DIPLOMARBEIT

Interprozedurale Registerallokation für eine WAM mit fixem Registersatz

ausgeführt am Institut für Computersprachen,
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
o. Univ.-Prof. Dipl.-Ing. Dr. Manfred Brockhaus
und
Univ.-Ass. Dipl.-Ing. Dr. Ulrich Neumerkel
als verantwortlich mitwirkenden Universitätsassistenten

durch

Walter Binder

Engilgasse 9
A-1160 Wien
Österreich

Matrikelnummer: 9200514

Wien, im April 1997

Inhaltsverzeichnis

1 Einführung

Moderne Prolog-Systeme, wie z.B. SICStus Prolog, erzeugen aus Prolog-Programmen zunächst Code für eine Abstrakte Maschine. Dieser Code wird dann entweder interpretiert oder auf Maschinencode für einen Prozessor weiterübersetzt. Die bekannteste und am meisten verwendete Abstrakte Maschine für Prolog ist die Warren Abstract Machine (WAM).

Die WAM kommt einer Maschine für eine imperative Programmiersprache sehr nahe. Beim Aufruf eines Prädikats werden zunächst die Argumente in sogenannte Argumentregister geschrieben, dann wird der Code des aufgerufenen Prädikats ausgeführt. Jede Klausel unifiziert dabei die Argumente aus ihrem Klauselkopf mit den Termen, die durch den Aufrufer in den Argumentregistern gespeichert worden sind. Da die Klausel bei der Exekution i.a. selbst weitere Prädikate aufruft (wenn die Klausel also eine Regel ist), muß der Aufrufer der Klausel davon ausgehen, daß der Inhalt sämtlicher Register verloren gegangen ist. Wird eine Variable noch für ein späteres Ziel gebraucht, muß sie im Environment (auf einem Stack) gespeichert und bei Bedarf erneut in ein Argumentregister geladen werden. In der WAM sind also alle Register caller-saved (d.h. vom Aufrufer gesichert). Das führt oft zu großem Overhead, z.B. bleiben beim Aufruf von Fakten meist alle Register erhalten (wenn keine Register für die Unifikation von Strukturen benötigt werden), der Aufrufer nimmt dennoch immer an, daß der Inhalt aller Register verloren gehen könnte, und muß deshalb später benötigte Register sichern.

In der Architektur (im Befehlssatz) moderner RISC Prozessoren wird genau spezifiziert, welche der meist 32 general purpose Register caller saved sind, deren Inhalt darf bei der Ausführung eines Unterprogramms verloren gehen (d.h. das Unterprogramm muß den Inhalt dieser Register vor deren Verwendung nicht retten und am Ende der Routine wieder herstellen; der Aufrufer muß annehmen, daß diese Register überschrieben werden und muß sie gegebenenfalls sichern). Neben diesen caller saved Registern, die den Registern der WAM entsprechen, haben RISC Prozessoren auch einige callee saved Register. Will ein Unterprogramm eines dieser Register verwenden, muß sein Inhalt zuvor gesichert und am Ende der Routine restauriert werden. Der Aufrufer darf annehmen, daß diese Register über den Prozeduraufruf hinweg erhalten bleiben. Durch diese calling convention (d.h. Aufteilung der Register in caller saved und callee saved) bei RISC Prozessoren erreichen Compiler für imperative Programmiersprachen bei geeigneter Registerallokation wesentlich bessere Performance-Werte, als wenn sämtliche Register entweder caller saved oder callee saved wären. Für sogenannte Leaf-Prozeduren (das sind Prozeduren, die selbst keine Unterprogramme aufrufen) ist es beispielsweise sinnvoll, bevorzugt caller saved Register zu verwenden, da dann in der Prozedur kein Overhead durch Sichern bzw. Restaurieren von Registern entsteht. Bei anderen Prozeduren ist es oft vorteilhaft, für Variablen, die besonders oft und in allen Teilen des Unterprogramms verwendet werden, callee saved Register zu allokieren, weil dann vor Prozeduraufrufen keinerlei Rettungsaktionen für diese Variablen durchgeführt werden müssen.

Ähnliche Performance-Gewinne, die bei imperativen Programmiersparchen durch eine calling convention mit caller saved und callee saved Registern erzielbar sind, können auch für WAM Code, und damit für Prolog-Programme, erreichbar sein, wenn man eine fixe Aufteilung der Argumentregister der WAM in caller saved und callee saved Register vornimmt. Dazu ist es notwendig, Prolog-Programme so zu transformieren, daß alle Prädikate dieselbe Arität haben, und es wird dann genau festgelegt, welche Argument-Positionen den caller-saved Registern und welche den callee saved Registern entsprechen. Diese Aufteilung ist für jedes Prädikat gleich, und jede Klausel muß nach außen hin diese neue calling convention befolgen. Optimierungen sind dadurch möglich, daß sich Prädikate intern nicht an diese calling convention halten und alle Register möglichst geschickt verwenden, nach außen hin muß aber die calling convention gewahrt werden, eventuell indem ein Hüll-Prädikat dem Prolog-Programm beigefügt wird, das alle callee saved Register rettet, die intern verwendet werden.

Zunächst wird nun die WAM genau erklärt, da ein genaues Verständnis dieser Abstrakten Maschine unerläßlich ist, um Optimierungsmöglichkeiten im WAM-Code zu erkennen bzw. um die Auswirkungen von Optimierungen zu begutachten. Sodann wird auf jene Variation der WAM näher eingegangen, die in SICStus Prolog implementiert ist, denn die neue calling convention soll mit Hilfe dieses Prolog-System ausprobiert werden. Dabei wird besonders darauf geachtet, den WAM-Code-Generator von SICStus Prolog möglichst wenig zu ändern, vor allem soll die SICStus WAM unverändert bleiben, um nicht auch den WAM-Code-Interpreter (bzw. den WAM-Code-Compiler für einige Prozessoren) modifizieren zu müssen (die Semantik sämtlicher WAM-Instruktionen muß gleich bleiben, und es dürfen auch keine neuen WAM-Befehle eingeführt werden). Es werden drei neue low level Prolog-Prädikate beschrieben (SOURCE/n, SINK/n, VOID/1), mit deren Hilfe die neue Registerallokation ausgedrückt werden kann. Danach wird an Hand einiger Beispiele gezeigt, wie Prolog-Programme transformiert werden können, um mit diesen drei low level Befehlen die neue calling convention zu verwirklichen. Dabei werden auch die durch die Transformation erreichten Verbesserungen im WAM-Code erläutert. Als nächstes wird beschrieben, wie eine derartige Transformationen automatisch durch ein Prolog-Programm durchgeführt werden kann, und welche Optimierungen dabei möglich sind. Zum Schluß werden Benchmarks vorgestellt, um die durch die neue calling convention erreichten Performance-Gewinne zu quantifizieren.

2 Warren Abstract Machine

David H.D. Warren entwickelte 1983 die nach ihm benannte Warren Abstract Machine (WAM) zur Ausführung von Prolog-Programmen. Diese Abstrakte Maschine, die heute Standard für die Implementierung von Prolog-Übersetzern ist, wurde erstmals in [War83] in einer sehr komprimierten Weise beschrieben, die ein Verständnis der Abläufe der WAM erschwerte. In den darauffolgenden Jahren gab es kaum offizielle Publikationen über die WAM, die Übersetzer-Bauer, welche die WAM für ihre Prolog-Compiler verwenden wollten, mußten viel Zeit investieren, um die WAM zu verstehen.

In [Aït90] werden erstmals nicht alle Aspekte der WAM auf einmal präsentiert, vielmehr werden in dieser Publikation alle Konzepte isoliert beschrieben und erst später zusammengefügt. Wegen der besseren Verständlichkeit einer derartig aufgebauten Beschreibung der WAM wird hier ein ähnlicher Stil gewählt. Eine kompakte Auflistung aller WAM-Instruktionen ist in [Kra94] zu finden, wo auch die WAM mit anderen abstrakten Maschinen für Prolog verglichen wird.

2.1 Unifikation

In diesem Abschnitt wird beschrieben, wie die WAM die Unifikation eines Prolog-Ziels mit dem Kopf einer Klausel durchführt. Eines der wesentlichsten Merkmale der WAM ist, daß die Unifikation zweigeteilt ist: Das aufrufende Ziel kopiert seine Argumente in die Argumentregister, die aufgerufene Klausel unifiziert dann die Argumente ihres Klauselkopfs mit den Argumentregistern. Diese Art der Parameter-Übergabe funktioniert sehr ähnlich wie in imperativen Programmiersprachen (wie z.B. in C), durch die Zweiteilung der Unifikation kommt die WAM dem Ausführungsmodell imperativer Sprachen näher als alle anderen Implementierungsmodelle für Prolog.

Die meisten Beispiele dieses Abschnitts stammen aus dem (eigenen) Abschnitt über die WAM der Seminarbeit [BFS95].

2.1.1 Termdarstellung

Prolog-Terme werden auf dem Heap erzeugt, das ist ein Feld von Speicherzellen, deren Inhalt (Typ) durch Markierungen (Tags) angegeben wird. Speicherzellen für Variablen sind mit REF, für Strukturen mit STR markiert. Der Inhalt einer REF-Zelle ist die Heap-Adresse der Speicherzelle, an die diese Variable gebunden ist. Ungebundene Variablen verweisen auf sich selbst. Eine STR-Zelle enthält die Adresse einer unmarkierten Zelle, in der Funktor und Arität der Struktur gespeichert sind. Hat ein Funktor die Arität n ($n \geq 0$), so werden die nächsten n

Heap-Zellen, die auf die unmarkierte Zelle folgen, für die Repräsentation der n Argumente der Struktur benötigt.

Beispielsweise könnte die Heap-Repräsentation des Terms `f(A,g(A,B))`, beginnend bei Heap-Adresse 4, folgendermaßen aussehen (ganz links stehen die Heap-Adressen):

0	STR	1
1	g/2	
2	REF	2
3	REF	3
4	STR	5
5	f/2	
6	REF	2
7	STR	1

2.1.2 Ziele

Um einen Term auf dem Heap zu erzeugen, müssen zuerst alle Subterme generiert werden. Damit Heap-Daten temporär gespeichert werden können, während ein Term auf dem Heap erzeugt wird, benötigt man Register Xi ($i \geq 0$). Der Inhalt eines Registers hat die gleiche Form wie der einer Heap-Zelle. Wird ein Register benötigt, verwendet man jenes mit dem kleinsten verfügbaren Index i. Für alle Vorkommnisse einer bestimmten Variable muß immer dasselbe Register verwendet werden. Um z.B. den Term `f(A,g(A,B))` auf dem Heap zu erzeugen, setzt man die Register so ein: X1=f(X2,X3), X2=A, X3=g(X2,X4), X4=B. Da vor dem Aufbau eines Terms alle seine Subterme erzeugt werden und weil die Variablennamen egal sind, verändert man diese Sequenz folgendermaßen: X3=g(X2,X4), X1=f(X2,X3). Nun ersetzt man alle Registerzuweisungen der Gestalt Xi=f(Xi_1,...,Xi_n) durch die Sequenz Xi=f/n,Xi_1,...,Xi_n und erhält somit X3=g/2, X2, X4, X1=f/2, X2, X3. Aus dieser Folge von Registern und Zuweisungen eines Strukturfunktors an ein Register läßt sich leicht der WAM-Code für die Erzeugung des Terms am Heap generieren: Für Xi=f/n wird die WAM-Instruktion **put_structure f/n, X**i erzeugt, für das erste Vorkommnis eines Registers Xi der Befehl **set_variable X**i und für jedes weitere Vorkommnis die Instruktion **set_value X**i. Aus obiger Sequenz erhält man auf diese Art folgenden WAM-Code für den Aufbau des Terms `f(A,g(A,B))`:

```
put_structure g/2, X3     % X3=g
set_variable X2           %     (A,
set_variable X4           %        B)
put_structure f/2, X1     % X1=f
set_value X2              %     (A,
set_value X3              %        X3)
```

put_structure f/n, Xi: Dieser Befehl legt eine STR-Zelle sowie eine daraufolgende unmarkierte Zelle (auf welche die STR-Zelle verweist), die den Funktor f/n enthält, auf dem Heap an und kopiert die STR-Zelle in das Register Xi.

set_variable Xi: Diese Instruktion generiert eine ungebundene Variable (eine REF-Zelle, die auf sich selbst verweist) auf dem Heap und kopiert sie in das Register Xi.

set_value Xi: Dieser Befehl generiert eine Heap-Zelle und weist ihr den Inhalt des Registers Xi zu.

Bei der Implementierung dieser drei WAM-Instruktionen wird ein Zeiger auf das momentane Heap-Ende benötigt, dieser heißt H (top of heap). Als letzte Aktion inkrementieren alle drei Befehle H entsprechend, sodaß er wieder auf die nächste freie Heap-Zelle verweist (**put_structure** muß also H um 2 erhöhen, **set_variable** und **set_value** nur um 1).

Obiger WAM-Code zur Erzeugung des Terms `f(A,g(A,B))` würde genau jene Heap-Repräsentation generieren, die im vorigen Abschnitt dargestellt ist, wenn man annimmt, daß vor der Exekution $H = 0$ gilt.

2.1.3 Klauselköpfe

Hier wird angenommen, daß bereits ein Anfrage-Term auf dem Heap erzeugt worden ist, wie im vorigen Abschnitt erklärt. Dabei soll das Register X1 auf die Wurzel des Terms verweisen. Bei der Unifikation dieses Terms mit einem Klauselkopf folgt man der Term-Struktur, die durch Register X1 gegeben ist, solange die Funktoren im Anfrage-Term der Struktur des Kopf-Termes entsprechen. Stößt man dabei auf eine ungebundene Variable (REF-Zelle, die auf sich selbst verweist), wird diese an den entsprechenden Subterm des Klauselkopfs gebunden, der auf dem Heap generiert werden muß. Die WAM-Instruktionen zur Unifikation in Klauselköpfen können also in einem Lese- oder in einem Schreib-Modus arbeiten.

Wie bei der Übersetzung von Zielen müssen auch bei der Kompilation von Klauselköpfen zuerst Register allokiert werden. Soll beispielsweise WAM-Code für den Klauselkopf `f(A,g(A,A))` erzeugt werden, so muß dieser Term zunächst in diese Sequenz umgewandelt werden: X1=f(X2,X3), X2=A, X3=g(X2,X2). Da Variablennamen keine Rolle spielen, erhält man daraus folgende Sequenz: X1=f(X2,X3), X3=g(X2,X2). Weil angenommen wird, daß der Anfrage-Term bereits am Heap zur Verfügung steht, darf hier die Reihenfolge dieser Sequenz von Registerzuweisungen nicht geändert werden; der Klauselkopf wird in einer Top-Down-Ordnung zerlegt, während der Anfrage-Term Bottom-Up generiert worden ist. Analog zur Kompilation von Zielen wandelt man die Termgleichungen in eine Sequenz von Registern und Zuweisungen eines Funktors an ein Register um. Für obiges Beispiel erhält man somit die Folge X1=f/2, X2, X3, X3=g/2, X2, X2, aus welcher der WAM-Code für den Klauselkopf `f(A,g(A,A))` folgendermaßen erzeugt wird: Für jede Zuweisung eines Funktors f/n an ein Register Xi verwendet man die WAM-Instruktion **get_structure f/n, X**i, für das erste Vorkommnis eines Registers Xi wird der Befehl **unify_variable X**i generiert, für jedes weitere Vorkommnis **unify_value** Xi. Für den Klauselkopf `f(A,g(A,A))` ergibt sich somit folgender WAM-Code:

```
get_structure f/2, X1     % X1=f
unify_variable X2         %     (A,
unify_variable X3         %        X3)
get_structure g/2, X3     % X3=g
unify_value X2            %     (A,
unify_value X2            %        A)
```

get_structure f/n, Xi**:** Falls die Heap-Zelle, auf die das Register Xi verweist, eine ungebundene Variable beinhaltet, erzeugt **get_structure** eine STR-Zelle mit nachfolgender unmarkierter Zelle (auf welche die STR-Zelle zeigt), die den Funktor f/n enthält, bindet die Variable an diese STR-Zelle und schaltet den Schreib-Modus ein. Andernfalls muß sich am Heap eine STR-Zelle befinden, die auf eine Struktur mit demselben Namen und derselben Arität verweist (sonst scheitert die Unifikation). In diesem Fall wird der Lese-Modus aktiviert, und ein Heap-Zeiger S (structure pointer) wird so gesetzt, daß er auf das erste Argument der Struktur zeigt.

unify_variable Xi**:** Diese Instruktion weist im Lese-Modus dem Register Xi den Inhalt der Heap-Zelle zu, auf die S zeigt, und inkrementiert S. Im Schreib-Modus erzeugt **unify_variable** eine ungebundene Variable auf dem Heap und weist dem Register Xi diese neue REF-Zelle zu.

unify_value Xi**:** Dieser Befehl unifiziert im Lese-Modus den Term, dessen Heap-Repräsentation an der Adresse S beginnt, mit jenem Term, auf den das Register Xi verweist. Dabei wird ein Unifikations-Stack, der Speicherbereich

PDL (Push-Down List), verwendet. Anschließend wird S um 1 erhöht. Im Unifikationsalgorithmus muß berücksichtigt werden, daß durch Variablenbindungen Referenz-Ketten entstehen können, d.h. daß Variablen auch auf andere Variablen verweisen können; es wird eine WAM-Hilfsroutine zum Dereferenzieren von Variablen benötigt, die einer Bindungskette solange folgt, bis sie auf eine ungebundene Variable bzw. eine STR-Zelle trifft. Wird im Verlauf der Unifikation eine Variable an eine Struktur gebunden, müßte eigentlich überprüft werden, daß durch die Unifikation kein zyklischer Term entsteht (d.h. die Struktur darf die Variable, die an die Struktur gebunden wird, nicht enthalten). Diese Überprüfung, auch Occur-Check genannt, wird jedoch aus Effizienzgründen von den meisten Implementierungen nicht durchgeführt, darüber hinaus kann man zyklischen Terme manchmal beim Programmieren sinnvoll einsetzen. Im Schreib-Modus generiert **unify_value** eine neue Heap-Zelle, der dann der Register-Inhalt von Xi zugewiesen wird.

Werden durch die WAM-Instruktionen **get_structure**, **unify_variable** bzw. **unify_value** Heap-Zellen allokiert, muß natürlich auch der Heap-Zeiger H (top of heap) entsprechend inkrementiert werden, sodaß er immer auf die nächste frei verfügbare Heap-Zelle verweist.

2.1.4 Argumentregister

Bisher ist nur gezeigt worden, wie die Unifikation zweier Prolog-Terme mit Hilfe der WAM durchgeführt werden kann. Jetzt sollen Prolog-Programme, die nur aus Anfragen und Fakten bestehen, in WAM-Code übersetzt werden. Einstweilen soll die Einschränkung gelten, daß für jeden Prädikatsnamen höchstens ein Fakt existiert.

Der WAM-Code für ein Prolog-Programm wird im Code-Bereich der WAM gespeichert, das ist ein adressierbares Feld von WAM-Instruktionsworten, wobei jedes aus Opcode und Operanden besteht. Die Instruktionen im Code-Bereich können durch symbolische Einsprungspunkte (labels) markiert sein. Diese Markierungen werden als Parameter für jene WAM-Befehle verwendet, die den WAM-Code von Prädikaten aufrufen. Es ist daher nicht notwendig, Prädikatsnamen im Heap zu speichern, denn im WAM-Code werden Prädikatsnamen einfach durch Einsprungspunkte im Code-Bereich repräsentiert. Die WAM-Instruktion **call p/n** bewirkt, daß der Kontrollfluß zu dem mit p/n markierten WAM-Befehl verzweigt. Das WAM-Register P (program counter) enthält immer die Adresse der Instruktion, die als nächste zu exekutieren ist. Alle Put- und Get-Befehle (bzw. Set- und Unify-Instruktionen), die bisher beschrieben worden sind, inkrementieren das Register P zum Schluß, sodaß der folgende WAM-Befehl als nächstes ausgeführt wird. Sofern nichts anderes angegeben, wird bei allen weiteren WAM-Instruktionen implizit angenommen, daß sie P entsprechend (um die Größe des jeweiligen Befehls) erhöhen.

Es muß nun zwischen Literalen (das sind Terme, deren Funktor ein Prädikatsname ist) und Termen (das sind die Argumente eines Prädikats) unterschieden werden. Dadurch muß eine Anfrage jetzt n Argumente an ein Prädikat der Arität n übergeben. Während im vorigen Abschnitt das Register X1 ausgereicht hat, um die Wurzel des kompletten Ziel-Termes aufzunehmen, dessen Hauptfunktor der Name des aufgerufenen Prädikats gewesen ist, benötigt man nun n Register für die Argumentübergabe. Die Register X1,...,Xn sollen immer für die Wurzeln der n Argument-Terme allokiert werden. Um deutlich zu machen, daß ein Register Xi zur Argumentübergabe verwendet wird, also die Wurzel eines Argument-Termes enthält, wird es absofort als Argumentregister Ai bezeichnet. Xi bezeichnet ein WAM-Register, das zum Aufbau (bzw. zur Unifikation) von Strukturen benötigt wird, man spricht auch vom temporären Register Xi. Ai und Xi bezeichnen dasselbe WAM-Register, die Bezeichnung hängt davon ab, wie dieses Register verwendet wird.

Weil der Prädikatsname nicht mehr am Heap erzeugt wird, ist es nun möglich, daß Variablen als Wurzeln von Argument-Termen auftreten. Es sind daher spezielle WAM-Instruktionen nötig, um Variablen in Argumentregistern zu speichern (für Anfragen) und um sie aus den Argumentregistern zu extrahieren (für Fakten). Für das erste Auftreten des Registers Xn in der i-ten Argumentposition bei einer Anfrage verwendet man den WAM-Befehl **put_variable Xn, Ai**, der eine ungebundene Variable am Heap erzeugt und diese sowohl dem Register Xn als auch Ai zuweist (der Heap-Zeiger H muß natürlich auch inkrementiert werden). Für weitere Vorkommnisse wird die Instruktion **put_value Xn, Ai** eingesetzt, die dem Register Ai den Inhalt des Registers Xi zuweist. Bei Fakten generiert man für das erste Auftreten eines Registers Xn in der Argumentposition i den Befehl **get_variable Xn, Ai**, der in Xn den Inhalt von Ai speichert. Für spätere Vorkommnisse erzeugt man die Instruktion **get_value Xn, Ai**, welche die Terme unifiziert, auf deren Wurzeln Xn und Ai zeigen. Dabei wird derselbe Unifikationsalgorithmus (dasselbe WAM-Unterprogramm) verwendet wie im Lese-Modus der WAM-Instruktion **unify_value Xi** (siehe voriger Abschnitt).

Es folgen nun zwei Beispiele für die Erzeugung von WAM-Code für eine Anfrage und für ein Fakt. Zunächst soll die Anfrage '`:- p(A,f(A,B),B).`' in WAM-Code übersetzt werden. Die Registerallokation ergibt folgende Sequenz: A1=X4, A2=f(X4,X5), A3=X5, X4=A, X5=B. Da Variablennamen keine Rolle spielen, erhält man daraus die Folge A1=X4, A2=f(X4,X5), A3=X5. Der WAM-

Code für die Anfrage '`:- p(A,f(A,B),B).`' sieht somit folgendermaßen aus:

```
put_variable X4, A1      % :- p(A,
put_structure f/2, A2    %      f
set_value X4             %       (A,
set_variable X5          %          B),
put_value X5, A3         %             B
call p/3                 %              ).
```

Jetzt soll WAM-Code für das Fakt '`p(A,f(A,B),B).`' erzeugt werden. Dabei wird das Fakt durch die WAM-Instruktion **proceed** abgeschlossen, deren Wirkungsweise später genau erklärt wird.

```
p/3: get_variable X4, A1      % p(A,
     get_structure f/2, A2    %    f
     unify_value X4           %     (A,
     unify_variable X5        %        B),
     get_value X5, A3         %           B
     proceed                  %            ).
```

2.2 Flat Resolution

In diesem Abschnitt wird erklärt, wie Prolog-Programme, die für jeden Prädikatsnamen höchstens eine Klausel (Fakt oder Regel) enthalten, in WAM-Code übersetzt werden. Es wird hier also die WAM-Code-Generierung für Prolog ohne Backtracking beschrieben.

2.2.1 Fakten

Die Übersetzungstechnik für Fakten, wie sie im vorigen Abschnitt dargestellt ist, kann direkt übernommen werden. Die einzige Ausnahme betrifft die WAM-Instruktion **proceed**: Während im vorigen Abschnitt die Exekution des Programms, das nur aus Fakten (je Prädikatsname höchstens ein Fakt) und einer Anfrage bestanden ist, die nur ein Ziel gehabt hat, nach dem Proceed-Befehl erfolgreich beendet war, muß hier sichergestellt werden, daß nach erfolgreicher Ausführung des WAM-Codes für das Fakt die Exekution bei jenem Befehl weiterläuft, der dem Aufruf des Fakts nachfolgt. Dazu wird das neue WAM-Register CP (continuation program counter) benötigt, das wie P (program counter) ebenfalls den Code-Bereich addressiert. CP wird beim Aufruf eines Prädikats so gesetzt, daß es auf die dem Aufruf folgende Instruktion verweist. Der WAM-Befehl

call p/n lädt also die Summe des Inhalts des Registers P plus die Größe der Instruktion, auf die P zeigt (das ist der Befehl **call p/n** selbst), in das Register CP, dann wird P auf den Anfang des WAM-Codes für das Prädikat p/n gesetzt. Die WAM-Instruktion **proceed** kopiert einfach den Inhalt des Registers CP in das Register P.

2.2.2 Regeln und Anfragen

Eine Regel '$kopf$`(...) :-` $ziel_1$`(...), ...,` $ziel_n$`(...).`' wird so übersetzt, daß Get-Instruktionen für $kopf$ und Put-Instruktionen für $ziel_i$ generiert werden, d.h. der WAM-Code könnte in etwa wie folgt aussehen:

> Argumente von $kopf$ mittels Get-Instruktionen unifizieren
> Argumente von $ziel_1$ mittels Put-Instruktionen schreiben
> call $ziel_1$
> ...
> Argumente von $ziel_n$ mittels Put-Instruktionen schreiben
> call $ziel_n$

Es müssen einerseits Vorkehrungen getroffen werden, damit die Exekution nach Ausführung des Codes einer Regel korrekt weiterläuft, d.h. CP (continuation program counter) muß gerettet, bevor die erste Call-Instruktion exekutiert wird (denn diese verändert CP), und ganz zum Schluß (nach dem letzten Call-Befehl) wiederhergestellt werden.

Andererseits müssen auch Variablen, die in mehr als einem Ziel auftreten, vor Seiteneffekten durch Put-Instruktionen geschützt werden; diese heißen permanente Variablen und weren hier mit Yi bezeichnet. Alle anderen Variablen nennt man temporär (Bezeichnung wie bisher Xi. Um zu erkennen, ob eine Variable permanent ist, wird der Regelkopf zusammen mit dem ersten Ziel betrachtet. Kommt nun eine Variable in mehreren Zielen vor, ist sie permanent. Betrachtet man z.B. die Regel '`p(A,C) :- q(A,B), r(B,C).`', erkennt man, daß die Variablen B und C permanent sind. Wären sie nur in Argumentregistern gespeichert, könnten sie nach Ausführung des WAM-Codes für das Prädikat q/2 verloren gegangen sein, da ja der Code für q/2 selbst die Argumentregister verwendet.

Permanente Variablen entsprechen lokalen Variablen in imperativen Programmiersprachen, deshalb kann auch die Lösung direkt von imperativen Sprachen übernommen werden: Für jede Aktivierung einer Regel wird Speicher für die permanenten Variablen (ein sogenanntes Environment) auf einem Stack allokiert. Dieser neue WAM-Speicherbereich heißt Environment-Stack, seine Spitze wird durch den Stack-Zeiger E (current environment pointer) markiert. Im Environment einer Regel wird neben den permanenten Variablen Yi auch CP gesichert.

Bei der Aktivierung des WAM-Codes für eine Regel wird ein neues Environment auf dem Stack angelegt (Push), und am Ende des Codes wird dieses Environment wieder vom Stack entfernt (Pop). Da die Environment-Größe bei jeder Regel unterschiedlich sein kann, wird der Stack als Liste organisiert, d.h. eine weitere Speicherstelle wird im Environment allokiert, um einen Zeiger auf das Vorgänger-Environment (den alten Wert von E) aufzunehmen. Ein Environment benötigt auf dem Stack also mindestens 2 Speicherzellen: eine für die Verkettung der Environments und eine, um CP aufzunehmen. Zusammenfassend sieht ein Environment folgendermaßen aus (diese Skizze ist aus [Aït90] genommen):

E	CE	(continuation environment)
E+1	CP	(continuation point)
E+2	Y1	(permanente Variable 1)
...	...	
E+n+1	Yn	(permanente Variable n)

Es werden zwei neue WAM-Instruktionen benötigt:

allocate: Dieser Befehl erzeugt ein neuen Environments auf dem Stack, wobei das Feld CE (continuation environment) dieses Environments den aktuellen Wert des WAM-Registers E (current environment pointer) aufnimmt und im Feld CP (continuation point) das WAM-Register CP (continuation program pointer) gesichert wird. Das Register E wird so gesetzt, daß es auf den Anfang des neuen Environments zeigt.

deallocate: Diese WAM-Instruktion restauriert die WAM-Register E und CP mit den gesicherten Werten des aktuellen Environments, d.h. das Register CP wird wiederhergestellt und das Environment wird deallokiert (E wird auf das Vorgänger-Environment gesetzt).

Für eine Regel der Gestalt '*kopf*`(...) :-` $ziel_1$`(...), ...,` $ziel_n$`(...).`' wird WAM-Code nach folgendem Muster erzeugt:

allocate
Argumente von *kopf* mittels Get-Instruktionen unifizieren
Argumente von $ziel_1$ mittels Put-Instruktionen schreiben
call $ziel_1$
...
Argumente von $ziel_n$ mittels Put-Instruktionen schreiben
call $ziel_n$
deallocate

Als Beispiel soll WAM-Code für die Regel '`p(A,C) :- q(A,B), r(B,C).`' generiert werden:

```
p/2: allocate                % p
     get_variable X3, A1     %  (A,
     get_variable Y1, A2     %     C) :-
     put_value X3, A1        %           q(A,
     put_variable Y2, A2     %               B
     call q/2                %                ),
     put_value Y2, A1        %           r(B,
     put_value Y1, A2        %               C
     call r/2                %                )
     deallocate              %                 .
```

Es ist zu beachten, daß die Put- bzw. Get-Instruktionen, deren Argumente permanente Register Yi sind, neue, bisher nicht vorgestellte WAM-Instruktionen sind. Sie funktionieren genauso wie die entsprechenden WAM-Befehle für temporäre Register Xi, jedoch greifen sie eben nicht auf ein Register Xi zu, sondern auf das entsprechende Feld im aktuellen Environment (für die permanente Variable Yi ist das die Stack-Adresse E+i+1).

2.3 Prolog

In diesem Abschnitt wird das bisherige WAM-Design so erweitert, daß es sich für die Ausführung von puren Prolog-Programmen eignet, d.h. ab sofort dürfen zu jedem Prädikatsnamen beliebig viele Klauseln (Regeln und Fakten) definiert werden. Das Scheitern einer Unifikation bedeuted also nicht mehr notwendigerweise das Scheitern einer Anfrage, sondern es kann mittels Backtracking der nächste Lösungsweg versucht werden.

Daher ist es erforderlich, daß der Zustand der Berechnung gesichert wird, falls ein Prädikat mehrere Alternativen bietet; der gesicherte Zustand wird Wahlpunkt (choice point) genannt. Beim Backtracking muß dann der alte Berechnungszustand wiederhergestellt werden, u.a. müssen Variablenbindungen eventuell rückgängig gemacht werden.

Die Wahlpunkte werden als Stack organisiert (analog zu den Environments), beim Backtracking wird immer die Information des obersten Wahlpunktes verwendet, wodurch sich die operationale Semantik von Prolog ergibt. Im folgenden wird der Environment-Stack auch Und-Stack, der Wahlpunkt-Stack auch Oder-Stack genannt. Wie der Und-Stack wird auch der Oder-Stack als verkettete Liste realisiert, an der Spitze des Oder-Stacks befindet sich immer der jüngste (aktuelle) Wahlpunkt, auf den das neue WAM-Register B (most recent choice point) zeigt. Beim Backtracking wird der Berechnungszustand vom Wahlpunkt, auf den B verweist, wiederhergestellt, und der Wahlpunkt selbst wird aktualisiert, damit im Fall von Backtracking die nächste Alternative probiert wird. Gibt es keine weiteren Alternativen, muß der Wahlpunkt vom Oder-Stack entfernt und B auf dessen Vorgänger gesetzt werden. Ist beim Backtracking der Oder-Stack leer, scheitert die Berechnung endgültig. Wahlpunkte müssen nur dann auf dem Oder-Stack generiert werden, wenn ein aufgerufenes Prädikat aus mehr als einer Klausel besteht. Der Wahlpunkt wird in diesem Fall von der ersten Alternative erzeugt, weitere Alternativen modifizieren ihn und die letzte entfernt den Wahlpunkt vom Oder-Stack.

2.3.1 Environment Protection

Während es im vorigen Abschnitt unproblematisch war, den Speicherplatz eines Environments, das durch **deallocate** freigegeben worden war, mit einem neuen Environment zu überschreiben, da noch keine Möglichkeit für Backtracking vorgesehen war, muß hier beachtet werden, daß durch Backtracking der Berechnungszustand mitten in einer Regel restauriert werden kann, deren Environment bereits freigegeben und möglicherweise schon überschrieben worden ist. Diese

Problematik soll an folgendem Beispiel illustriert werden:

```
p :-                t(a).
     q(A),
     r(A).          u(b).

q(A) :-             v(a).
     s(A).

r(A) :-
     t(A)

s(A) :-
     u(A).
s(A) :-
     v(A).
```

Die Ausführung der Anfrage '`:- p.`' erzeugt zunächst ein Environment für p/0 (mit der permanenten Variable A), dann wird q/1 aufgerufen. Nachdem ein Environment für q/1 erzeugt worden ist, wird s/1 exekutiert. Weil s/1 aus zwei Klauseln besteht, wird ein Wahlpunkt am Oder-Stack erzeugt, dann wird ein Environment für die erste Alternative von s/1 am Und-Stack generiert. Die beiden Stacks sehen nun folgendermaßen aus:

E →	Environment für s/1
	Environment für q/1
	Environment für p/0
	...

B →	Wahlpunkt für s/1
	...

Als nächstes wird u/1 aufgerufen (es entsteht kein neues Environment, weil u/1 ein Fakt ist), wodurch die Variable A an das Atom `b` gebunden wird. Nun können die Environments für s/1 und q/1 deallokiert werden, die beiden Stacks befinden sich jetzt in diesem Zustand:

E →	Environment für p/0
	...

B →	Wahlpunkt für s/1
	...

Nun wird der Regelkörper von p/0 weiter exekutiert, r/1 wird ausgeführt und erzeugt ein neues Environment, welches das deallokierte Environment von q/1 überschreibt. Die Stacks sehen somit folgendermaßen aus:

E →	Environment für r/1
	Environment für p/0
	...

B →	Wahlpunkt für s/1
	...

Jetzt wird t/1 aufgerufen, aber die Unifikation mit dem Kopf dieses Fakts scheitert, da das Argument bereits an das Atom `b` gebunden ist. Der Wahlpunkt auf dem Oder-Stack zeigt an, daß es noch eine Alternative für s/1 gibt, aber zu diesem Zeitpunkt ist das Environment von q/1 bereits durch jenes von r/1 überschrieben.

Um derartige Fehler zu vermeiden, dürfen keine Environments überschrieben werden, die vor irgendeinem Wahlpunkt auf dem Oder-Stack erzeugt worden sind. Das bedeuted, daß ein Wahlpunkt alle Environments vor Überschreiben schützen muß, die vor seiner Erzeugung schon existiert haben. Aus diesem Grund allokiert man sowohl Environments als auch Wahlpunkte im selben Speicherbereich, der jetzt nur mehr Stack genannt wird. Sowohl E (current environment pointer) als auch B (most recent choice point) verweisen auf diesen Stack. Solange sich ein Wahlpunkt am Stack befindet, werden alle neu allokierten Environments oberhalb dieses Wahlpunkts erzeugt, wenn Environments, die älter als der Wahlpunkt sind, deallokiert werden, kann es somit nicht passieren, daß ihr Speicherbereich durch neue Environments überschrieben wird. Dieser Schutz der deallokierten Environments wirkt nur so lange, wie unbedingt notwendig; nachdem der Wahlpunkt vom Stack entfernt worden ist, können die deallokierten Environments überschrieben werden.

Beim obigen Beispiel ergibt sich nach Aufruf von p/0, q/1, s/1 folgender Stack-Zustand:

E →	Environment für s/1
B →	Wahlpunkt für s/1
	Environment für q/1
	Environment für p/0
	...

Nachdem u/1 aufgerufen und die Variable A an das Atom `b` gebunden worden ist, werden die Environments für s/1 und q/1 deallokiert, wodurch der Stack nun so aussieht:

B →	Wahlpunkt für s/1
	deallokiertes Environment für q/1
E →	Environment für p/0
	...

Jetzt wird das Environment für r/1 allokiert; hier zeigt sich die Wirkung der Environment Protection, das neue Environment entsteht oberhalb des Wahlpunkts für s/1, wodurch das deallokierte Environment für q/1 erhalten bleibt:

E →	Environment für r/1
B →	Wahlpunkt für s/1
	deallokiertes Environment für q/1
	Environment für p/0
	...

Da t/1 scheitert, wird der im Wahlpunkt von s/1 gespeicherte Zustand restauriert, wodurch das Environment von q/1, welches zwar deallokiert, aber nicht überschrieben worden ist, wieder aktuell wird (d.h. das Register E zeigt wieder darauf). Als nächstes wird die zweite Klausel für s/1 probiert. Da dies die letzte Alternative ist, wird der Wahlpunkt vom Stack entfernt (das Register B verweist somit auf den Anfang des Stacks, um anzuzeigen, daß es keine weiteren Wahlpunkte mehr gibt). Es wird nun das Environment für die zweite Klausel von s/1 allokiert:

E →	Environment für s/1
	Environment für q/1
	Environment für p/0
B →	...

In weiterer Folge bindet v/1 die Variable A an das Atom `a`, und somit gelingt auch r/1 und die Anfrage '`:- p.`'.

2.3.2 Trail

Während versucht wird, eine Klausel zu beweisen, entstehen am Heap und am Stack Variablenbindungen, die beim Backtracking rückgängig gemacht werden müssen. Die Adressen jener Variablen (am Heap oder am Stack), deren Bindungen beim Backtracking zurückzunehmen sind, werden im Trail registriert, einer neuen WAM-Speicherregion. Der Trail ist als Stack organisiert, seine Spitze wird durch das WAM-Register TR (top of trail) angezeigt. Im Trail müssen sich nur Verweise auf jene Variablen befinden, die bereits vor der Erzeugung des aktuellen Wahlpunkts existiert haben. Bei der Generierung eines Wahlpunkts wird das momentane Heap-Ende (also der Wert des WAM-Registers H) im neuen WAM-Register HB abgespeichert, es müssen dann nur Bindungen von Heap-Variablen, deren Adressen kleiner als der Wert des Registers HB sind, im Trail registriert werden. Ebenso müssen nur jene Stack-Variablen im Trail aufgenommen werden, deren Adressen kleiner als B sind (d.h. sie haben schon vor dem jüngsten Wahlpunkt existiert).

2.3.3 Wahlpunkt

Hier wird zusammengefaßt, welche Informationen in einem Wahlpunkt gespeichert werden müssen:

- Die Argumentregister A1,...,An, für ein Prädikat der Arität n, das aus mehreren alternativen Definitionen (Klauseln) besteht. Dies ist nötig, da beim Aufruf von Klauseln, die während der Ausführung einer Alternative zu beweisen sind, Argumentregister mittel Put-Instruktionen neu geladen werden.
- Der momentane Wert des WAM-Registers E (current environment pointer), um jenes Environment, das vor Erzeugung des Wahlpunkts aktuell war, wiederherzustellen (dieses Environment und seine Vorgänger werden ja durch den Wahlpunkt vor Überschreiben geschützt).
- Das WAM-Register CP (continuation program counter).
- Der alte Wert des WAM-Registers B (most recent choice point), denn B muß wieder auf den Vorgänger-Wahlpunkt gesetzt werden, falls der neue Wahlpunkt keine Alternativen mehr bietet. Dadurch sind die Wahlpunkte als Liste verkettet.
- Ein Verweis auf jene Klausel, deren Beweis beim Backtracking als nächstes versucht werden soll. Dieses Feld wird beim Backtracking aktualisiert.
- Der aktuelle Wert des WAM-Registers TR (top of trail). Beim Backtracking werden dann die Bindungen all jener Variablen zurückgenommen, die sich im Trail oberhalb der in diesem Feld gespeicherten Adresse befinden.
- Das momentane Heap-Ende, das WAM-Register H (top of heap), denn H soll beim Backtracking auf diesen Wert zurückgesetzt werden, um den Heap-Speicher, der während der letzten Alternative verbraucht worden ist, freizugeben. Durch dieses Feld ist das WAM-Register HB eigentlich überflüssig, aus Effizienzgründen ist die Verwendung des Registers HB trotzdem sinnvoll, da bei jeder Bindung einer Variable geprüft werden muß, ob sie im Trail zu registrieren ist. Ohne das Register HB müßte bei dieser Überprüfung mittels des WAM-Registers B auf dieses Feld des aktuellen Wahlpunkts zugegriffen werden, was einen Speicherzugriff mehr (und eventuell Adreßarithmetik) verursacht (in der Wahlpunkt-Organisation, wie sie in [Aït90] beschrieben ist, wären zwei Speicherzugriffe notwendig, weil sich dort dieses Feld an der Adresse B+n+6 befindet, wobei n an der Adresse B gespeichert ist).

Die folgende graphische Darstellung eines Wahlpunkts ist aus [Aït90] übernommen, die bisher nicht verwendete Bezeichnungen BP für den Zeiger auf die nächste Alternative stammt ebenfalls aus [Aït90]:

B	n	Anzahl der Argumente
B+1	A1	Argumentregister 1
...	...	
B+n	An	Argumentregister n
B+n+1	E	current environment pointer
B+n+2	CP	continuation program counter
B+n+3	B	letzer Wahlpunkt
B+n+4	BP	nächste Klausel
B+n+5	TR	top of trail
B+n+6	H	top of heap

2.3.4 Backtracking

Drei neue WAM-Instruktionen dienen der Erstellung, Modifikation und Deallokation von Wahlpunkten am Stack (L bezeichnet dabei das Label einer WAM-Instruktion im Code-Bereich):

try_me_else L: Dieser Befehl wird bei der Übersetzung der ersten Klausel eines Prädikats, das aus mehreren Definitionen besteht, verwendet. Er allokiert einen neuen Wahlpunkt am Stack und speichert darin die Arität des Prädikats n, die Argumentregister A1,...,An und die WAM-Register E, CP, B, TR sowie H. Weiters muß der neue Wahlpunkt auch den Zeiger auf den Code der nächsten Alternative L (das Argument der Instruktion **try_me_else**) aufnehmen. Außerdem wird B so gesetzt, daß dieses Register auf den neuen Wahlpunkt verweist, und HB wird mit dem Wert von H geladen.

retry_me_else L: Diese Instruktion wird für alle „mittleren“ Klauseln einer Prädikatsdefinition eingesetzt (d.h. für alle Klauseln bis auf die erste und letzte). Ihre Aufgabe ist es, alle Variablenbindungen zurückzunehmen, die seit der Erzeugung des Wahlpunkts im Trail aufgenommen worden sind. Weiters werden die im Wahlpunkt gespeicherten Argumentregister, E, CP, TR und H mit der Information aus dem Wahlpunkt restauriert. HB muß ebenfalls auf den neuen Wert von H gesetzt werden. Schließlich ist noch der Zeiger auf den Code der nächsten Klausel im Wahlpunkt zu aktualisieren (mit dem Parameter L von **retry_me_else**).

trust_me: Dieser Befehl wird für die letzte Klausel benötigt, seine Aufgabe ist die Deallokation des Wahlpunkts. Zunächst werden genauso wie bei der WAM-Instruktion **retry_me_else** im Trail registrierte Variablenbindungen zurückgenommen und die Argumentregister sowie die Register E, CP, TR, H und HB wiederhergestellt. Dann wird das WAM-Register B auf den vorangegangenen Wahlpunkt gesetzt.

WAM-Code für ein Prädikat p, dessen Definition aus 2 Klauseln besteht, wird nach folgendem Schema erzeugt:

```
p:   try_me_else p_1           % Wahlpunkt erzeugen
     Code für erste Klausel
p_1: trust_me                  % Wahlpunkt entfernen
     Code für zweite Klausel
```

Das Muster für die Generierung von WAM-Code für ein Prädikat p, das mindestens 3 Alternativen bietet, sieht folgendermaßen aus:

```
p:   try_me_else p_1           % Wahlpunkt erzeugen
     Code für erste Klausel
p_1: retry_me_else p_2         % Wahlpunkt aktualisieren
     Code für zweite Klausel
...
p_i: retry_me_else p_i+1       % Wahlpunkt aktualisieren
     Code für i-te Klausel
...
p_n: trust_me                  % Wahlpunkt entfernen
     Code für letzte Klausel
```

2.4 Optimierungen

Die bisher vorgestellten WAM-Instruktionen reichen bereits aus, um pure Prolog-Programme auf WAM-Code zu übersetzen, der allerdings in vielen Fällen sehr ineffizient ist. In diesem Abschnitt werden Optimierungen vorgestellt, die durch folgende Richtlinien motiviert sind:

1. Der Heap soll so wenig wie möglich verwendet werden, da Terme, die am Heap erzeugt werden, eine relativ lange Lebensdauer haben (Heap-Speicher wird nur beim Backtracking freigegeben).

2. Bei der Registerallokation ist darauf zu achten, daß unnötige Daten-Bewegungen möglichst vermieden werden. Somit wird auch der resultierende WAM-Code kleiner.

3. Situationen (die zwar durch die bisher beschriebenen WAM-Instruktionen korrekt behandelt werden), die in Prolog-Programmen häufig auftreten, sollen auf spezielle WAM-Befehle übersetzt werden, wenn dadurch Speicherplatz gespart werden kann bzw. falls die speziellen Instruktionen einfacher sind und damit die Laufzeit kürzer wird (wenn der WAM-Code interpretiert oder auf Code für einen Prozessor weiterübersetzt wird).

2.4.1 Konstanten, Listen, anonyme Variablen

Konstanten Hier werden spezielle WAM-Befehle zur Behandlung von Konstanten (Strukturen der Arität 0), Listen und anonymen Variablen (das sind Variablen, die in einer Klausel nur 1 Mal vorkommen) vorgestellt. Diese Situationen treten in Prolog-Programmen sehr häufig auf und sollen deshalb mit speziellen WAM-Instruktionen behandelt werde, die Heap-Speicher sparen, die Code-Größe verringern und Register sparen.

Um beispielsweise den Term `f(a,b)` auf dem Heap zu erzeugen, wurde bisher folgender WAM-Code generiert:

```
put_structure a/0, X2     % X2=a
put_structure b/0, X3     % X3=b
put_structure f/2, A1     % A1=f
set_value X2              %     (X2,
set_value X3              %         X3)
```

Die Heap-Repräsentation dieses Terms (beginnend an der Heap-Adresse 4) sieht dann so aus (unter der Annahme, daß das WAM-Register H zuvor den Wert 0 enthalten hat):

0	STR	1
1	a/0	
2	STR	3
3	b/0	
4	STR	5
5	f/2	
6	STR	1
7	STR	3

Durch Verwendung einer speziellen WAM-Instruktion **set_constant c**, die am Heap ein Feld mit der neuen Markierung CON erzeugt, das die Konstante c enthält, können Register- und Heap-Speicher-Verbrauch sowie Code-Größe verringert werden, was natürlich auch zu großen Performance-Gewinnen zur Laufzeit führt. Der Code für obiges Beispiel sieht mit diesem neuen Befehl folgendermaßen aus:

```
put_structure f/2, A1     % A1=f
set_constant a            %     (a,
set_constant b            %        b)
```

Am Heap werden in diesem Beispiel 4 Speicherzellen eingespart (die neue Heap-Darstellung des Terms `f(a,b)` beginnt hier an der Adresse 0):

0	STR	1
1	f/2	
2	CON	a
3	CON	b

Eine ähnliche Optimierungsmöglichkeit durch eine spezielle Instruktion besteht auch bei der Unifikation in einem Klauselkopf mit einem Term, der Konstanten enthält. Angenommen, es soll mit dem Term `f(a,b)` unifiziert werden; bisher wurde derartiger WAM-Code erzeugt:

```
get_structure f/2, A1     % A1=f
unify_variable X2         %     (X2,
unify_variable X3         %         X3)
get_structure a/0, X2     % X2=a
get_structure b/0, X3     % X3=b
```

Mit einem neuen WAM-Befehl **unify_constant c** vereinfacht sich obige Code-Sequenz folgendermaßen:

```
get_structure f/2, A1     % A1=f
unify_constant a          %     (a,
unify_constant b          %        b)
```

Neben den neuen Befehlen **set_constant** und **unify_constant** werden noch zwei weitere WAM-Instruktionen benötigt, um Konstanten in Argumentregister zu schreiben bzw. um sie aus Argumentregistern zu laden. Es folgt eine kurze Beschreibung jener vier WAM-Befehle, die der Behandlung von Konstanten dienen:

put_constant c, A*i*: Dieser Befehl weist dem Argumentregister A*i* die Konstante c zu (mit der Markierung CON); es werden keine Heap-Speicherzellen verbraucht.

set_constant c: Mittels dieser Instruktion wird am Heap eine neue Speicherzelle angelegt, welche die Markierung CON erhält und die Konstante c aufnimmt.

get_constant c, A*i*: Falls das Argumentregister A*i* (dereferenziert, d.h. eventuell über eine Kette von beliebig vielen Variablenbindungen) auf eine ungebundene Variable verweist, wird die Heap-Speicherzelle dieser Variable mit der Konstante c überschrieben und die Markierung auf CON gesetzt. Andernfalls muß A*i* auf eine CON-Zelle verweisen, die bereits die Konstante c enthält, sonst scheitert die Unifikation, und Backtracking findet statt.

unify_constant c: Im Lese-Modus funktioniert dieser Befehl genauso wie die Instruktion **get_constant**, nur daß das WAM-Register S (structure pointer) die Rolle des Argumentregisters A*i* übernimmt (d.h. zunächst wird S dereferenziert). Weiters muß S nach gelungener Unifikation erhöht werden. In [Aït90] wird das WAM-Register S nicht inkrementiert, hierbei handelt es sich offensichtlich um einen Fehler. Im Schreib-Modus bewirkt **unify_constant** dasselbe wie **set_constant**, es wird dann also eine neue mit CON markierte Heap-Zelle für die Konstante c allokiert.

Die neue Markierung LIS muß natürlich auch vom Unifikationsalgorithmus (ein WAM-Unterprogramm), den **get_value** und **unify_value** aufrufen, berücksichtigt werden.

Listen Die Notation `[A|As]` beschreibt eine Liste, deren erstes Element die Variable A ist; As ist der Rest der Liste. `[A|As]` nennt man auch ein Listen-Paar. Intern wird diese Liste durch die Struktur `'.'(A,As)` dargestellt, der Funktor '.'/2 heißt auch Listen-Konstruktor. Die Konstante `[]` markiert das Listen-Ende. Beispielsweise entspricht der Liste `[a,b,c]` die Struktur `'.'(a,'.'(b,'.'(c,[])))`.

Da Listen von Prolog-Programmierern besonders oft verwendet werden, ist es sinnvoll, spezielle WAM-Instruktionen zur Behandlung von Listen einzuführen. Damit der Listen-Konstruktor '.'/2 nicht explizit als Funktor am Heap erzeugt werden muß, verwendet man die neue Markierung LIS für eine Speicherzelle, die auf ein Listen-Paar im Heap verweist. Der Unifikationsalgorithmus, der von **get_value** und von **unify_value** verwendet wird, muß angepaßt werden, um die Markierung LIS korrekt zu behandeln.

Folgende zwei neue WAM-Instruktionen werden zur effizienten Unterstützung von Listen benötigt:

put_list Xi: Dieser Befehl allokiert am Heap eine Speicherzelle, markiert diese mit LIS und weist ihr die Adresse der nächsten (noch unbenützten) Heap-Zelle zu. Die neu allokierte Heap-Speicherzelle wird in das Register Xi kopiert.

get_list Xi: Falls das dereferenzierte Register Xi auf eine ungebundene Variable verweist, allokiert diese Instruktion eine mit LIS markierte Heap-Zelle und weist ihr die Adresse der nächsten (noch freien) Heap-Speicherzelle zu. Die Variable wird an die neu allokierte Heap-Zelle gebunden. Dann wird der Schreib-Modus aktiviert. Falls Xi bereits auf eine LIS-Zelle verweist, wird das WAM-Register S (structure pointer) mit jener Adresse geladen, die in der LIS-Zelle gespeichert ist (das ist die Adresse der ersten Komponente des Listen-Paars). Sodann muß der Lese-Modus eingeschaltet werden. Zeigt Xi weder auf eine ungebundene Variable noch auf eine LIS-Zelle, scheitert **get_list**, und Backtracking findet statt.

Um die Liste `[a,b,c]` auf dem Heap zu erzeugen, wird folgender WAM-Code generiert:

```
put_list X3        % X3=[
set_constant c     %     c
set_constant []    %      ]
put_list X2        % X2=[
set_constant b     %     b|
set_value X3       %       X3]
put_list A1        % A1=[
set_constant a     %     a|
set_value X2       %       X2]
```

Die Heap-Repräsentation der Liste `[a,b,c]` sieht dann so aus, beginnend an der Heap-Adresse 6 (wenn man annimmt, daß H zuvor den Wert 0 enthalten hat):

0	LIS	1
1	CON	c
2	CON	[]
3	LIS	4
4	CON	b
5	LIS	1
6	LIS	7
7	CON	a
8	LIS	4

Folgender WAM-Code unifiziert in einem Klauselkopf mit der Liste `[a,b,c]`:

```
get_list A1          % A1=[
unify_constant a     %     a|
unify_variable X2    %       X2]
get_list X2          % X2=[
unify_constant b     %     b|
unify_variable X3    %       X3]
get_list X3          % X3=[
unify_constant c     %     c
unify_constant []    %      ]
```

Anonyme Variablen Für eine anonyme Variable, die nicht in Argument-Position, sondern in einer Struktur auftritt, muß kein WAM-Register Xi allokiert werden. Deshalb verwendet man zur Behandlung von anonymen Variablen in Strukturen spezielle WAM-Instruktionen:

set_void n**:** Dieser Befehl erzeugt n ungebundene Variablen auf dem Heap, er wird also eingesetzt, wenn in einer Struktur n anonyme Variablen in aufeinanderfolgenden Positionen vorkommen.

unify_void n**:** Im Schreib-Modus funktioniert diese Instruktion genauso wie der Befehl **set_void**, im Lese-Modus werden einfach die nächsten n Heap-Zellen übersprungen, beginnend an der Adresse, auf die das WAM-Register S (structure pointer) verweist, d.h. S wird um den Wert n erhöht.

Anonyme Variablen, die im Kopf einer Klausel in Argument-Position auftreten, können ignoriert werden, es müssen also keinerlei Get-Instruktionen für diese anonymen Variablen generiert werden. Tritt jedoch eine anonyme Variable in Argument-Position eines Ziels auf, muß das entsprechende Argumentregister mit einer ungebundenen Variable initialisiert werden, wofür die bereits vorgestellte Instruktion **put_variable** eingesetzt wird.

Im folgenden Beispiel soll WAM-Code für eine Regel erzeugt werden, die sowohl Konstanten, Listen, als auch anonyme Variablen enthält:

```
p(_, f(_,_,_,[a,g(b)],_)) :-
     q([_,_], a, _),
     r(f(_,_)).
```

```
p/2: allocate                  % p(_,
     get_structure f/5, A2     %     f
     unify_void 3              %      (_,_,_,
     unify_variable X4         %             X4,
     unify_void 1              %                _))
     get_list X4               % X4=[
     unify_constant a          %     a|
     unify_variable X5         %       X5]
     get_list X5               % X5=[
     unify_variable X6         %     X6
     unify_constant []         %       ]
     get_structure g/1, X6     % X6=g
     unify_constant b          %     (b)
                               % :-
     put_list X4               % X4=[
     set_void 1                %     _
     set_constant []           %      ]
     put_list A1               % q([
     set_void 1                %    _|
     set_value X4              %      X4],
     put_constant a, A2        %          a,
     put_variable A3, A3       %            _
     call q/3                  %             ),
     put_structure f/2, A1     % r(f
     set_void 2                %    (_,_)
     call r/1                  %         )
     deallocate                %          .
```

2.4.2 Registerallokation

Oft enthält der WAM-Code unnötige Datenbewegungen. Diese können manchmal mittels Guckloch-Optimierung (peep-hole optimization) entfernt werden, wobei immer ein kleiner Code-Ausschnitt betrachtet und lokal optimiert wird. Diese

Technik wird auch in Übersetzern für imperative Programmiersprachen auf den Zwischencode angewendet.

Man betrachte beispielsweise den WAM-Code für folgendes Fakt:

```
append([],L,L).
```

```
append/3: get_constant [], A1     % append([],
          get_variable X4, A2     %          L,
          get_value X4, A3        %            L)
          proceed                 %              .
```

Man erkennt leicht, daß das temporäre Register X4, welches als Zwischenspeicher für die Variable L dient, eingespart werden kann, denn A2 ist ja ebenfalls für die Variable L allokiert worden. Obiger Code ist also äquivalent zu folgendem:

```
append/3: get_constant [], A1     % append([],
          get_variable A2, A2     %          L,
          get_value A2, A3        %            L)
          proceed                 %              .
```

Da der Befehl **get_variable A2, A2** offensichtlich überflüssig ist (er weist dem Register A2 es selbst zu), läßt sich der Code folgendermaßen vereinfachen:

```
append/3: get_constant [], A1     % append([],
          get_value A2, A3        %          L,L)
          proceed                 %              .
```

Da Xi und Ai dieselben WAM-Register bezeichnen, sind die Instruktionen **get_variable Xi, Ai** und **put_value Xi, Ai** redundant, sie können mit Hilfe der Guckloch-Optimierung immer eliminiert werden. Um derartige Vereinfachungen zu ermöglichen, soll bei der Registerallokation versucht werden, Argumentregister für temporäre Variablen zu verwenden, soweit das die Korrektheit des generierten Codes nicht beeinträchtigt. Derartige Algorithmen werden in [Deb86] und [JDM88] vorgestellt, in [JDM88] wird auch versucht, durch Vertauschen von Unifikations-Instruktionen bessere Optimierungsmöglichkeiten zu schaffen.

In den folgenden Beispielen wird immer eine möglichst günstige Registerallokation gewählt, sofern darunter die Verständlichkeit des WAM-Codes nicht leidet.

2.4.3 Last Call Optimization

Bei endständig rekursiven Prozeduren (d.h. der letzte Befehl der Routine ist ein rekursiver Aufruf) einer beliebigen imperativen Programmiersprache kann der letzte rekursive Prozedur-Aufruf durch einen Sprungbefehl ersetzt werden, weil das aktuelle Environment (die lokalen Variablen der Prozedur) nicht mehr benötigt wird. Der rekursive Aufruf verwendet das alte Environment wieder, es wird Stack gespart, und die rekursive Prozedur verhält sich in Bezug auf Stack-Verbrauch wie eine äquivalente iterative Routine (der Stackbedarf ist also konstant). Diese Technik ist als Tail-Recursion Optimization (TRO) bekannt und läßt sich auch für deklarative Programmiersprachen, wie LISP (funktionale Programmiersprache) oder Prolog (logik-orientierte Programmiersprache) übertragen.

In der WAM ist eine Verallgemeinerung von TRO implementiert, die auch dann anwendbar ist, wenn das letzte Ziel einer Klausel kein rekursiver Aufruf ist; man spricht hier von Last-Call Optimization (LCO). Die Grundidee ist, daß permanente Variablen nicht mehr benötigt werden, nachdem die Put- (bzw. Set-) Instruktionen vor dem letzten Call-Befehl in einer Regel ausgeführt worden sind. Das aktuelle Environment darf bereits vor der letzten Call-Instruktion deallokiert werden, was durch einfaches Vertauschen des letzten Call-Befehls mit der Deallocate-Instruktion erreicht werden kann. Dabei muß allerdings beachtet werden, daß das WAM-Register CP (continuation program counter) schon durch **deallocate** richtig gesetzt wird, der letzte Call-Befehl darf daher CP nicht verändern. Deshalb wird eine neue WAM-Instruktion **execute** eingeführt:

execute p/n: Dieser Befehl lädt das WAM-Register P (program counter) mit der Anfangsadresse des WAM-Codes für das Prädikat p/n. Das Register CP wird nicht verändert.

Regeln der Gestalt '$kopf$`(...) :-` $ziel_1$`(...), ...,` $ziel_n$`(...).`' werden mittels LCO nach folgendem Muster in WAM-Code übersetzt:

allocate
Argumente von $kopf$ mittels Get-Instruktionen unifizieren
Argumente von $ziel_1$ mittels Put-Instruktionen schreiben
call $ziel_1$
...
Argumente von $ziel_n$ mittels Put-Instruktionen schreiben
deallocate
execute $ziel_n$

Falls sich das aktuelle Environment (und nicht der jüngste Wahlpunkt) an der Spitze des Stacks befindet (d.h. $E > B$), wird durch LCO der Speicherplatz des momentanen Environments beim letzen Aufruf (durch **execute**) entweder für einen Wahlpunkt (mittels **try_me_else**) oder für ein neues Environment (durch **allocate**) wiederverwendet werden. Dadurch können endständig rekursive (tail recursive) Prädikate wie iterative Schleifen mit konstantem Stackverbrauch exekutiert werden. Befindet sich jedoch ein Wahlpunkt an der Spitze des Stacks, hat LCO wegen der Environment-Protection keine sofortige Auswirkung auf den Stackverbrauch.

2.4.4 Kettenregeln

Unter einer Kettenregel versteht man eine Regel, die nur ein einziges Ziel hat; eine Kettenregel hat also die Gestalt '*kopf*`(...) :-` *ziel*`(...).`'. Unter Verwendung von LCO ergibt sich folgendes Muster für den WAM-Code einer derartigen Kettenregel:

allocate
Argumente von *kopf* mittels Get-Instruktionen unifizieren
Argumente von *ziel* mittels Put-Instruktionen schreiben
deallocate
execute *ziel*

Weil es in einer Kettenregel keine permanenten Variablen gibt (der Kopf und das erste Ziel einer Regel werden bei der Variablen-Klassifikation zusammen betrachtet), wird im Environment nur CP (continuation program counter) gespeichert (abgesehen von der Listen-Verkettung der Environments). Da jedoch vor der Deallocate-Instruktion keinerlei Call-Befehle ausgeführt werden, die CP verändern, ist es völlig überflüssig, ein Environment zu für die Kettenregel zu allokieren. Deshalb kann obige Code-Sequenz folgendermaßen optimiert werden:

Argumente von *kopf* mittels Get-Instruktionen unifizieren
Argumente von *ziel* mittels Put-Instruktionen schreiben
execute *ziel*

Als Beispiel soll WAM-Code für folgende Kettenregel generiert werden, man beachte dabei auch die geschickte Registerallokation (die Argumentregister A1 und A3 werden bereits durch Unify-Instruktionen für den rekursiven Aufruf neu geladen, die Variable Bs wird einfach im Argumentregister A2 „durchgereicht“):

```
append([A|As], Bs, [A|Cs]) :-
     append(As, Bs, Cs).
```

```
append: get_list A1          % append([
        unify_variable X4    %         A|
        unify_variable A1    %           As],Bs,
        get_list A3          %                  [
        unify_value X4       %                   A|
        unify_variable A3    %                     Cs]) :-
        execute append/3     %      append(As,Bs,Cs).
```

2.4.5 Environment Trimming

In diesem Abschnitt wird eine Verallgemeinerung von LCO vorgestellt, um den Stackverbrauch zu minimieren. Environment Trimming basiert auf der Idee, daß eine permanente Variable nur solange benötigt wird, bis die Argumente jenes Ziels, in dem die Variable zum letzten Mal auftritt, mittels Put-Instruktionen in die entsprechenden Argumentregister geladen worden sind. Vor dem Aufruf dieses Ziels darf der Stack-Speicherplatz, den die permanente Variable eingenommen hat, freigegeben werden. Dadurch kann das Environment einer Klausel bei jedem Ziel-Aufruf ein wenig „schrumpfen“, vor dem letzten Aufruf wird es durch LCO deallokiert. Um Environment Trimming zu ermöglichen, müssen die permanenten Variablen im Environment so angeordnet werden, daß sich eine permanente Variable umso näher an der Spitze des Stacks befindet, je früher sie freigegeben werden darf. D.h. Yi und Yj $(j > i)$ werden für die permanenten Variablen Variablen PV_i und PV_j allokiert, deren letzte Vorkommnisse in den Ziele $ziel_i$ und $ziel_j$ liegen, genau dann, wenn $ziel_i$ nach $ziel_j$ bewiesen wird oder $ziel_i$ mit $ziel_j$ identisch ist, denn unter der Annahme $j > i$ liegt Yj näher an der Spitze des Stacks (an der höheren Stack-Adresse) als Yi. Der Call-Instruktion wird neben der Adresse des aufgerufenen Prädikats auch die Anzahl der permanenten Variablen übergeben, die nach dem Aufruf noch benötigt werden. Mit Hilfe dieses Parameters (der in einem internen WAM-Register gespeichert wird) können jene Instruktionen, die Stack-Speicher allokieren (**allocate** und **try_me_else**), die momentane Spitze des Stacks berechnen. Wegen der Environment-Protection hat Environment Trimming, wie auch LCO, nur dann sofortige Auswirkung auf den Stackverbrauch, wenn sich das aktuelle Environment oberhalb des jüngsten Wahlpunkts befindet (d.h. wenn $E > B$ gilt).

Als Beispiel für die Code-Erzeugung mit Environment Trimming diene folgendes Prädikat p/2:

```
p(A,B) :-
     q(C,D,E),
     r(C,B,E),
     s(A,D,F),
     t(F).
```

Da alle 6 Variablen permanent sind, werden für sie die Register Y1,...,Y6 folgendermaßen allokiert, um Environment Trimming zu ermöglichen (diese Allokation is nicht eindeutig, es könnte z.B. auch die Variable C dem Register Y4 und B dem Register Y5 zugeordnet werden):

Variable	letztes Ziel	Register
A	s/3	Y2
B	r/3	Y4
C	r/3	Y5
D	s/3	Y3
E	r/3	Y6
F	t/1	Y1

Mit obiger Registerallokation wird folgender WAM-Code für das Prädikat p/2 generiert:

```
p/2: allocate                % p
     get_variable Y2, A1     %  (A,
     get_variable Y4, A2     %     B) :-
     put_variable Y5, A1     %            q(C,
     put_variable Y3, A2     %                D,
     put_variable Y6, A3     %                  E
     call q/3, 6             %                   ),
     put_value Y5, A1        %            r(C,
     put_value Y4, A2        %                B,
     put_value Y6, A3        %                  E
     call r/3, 3             %                   ),
     put_value Y2, A1        %            s(A,
     put_value Y3, A2        %                D,
     put_variable Y1, A3     %                  F
     call s/3, 1             %                   ),
     put_value Y1, A1        %            t(F
     deallocate              %               )
     execute t/1             %                .
```

Bei der Überprüfung des WAM-Codes beachte man, daß nach der Instruktion **call q/3, 6** die Register Y1,...,Y6 verwendet werden, nach dem WAM-Befehl **call r/3, 3** nur mehr die Register Y1,...,Y3 und schließlich nach der Instruktion **call s/3, 1** nur noch das Register Y1 vorkommt.

2.4.6 Stack Variablen

2.5 Indexing

2.6 Cut

2.7 Speicherbereiche

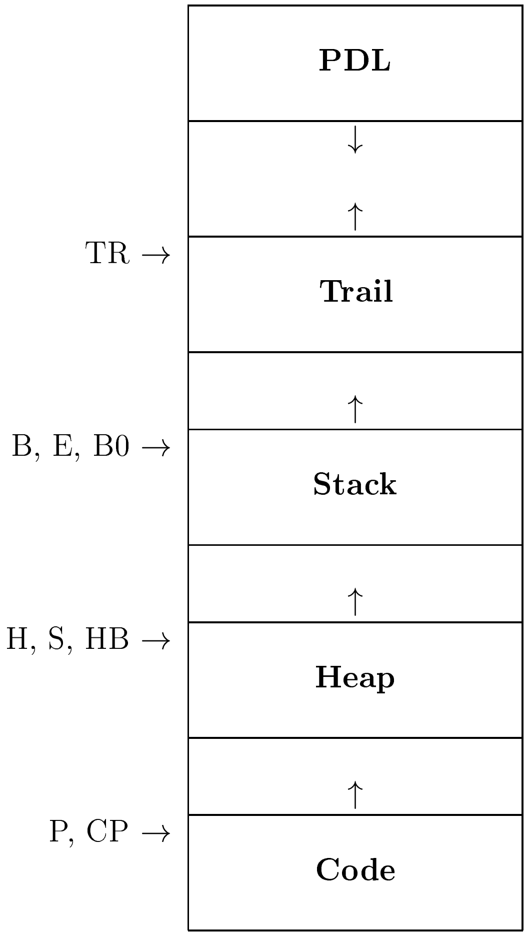
Hier werden alle Speicherbereiche der WAM kurz zusammengefaßt. Die Verwendung dieser Speicherregionen ist in den vorigen Abschnitten zu finden, wo auch auf das Format der gespeicherten Daten genau eingegangen worden ist.

Der Code-Bereich enthält die WAM-Instruktionen, der Heap dient der Erzeugung und Speicherung von Termen, der Stack wird sowohl für Environments (zur

Speicherung permanenter Variablen) als auch für die Oder-Kontrolle (Speicherung der Register, die beim Backtracking wiederhergestellt werden müssen) verwendet. Im Trail sind jene Variablenbindungen vermerkt, die beim Backtracking zurückzunehmen sind. Weiters gibt es noch den Unifikationsstack (Push-Down List, PDL).

Außerdem verwendet die WAM zahlreiche Register: P (program counter) und CP (continuation program counter) zeigen in den Code-Bereich, H (top of heap), HB sowie S (structure pointer) verweisen auf den Heap; B (most recent choice point), B0 und E (current environment pointer) addressieren den Stack, und TR (top of trail) markiert das Ende des Trails. Daneben gibt es noch eine ausreichend große Zahl an Argumentregistern (A1,A2,...,An,...) bzw. temporären Registern (X1,X2,...,Xn,...). Ai und Xi bezeichnen dabei dasselbe Register, je nach Verwendungsart des Registers wird die entsprechende Bezeichnung bevorzugt.

Die folgende Graphik veranschaulicht die Speicherregionen:



Literatur

[Aït90] Aït-Kaci, Hassan. *The WAM: A (Real) Tutorial.* Digital PRL, Report No. 5, Jänner 1990.

[BFS95] Binder, W, Faber, W und Schordan, M. *Binäres Prolog und EBC-Transformation. Seminararbeit. Wintersemester 1994/95.* Institut für Computersprachen, Technische Universität Wien, 1995.

[Deb86] Debray, Saumya K. Register allocation in a prolog machine. *Proceedings of the Symposium on Logic Programming (Salt Lake City, UT)*, IEEE Computer Society, September 1986.

[JDM88] Janssens, Gerda, Demoen, Bart und Mariën, Andre. Improving the register allocation in WAM by reordering unification. *Proceedings of the Fifth International Conference and Symposium on Logic Programming (Seattle, WA)*, MIT Press, 1988.

[Kra94] Krall, Andreas. Implementation Techniques for Prolog. *10. Workshop Logische Programmierung (Universität Zürich)*, 1994.

[War83] Warren, David H.D. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park, CA, Oktober 1983.