt diese `v1`-Befehle, wie in Kapitel 5 beschrieben, aus. Der bei den Operanden vorkommende Ausdruck `varoffset` bezeichnet immer den Wert des Variablenoffsets relativ zum Beginn des aktuellen Variablenblocks im *stack frame*. Der bei den Operanden vorkommende Ausdruck `tmpoffset` bezeichnet immer die Speicherstelle der temporären *Variablen* in den dafür vorgesehenen Speicherbereich.

const_fstvar **const, varoffset**

Dieser Befehl initialisiert die *Variable* `varoffset` mit der Konstanten `const`.

const_nxtvar **const, varoffset**

Dieser Befehl unifiziert die *Variable* `varoffset` mit der Konstanten `const`.

const_fsttmp **tmpoffset, const**

Der $VAM_{2\&1P}$-Interpreter initialisiert die temporäre *Variable* `tmpoffset` mit der Konstanten `const`.

const_nxttmp **tmpoffset, const**

Der $VAM_{2\&1P}$-Interpreter unifiziert die temporäre *Variable* `tmpoffset` mit der Konstanten `const`.

nil_fstvar **varoffset**

Das erste Auftreten einer lokalen *Variablen* `varoffset` wird mit der leeren Liste (`nil`) initialisiert.

nil_nxtvar **varoffset**

Das weitere Auftreten einer lokalen Variablen `varoffset` wird mit der leeren Liste (`nil`) unifiziert.

nil_fsttmp **tmpoffset**

Der Interpreter initialisiert die temporäre *Variable* `tmpoffset` mit der leeren Liste (`nil`).

nil_nxttmp **tmpoffset**

Der $VAM_{2\&1P}$-Interpreter unifiziert das weitere Auftreten der temporären *Variablen* `tmpoffset` mit der leeren Liste (`nil`).

void_fstvar **varoffset**

Dieser Befehl legt eine ungebundene Speicherzelle am *copy stack* für die lokale *Variable* `varoffset` an.

void_fsttmp **tmpoffset**

Dieser Befehl legt eine ungebundene Speicherzelle am *copy stack* für die temporäre *Variable* `tmpoffset` an.

list_fstvar **varoffset**

Der $VAM_{2\&1P}$-Interpreter initialisiert das erste Auftreten einer lokalen *Variablen* `varoffset` mit einer Liste. Die Elemente der Liste werden durch die nachfolgenden `unify`-Befehle initialisiert und am *copy stack* angelegt.

list_nxtvar **varoffset**

Der Interpreter versucht eine bereits aufgetretene lokale *Variable* mit einer Liste zu unifizieren. Er überprüft ob diese *Variable* gebunden ist oder nicht. Ist sie ungebunden, so wird die neue Liste anhand der nachfolgenden `unify`-Befehle am *copy stack* angelegt, sonst unifiziert.

list_fsttmp **tmpoffset**

Der $VAM_{2\&1P}$-Interpreter unifiziert das erste Auftreten einer temporären *Variablen* `tmpoffset` mit einer Liste. Die Elemente der Liste werden durch die nachfolgenden `unify`-Befehle initialisiert und am *copy stack* angelegt.

list_nxttmp **tmpoffset**

Der Interpreter versucht eine bereits aufgetretene temporäre *Variable* mit einer Liste zu unifizieren. Er überprüft ob diese *Variable* gebunden ist oder nicht. Ist sie ungebunden, so wird die neue Liste anhand der nachfolgenden `unify`-Befehle am *copy stack* angelegt, sonst unifiziert.

struct_fstvar **functor, varoffset**

Der $VAM_{2\&1P}$-Interpreter unifiziert das erste Auftreten einer lokalen *Variablen* `varoffset` mit einer Struktur `functor`. Die Elemente der Struktur werden durch die nachfolgenden `unify`-Befehle initialisiert und am *copy stack* angelegt.

struct_nxtvar functor, varoffset

Der Interpreter versucht eine bereits aufgetretene lokale *Variable* mit der Struktur `functor` zu unifizieren. Er überprüft ob diese *Variable* gebunden ist oder nicht. Ist sie ungebunden, so wird die neue Struktur anhand der nachfolgenden `unify`-Befehle am *copy stack* angelegt, sonst unifiziert.

struct_fsttmp functor, tmpoffset

Der $VAM_{2\&1P}$-Interpreter unifiziert das erste Auftreten einer temporären *Variablen* `tmpoffset` mit einer Struktur `functor`. Die Elemente der Struktur werden durch die nachfolgenden `unify`-Befehle initialisiert und am *copy stack* angelegt.

struct_nxttmp functor, tmpoffset

Der Interpreter versucht eine bereits aufgetretene temporäre *Variable* mit der Struktur `functor` zu unifizieren. Er überprüft ob diese *Variable* gebunden ist oder nicht. Ist sie ungebunden, so wird die neue Struktur anhand der nachfolgenden `unify`-Befehle am *copy stack* angelegt, sonst unifiziert.

fstvar_fstvar varoffset1, varoffset2

Dieser Befehl kombiniert die ersten Auftreten zweier lokaler *Variablen*. Der Interpreter legt dabei eine freie Speicherzelle am *copy stack* an. Dieser Speicheradresse weist er den *Variablen* zu.

fstvar_nxtvar varoffset1, varoffset2

Das erste Auftreten der lokalen *Variablen* `varoffset1` wird mit dem weiteren Auftreten der lokalen *Variablen* `varoffset2` unifiziert. Dabei wird der Inhalt der *Variablen* `varoffset2` der lokalen *Variablen* `varoffset1` zugewiesen.

fsttmp_fstvar tmpoffset, varoffset2

Dieser Befehl kombiniert die ersten Auftreten einer temporären *Variablen* mit einer lokalen *Variablen*. Der Interpreter legt dabei eine freie Speicherzelle am *copy stack* an. Diese Speicheradresse weist er den *Variablen* zu.

fsttmp_nxtvar tmpoffset, varoffset

Das erste Auftreten der temporären *Variablen* `tmpoffset` wird mit dem weiteren Auftreten der lokalen *Variablen* `varoffset` unifiziert. Dabei wird der Inhalt der *Variablen* `varoffset` der temporären *Variablen* `tmpoffset` zugewiesen.

nxtvar_nxtvar varoffset1, varoffset2
nxttmp_fstvar/nxtvar tmpoffset, varoffset

Die VAM liest die Variableneinträge vom *stack frame*/temporären Speicherbereich und unifiziert sie.

nxttmp_fstvar **tmpoffset, varoffset**

Das erste Auftreten der lokalen *Variablen* `varoffset` wird mit dem weiteren Auftreten der temporären *Variablen* `tmpoffset` unifiziert. Dabei wird der Inhalt der *Variablen* `tmpoffset` der lokalen *Variablen* `varoffset` zugewiesen.

Kontroll- und Abschluß-Befehle

v1_Lastcall **goalptr**

Liest den nächsten *Unterzielbefehl* aus dem Operanden `goalptr` (goal, builtin, cut), initialisiert den Zeiger auf den nächsten Befehl des *Klausenkopfes* und schaltet in den Zwei-Instruktionszeiger Modus der $VAM_{2\&1P}$.

v1_xgoal **[headvars], procptr, headptr, v1_codeptr, varoffset**

Der optionale Operand `headvars` gibt die Anzahl der zu kopierenden *Variablen* im *stack frame* bei der lastcall-Optimierung an. Der Operand `headptr` wird zur Feststellung der Selbstrekursion benötigt. Mit Hilfe des Operanden `procptr`, der auf den Strukturblock des selbstrekursiven letzen *Unterziels* einer *Klausel* zeigt, wird der *Klauselindizierungsalgorithmus* aufgerufen. Dabei überprüft der $VAM_{2\&1P}$-Interpreter, ob der lastcall-Aufruf der selbstrekursiven *Klausel* deterministisch ist. Ist der Aufruf deterministisch, so schaltet der $VAM_{2\&1P}$-Interpreter in den Ein-Instruktionszeiger Modus und arbeitet die VAM_{1P}-Befehlssequenz an der Speicherstelle `v1_codeptr` im *heap* ab. Der Operand `varoffset` bezeichnet den Variablenoffset des im `xgoal`-Befehl inkludierten `nxtvar`-Kommandos. Nach Abarbeitung der VAM_{1P}-Befehlssequenz (siehe `v1_lastcall`-Befehl oben) schaltet der $VAM_{2\&1P}$-Interpreter vom Ein- in den Zwei-Instruktionszeiger Modus zurück.
Bei der Abarbeitung der *Klauselrekursion* wird kein neues *stack frame* am *environment stack* angelegt, sondern es werden die Variablenzugriffe direkt an demselben *stack frame* durchgeführt. Das ermöglicht die in den Kapiteln 3,4 und 5 beschriebene Lastcall-Optimierung.

Unifikationsbefehle für Listen- und Strukturelemente:

Im Gegensatz zur VAM_{1P} werden hier bei der $VAM_{2\&1P}$ die `Offsets` der zu unifizierenden Listen-/Strukturelemente nicht explizit relativ zum *copy stack* angegeben, sondern werden intern vom $VAM_{2\&1P}$-Interpreter zur Laufzeit berechnet.
Ob die unify-Befehle ein Listen- oder Strukturelement am *copy stack* anlegen (initialisieren) oder unifizieren hängt von der aktuellen Betriebsart ab, in der sich der Interpreter befindet (write- oder read-Modus) (siehe Kapitel 5.3.1).

unify_const **C**

Das aktuelle Listen-/Strukturelement am *copy stack* wird mit der Konstanten `C` unifiziert/initialisiert.

unify_nil

Das aktuelle Listen-/Strukturelement am *copy stack* wird mit der leeren Liste (`nil`) unifiziert/initialisiert.

unify_void

Der $VAM_{2\&1P}$-Interpreter überliest das aktuelle Element am *copy stack*.

unify_fstvar varoffset

Das erste Auftreten einer lokalen *Variablen* wird mit dem aktuellen Listen-/Strukturelement am *copy stack* unifiziert. Der lokalen *Variablen* wird das Listen-/Strukturelement zugewiesen.

unify_nxtvar varoffset

Das weitere Auftreten einer lokalen *Variablen* wird mit dem aktuellen Listen-/Strukturelement am *copy stack* unifiziert.

unify_fsttmp tmpoffset

Das erste Auftreten einer temporären *Variablen* wird mit dem aktuellen Listen-/Strukturelement am *copy stack* unifiziert. Der temporären *Variablen* wird das Listen-/Strukturelement zugewiesen.

unify_nxttmp tmpoffset

Das weitere Auftreten einer temporären *Variablen* wird mit dem aktuellen Listen-/Strukturelement am *copy stack* unifiziert.

unify_list

Eine Liste/Struktur enthält eine Liste. Für die Bearbeitung der Liste wird der Listenzeiger für das nächste Listenelement auf den *environment stack* geschrieben. Die *Argumente* der Unterliste/-Struktur werden wieder als `unify`-Befehle generiert. Am Ende der Listenbearbeitung wird die am *environment stack* angelegte Adresse auf das erste Listenelement der Unterliste entfernt (siehe pop-Befehl).

unify_last_list

Wenn das letzte Element einer Liste/Struktur eine Liste ist, dann wird nichts mehr auf den *Stack* "gepusht" (siehe unify_list), und am Ende ist auch kein `pop`-Befehl mehr notwendig. Existiert am *copy stack* bereits eine Liste so wird sie mit den nachfolgenden `unify`-Befehlen für die *Argumente* der Liste unifiziert, sonst initialisiert (erzeugt). Der Interpreter codiert implizit die Rücksprungadresse. Dabei bedeutet eine positive Adresse, daß die Liste unifiziert (read-Modus) wird. Bei einer negativen Adresse wird die Liste am *copy stack* angelegt (write-Modus).

unify_struct functor

Eine Liste/Struktur enthält eine Struktur `functor`. Für die Bearbeitung der Struktur wird der Strukturzeiger für das nächste Strukturelement auf den *environment stack* geschrieben. Die *Argumente* der Unterstruktur werden wieder als `unify`-Befehle generiert. Am Ende der Strukturbearbeitung wird die am *environment stack* angelegte Adresse auf das erste Strukturelement der Unterstruktur entfernt (siehe pop-Befehl).

unify_last_struct functor

Wenn das letzte Element einer Liste/Struktur eine Struktur ist, dann wird nichts mehr auf den *Stack* "gepusht" (siehe unify_struct), und am Ende ist auch kein `pop`-Befehl mehr notwendig. Existiert am *copy stack* bereits eine Struktur `functor`, so wird sie mit den nachfolgenden `unify`-Befehlen für die *Argumente* der Struktur unifiziert, sonst initialisiert (erzeugt). Der Interpreter codiert implizit die Rücksprungadresse. Dabei bedeutet eine positive Adresse, daß die Struktur unifiziert (read-Modus) wird. Bei einer negativen Adresse wird die Struktur am *copy stack* angelegt (write-Modus).

pop

Bedeutet das Ende der Listen-/Strukturbearbeitung (unify_list/struct). Die am *environment stack* angelegte Adresse auf das Fortsetzungsargument der Listen/Struktur bzw. der Unterliste/-Struktur wird entfernt.

8. Literaturverzeichnis

[Berg91] Thomas Berger, Implementierung eines Prologcompilers basierend auf der Vienna Abstract Machine, Diplomarbeit, Inst. für Computersprachen, TU Wien, 1991

[Bruy82] M. Bruynooghe82: The Memory Management of Prolog Implementations, Logic Programming, Clark & Tarnlund (ed.), Academic Press, 83-98, 1982

[Chen94] Dongyang Cheng, The Implementation of a VAM based Incremental Compiler for Prolog, Diplomarbeit, Inst. für Computersprachen, TU Wien, 1994

[ClMe84] W.F. Clocksin, C.S. Mellish: Programming in Prolog; Springer Verlag, 1984

[Colm93] Alain Colmerauer. The birth of Prolog. In the Second ACM-SIGPLAN History of Programming Languages Conference, SIG-PLAN Notices, pages 37-52, ACM, March 1993

[Colo91] Robert M. Colomb. Enhancing unification in Prolog through clause indexing. Journal of Logic Programming, 10(1), 1991

[Gran96] Andreas Granzer, Garbage Collector für VIP-Prolog, Diplomarbeit, Inst. f. Computersprachen, TU Wien, 1996

[Kral86] A. Krall: Comparing Implementation Techniques for Prolog, VIP TR 1802/86/7, TU Wien, 1986

[Kral87] A. Krall: Implementation of a High-Speed Prolog Interpreter, ACM Sigplan, Conf. Interpr. and Interpretive Techn., 7(7), 1987

[Kral88] Analyse und Implementierung von Prologsystemen, Dissertation TU Wien, 1988

[KrBe95_1] Andreas Krall and Thomas Berger. The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog. In ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages. Science University of Tokyo, 1995.

[KrBe95_2] Andreas Krall and Thomas Berger. Incremental global compilation of Prolog with the Vienna Abstract Machine. In Twelfth International Conference on Logic Programming, Tokyo, 1995. MIT Press.

[KrNe90] A. Krall, U. Neumerkel: The Vienna Abstract Machine, PLIP, ´90 Programming Language Implementations and Logic Programming, Linköping, 1990. Lecture Notes in Computer Science, Springer-Verlag

[KuFr86] Eva Kühn, Thomas Frühwirth: VIP-DBS ein logikorientiertes Datenbank Management System unter VIP-Prolog, in Die Zukunft der Informationssysteme, Oldenburg, 1986

[Lloy87] J.W. Lloyd: Foundations of Logic Programming; Springer-Verlag, 1987

[Mell85] Cristopher S. Mellish. Some global optimizations for a Prolog compiler. Journal of Logic Programming, 2(1), 1985

[StSh86] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, 1986

[Warr77] David H.D.Warren: Implementing Prolog - compiling predicate logic programs; University of Edinburgh, Department of Artificial Intelligence, TR 39, TR 40, 1977

[Warr83] David H.D.Warren: An Abstract Prolog Instruction Set; SRI International, Technical Note 309, 1983