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 **4**

2 Warren Abstract Machine **6**

2.1 Unifikation 6
2.1.1 Termdarstellung 6
2.1.2 Ziele 7
2.1.3 Klauselköpfe 8
2.1.4 Argumentregister 10
2.2 Flat Resolution 12
2.2.1 Fakten 12
2.2.2 Regeln und Anfragen 13
2.3 Prolog 15
2.3.1 Environment Protection 16
2.3.2 Trail 19
2.3.3 Wahlpunkt 20
2.3.4 Backtracking 21
2.4 Optimierungen 22
2.4.1 Konstanten, Listen, anonyme Variablen 23
2.4.2 Registerallokation 28
2.4.3 Last Call Optimization 30
2.4.4 Kettenregeln 31
2.4.5 Environment Trimming 32
2.4.6 Stack Variablen 34
2.4.7 Klassifikation von Variablen 40
2.5 Indexing 41
2.6 Cut 47
2.7 Speicherbereiche 49

3 SICStus Prolog 51

3.1 Merkmale . . . 51

3.2 SICStus WAM . . . 52

3.2.1 Register . . . 52

3.2.2 Unifikation . . . 53

3.2.3 Environment . . . 58

3.2.4 Kontrollfluß . . . 58

3.2.5 Indexing . . . 59

3.2.6 Cut . . . 59

3.2.7 Blockierte Ziele . . . 60

3.2.8 Profiling . . . 60

4 SOURCE, SINK, VOID 61

4.1 SOURCE, SINK . . . 62

4.1.1 Syntax und Semantik der Befehle . . . 62

4.1.2 Beispiel . . . 63

4.1.3 Definite Clause Grammar . . . 69

4.2 VOID . . . 69

4.2.1 Syntax und Semantik des Befehls . . . 69

4.2.2 Implementierung des Befehls . . . 70

4.2.3 Beispiel . . . 72

5 Interprozedurale Registerallokation 76

5.1 Transformation . . . 76

5.2 Beispiele . . . 76

5.2.1 Verwandtschaftsbeziehungen . . . 76

5.2.2 Komplementäre DNA-Sequenzen . . . 79

5.2.3 Aufteilen einer Liste . . . 85

6 Benchmarks **91**

6.1 Benchmark-Programme 91

6.2 Meßverfahren 93

6.3 Performance-Gewinne durch Optimierungen 94

6.4 Overhead durch fixen Registersatz 96

6.5 Vergleich Performance-Gewinne — Overhead 101

7 Andere Optimierungsverfahren **104**

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 daraufWfolgende 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.

Sämtliche Unify-Instruktionen arbeiten im Schreib-Modus genauso wie die entsprechenden Set-Befehle. Wenn man also **put_structure** so modifiziert, daß der Schreib-Modus aktiviert wird, kann man anstatt der Set-Instruktionen Unify-Befehle verwenden. Obwohl in David H.D. Warrens ursprünglicher Beschreibung der WAM [War83] nur die Unify-Instruktionen dokumentiert sind, werden hier auch Set-Instruktionen verwendet, was vor allem der besseren Lesbarkeit des WAM-Codes dient. Außerdem ist es effizienter, zwischen Unify- und Set-Instruktionen zu unterscheiden, weil dadurch das Aktivieren des Schreib-Modus in **put_structure** entfällt. Darüber hinaus benötigen die Set-Befehle im Gegensatz zu den Unify-Instruktionen keine Abfrage, ob der Schreib- oder der Lese-Modus momentan aktiv ist.

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 Argument-Position 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 Argument-Position 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 X**i (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 $\mathrm{Y}i$ 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 X*i*: 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 X*i* kopiert.

get_list X*i*: Falls das dereferenzierte Register X*i* 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 X*i* 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 X*i* weder auf eine ungebundene Variable noch auf eine LIS-Zelle, scheitert **get_list**, und Backtracking findet statt.

Wie bereits weiter vorne beschrieben, ist es möglich, anstatt der Set-Befehle Unify-Instruktionen zu verwenden, sofern **put_structure** den Schreib-Modus aktiviert, da alle Unify-Befehle im Schreib-Modus genauso funtionieren wie die entsprechenden Set-Instruktionen. Soll auf die Set-Befehle verzichtet werden, muß auch die soeben beschriebene Wam-Instruktion **put_list** den Schreib-Modus einschalten.

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 X*i*, A*i*** und **put_value X*i*, A*i*** 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- oder 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 letzten 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 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_variable 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 PV_i und PV_j allokiert, deren letzte Vorkommnisse in den Zielen $ziel_i$ 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

Weil Heap-Speicher möglichst sparsam allokiert werden soll, wird in diesem Abschnitt gezeigt, daß in vielen Fällen für eine permanente Variable Yn, die das erste Mal in einem Ziel in Argument-Position auftritt (d.h. sie wird mittels **put_variable Yn, Ai** initialisiert), keine Heap-Zelle allokiert werden muß, denn Yn entspricht einer lokalen Variable, für welche Speicherplatz nur während der Exekution des Regelkörpers benötigt wird. Unter der Annahme, daß LCO und Environment Trimming nicht durchgeführt werden, ist die für Yn allokierte Speicherzelle im Environment ausreichend. Statt wie bisher am Heap eine ungebundene Variable zu erzeugen und die Stack-Zelle von Yn an diese Heap-Zelle zu binden, genügt es also, am Stack eine ungebundene Variable (eine Stack-Variable) zu erzeugen.

Die WAM-Instruktion **put_variable** ist für permanente Variablen folgendermaßen zu ändern:

put_variable Yn, Ai: Dieser Befehl initialisiert die Stack-Speicherzelle für Yn im aktuellen Environment (d.h. die Stack-Zelle mit der Adresse $E+n+1$) mit einer ungebundenen Variable (d.h. eine REF-Zelle, die auf sich selbst zeigt) und weist dem Argumentregister Ai diese Stack-Zelle zu; der Heap bleibt unverändert.

Diese Optimierung bringt jedoch einige Probleme mit sich, da Referenzen auf eine Stack-Variable ungültig werden, sobald das Environment für diese Stack-Variable deallokiert wird. Zunächst wird gezeigt, wie mit Hilfe einer einfachen Variablen-Bindungsregel die meisten Probleme behoben werden können, danach werden jene Fälle beschrieben, für die wegen LCO und Environment Trimming spezielle neue WAM-Instruktion eingeführt werden müssen, um sie korrekt zu behandeln.

Bindung von Variablen Weil es nun sowohl Heap- als auch Stack-Variablen gibt, sind folgende Bindungen zweier (zuvor ungebundener) Variablen denkbar:

Heap → Heap: Dieser Fall ist unproblematisch, da die Richtung der Bindung die Korrektheit nicht beeinflußt. Aus Effizienzgründen soll jedoch immer die „jüngere" Heap-Variable (jene an der höheren Heap-Adresse) an die „ältere" gebunden werden. Befinden sich beispielsweise die Variablen Var_i und Var_j an den Heap-Adressen i und j $(i < j)$ und gilt sowohl $i < HB$ als auch $j \geq HB$, so müßte die Bindung von Var_i and Var_j im Trail registriert und beim Backtracking zurückgenommen werden, während die Bindung von Var_j an Var_i nicht im Trail aufgenommen werden müßte, weil Var_j beim Backtracking automatisch durch das Zurücksetzen von H freigegeben wird.

Stack → Stack: Hier ist es für die Korrektheit unerläßlich, daß die „jüngere" Stack-Variable (jene an der höheren Stack-Adresse) an die "ältere" gebunden wird, weil die „jüngere" Stack-Variable keinesfalls noch länger existieren wird als die „ältere" und daher eine Bindung der „älteren" ungültig wird, sobald das Environment der „jüngeren" überschrieben worden ist. Die Regel, daß immer die Variable an der höheren Adresse gebunden wird, ist glücklicherweise mit der Variablen-Ordnung innerhalb eines Environments konsistent, um Environment-Trimming zu ermöglichen.

Heap → Stack: Diese Bindung darf niemals auftreten, da Heap-Variablen eine längere Lebensdauer als Stack-Variablen haben.

Stack → Heap: Dieser Fall stellt kein Problem dar. Um zu erreichen, daß immer die Stack-Variable an die Heap-Variable gebunden wird, soll sich der Stack im Speicher oberhalb des Heaps befinden, dann erzeugt die Regel, daß immer die Variable an der höheren Adresse gebunden wird, korrekte Bindungen, Heap → Stack Bindungen können nicht mehr entstehen.

Zusammenfassend müssen folgende drei Regeln eingehalten werden, um korrekte (und effiziente) Variablenbindungen zu erreichen:

1. Eine Variable an einer höheren Adresse darf nur auf eine Variable an einer niedereren Adresse verweisen (bzw. auf sich selbst, falls die Variable noch ungebunden ist).
2. Heap-Variablen dürfen niemals auf Stack-Variablen zeigen.
3. Der Stack muß sich im selben Adreßraum oberhalb des Heaps befinden.

Durch diese Regeln ist sichergestellt, daß Stack-Variablen nur am Anfang einer Referenz-Kette vorkommen. Leider verletzen die WAM-Instruktionen **put_value** und **set_value** (daher auch **unify_value** im Schreib-Modus) manchmals diese Regeln. Diese Fälle werden in den nächsten zwei Abschnitten genau untersucht und durch spezielle WAM-Befehle korrigiert.

Unsichere Variablen Jede Stack-Variable wird freigegeben, bevor das letzte Ziel aufgerufen wird, in dem diese Variable vorkommt (durch Environment-Trimming oder mittels LCO). Falls diese Variable noch ungebunden oder an eine anderere permanente Variable desselben Environments gebunden ist, die ebenfalls vor dem Aufruf freigegeben wird, bekommt das aufgerufene Ziel (in einem Argumentregister) eine Referenz auf eine freigegebene Variable, deren Speicherplatz möglicherweise schon überschrieben worden ist. Eine permanente Variable, die mittels **put_variable** initialisiert wird, nennt man aus diesem Grund auch eine unsichere Variable (unsafe variable).

Betrachte man beispielsweise den (nicht korrekten) WAM-Code für folgende Regel p/2:

```
p(A,C) :-
     q(A,B),
     r(B,C).


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

In diesem Beispiel sind die Variablen B (im Register Y2) und C (im Register Y1) permanent, die Variable A ist temporär, da sie nur im Regelkopf und im ersten Ziel vorkommt. Für die Variable A wird das Argumentregister A1 allokiert, es müssen weder Get- noch Put-Instruktionen für A generiert werden. Von den zwei permanenten Variablen ist nur B unsicher, denn B wird mittels **put_variable** erzeugt, während die Variable C wird durch **get_variable** initialisert wird. Angenommen, der Aufruf von q/2 läßt die permanente Variable B ungebunden, d.h. die Stack-Speicherzelle für Y2 enthält nach der Exekution von q/2 noch immer eine REF-Zelle, die auf sich selbst verweist. Nun wird mittels **put_value** diese ungebundene Stack-Variable in das Argumentregister A1 geladen, danach wird Y1 (die Variable C) in das Argumentregister A2 kopiert und das Environment für die Regel p/2 deallokiert. Jetzt wird das Prädikat r/2 aufgerufen, welches im Argumentregister A1 eine Referenz auf eine Stack-Speicherzelle übergeben bekommt, die nicht mehr der Variable B entspricht, sondern möglicherweise durch das neue Environment von r/2 oder durch einen Wahlpunkt überschrieben worden ist.

Es muß daher zur Laufzeit beim letzten Vorkommnis einer unsicheren Variable Yn in Argument-Position geprüft werden, ob die Referenz-Kette von Yn zu einer ungebundenen Stack-Variable im selben Environment führt. Nur in diesem Fall sind Maßnahmen nötig, um zu gewährleisten, daß das aufgerufene Ziel gültige Daten in den Argumentregistern erhält. Der Fall, daß Yn zuletzt in einer Struktur vorkommt, wird im nächsten Abschnitt behandelt. Für eine unsichere Variable Yn, die zuletzt in Argument-Position i auftritt, muß statt der WAM-Instruktion **put_value Yn, Ai** der neue WAM-Befehl **put_unsafe_value Yn, Ai** generiert werden:

put_unsafe_value Yn, Ai: Falls der dereferenzierte Wert von Yn eine ungebundene Stack-Variable im aktuellen Environment ist, wird eine neue ungebundene Heap-Variable erzeugt, an welche die ungebundene Stack-Variable gebunden wird. Die neue Heap-Variable wird auch in das Argumentregister Ai kopiert. Andernfalls wird dem Argumentregister Ai der dereferenzierte Wert von Yn zugewiesen, der Heap bleibt unverändert. In [Aït90] heißt es, daß sich **put_unsafe_value** in diesem Fall genauso wie **put_value** verhält, was nicht ganz korrekt ist, da **put_value Yn, Ai** den Wert von Yn nicht derefernziert, sondern unverändert dem Argumentregister Ai zuweist.

Der korrekte WAM-Code für obiges Beispiel muß folgendermaßen aussehen:

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

Stack Variablen in Strukturen Wenn das Argument von **set_value** oder von **unify_value** im Schreib-Modus auf eine Stack-Variable verweist, entsteht eine Referenz vom Heap in den Stack, wodurch die Variablen-Bindungsregel verletzt wird, die Referenzen vom Heap in den Stack verbietet. Dieses Problem wird an Hand des folgenden Beispiels erläutert:

```
p :-
     q(A),
     r(A).

q(A) :-
     s(f(A)).
```

Bisher ist für diese zwei Regeln folgender (nicht korrekter) WAM-Code generiert worden:

```
p/0: allocate                   % p :-
     put_variable Y1, A1        %      q(A
     call q/1, 1                %         ),
     put_unsafe_value Y1, A1    %      r(A
     deallocate                 %         )
     execute r/1                %          .

q/1: get_variable X2, A1        % q(A) :-
     put_structure f/1, A1      %         s(f
     set_value X2               %            (A)
     execute s/1                %               ).
```

Wird nun das Prädikat p/0 aufgerufen, wird für die permanente Variable A das Register Y1 allokiert und mittels **put_variable** initialisiert, A ist also eine unsichere Variable, welche keine Heap-Zelle benötigt. Vor dem Aufruf von q/1 verweist das Argumentregister A1 auf diese Stack-Variable. Zunächst kopiert q/1 die Stack-Variable in das Register X2, dann erzeugt q/1 die Struktur f/1 auf dem Heap, deren Argument der Wert von X2 ist. Somit entsteht eine verbotene Referenz vom Heap in den Stack.

Wie obiges Beispiel zeigt, tritt dieses Problem nicht nur bei permanenten Variablen auf, sondern auch bei temporären. Die Instruktion **set_value V**n, wobei Vn eine temporäre Variable Xn oder eine permanente Variable Yn bezeichnet, funktioniert nur dann sicher, wenn Vn mittels **set_variable V**n, durch **unify_variable V**n oder mittels **put_variable X**n**, A**i (falls Vn eine temporäre Variable Xn bezeichnet) initialisiert worden ist. Da **unify_value** im Schreib-Modus genauso wie **set_value** funktioniert, gilt dasselbe auch für **unify_value**.

Folgende zwei neue WAM-Instruktionen ersetzen die erste Verwendung von **set_value V**n bzw. von **unify_value V**n, wenn nicht garantiert werden kann, daß sich der dereferenzierte Wert von Vn im Heap befindet:

set_local_value Vn**:** Falls der dereferenzierte Wert von Vn im Heap liegt, wird eine neue REF-Zelle auf dem Heap allokiert und ihr der dereferenzierte Wert von Vn zugewiesen. Andernfalls verweist Vn auf eine Stack-Variable, die dann an eine ungebundene, neu auf dem Heap allokierte Variable gebunden wird. Es entsteht somit keine verbotene Heap → Stack Bindung.

unify_local_value Vn**:** Im Lese-Modus funktioniert dieser Befehl genauso wie **unify_value V**n, d.h. Vn wird mit dem Wert des WAM-Registers S (structure pointer) unifiziert. Im Schreib-Modus arbeitet **unify_local_value V**n genauso wie **set_local_value V**n.

Der korrekte WAM-Code für die Regel q/1 aus obigem Beispiel sieht folgendermaßen aus:

```
q/1: get_variable X2, A1       % q(A) :-
     put_structure f/1, A1     %         s(f
     set_local_value X2        %            (A)
     execute s/1               %               ).
```

Kommt eine unsichere Variable Yn (d.h. Yn ist mittels **put_variable** initialisiert worden) in einer Struktur vor, muß das letzte Auftreten der Instruktion **put_value Y**n nicht durch **put_unsafe_value Y**n ersetzt werden (falls Yn zuletzt in Argument-Position vorkommt), da durch **set_local_value** bzw. durch **unify_local_value** sichergestellt ist, daß Yn in den Heap verweist.

2.4.7 Klassifikation von Variablen

In [Aït90] werden temporäre Variablen folgendermaßen definiert:

> Eine temporäre Variable darf nur in einem Ziel eines Regelkörpers vorkommen, wobei der Regelkopf zusammen mit dem ersten Ziel betrachtet wird. Weiters muß sie zuerst im Regelkopf, in einer Struktur oder im letzten Ziel auftreten. Alle anderen Variablen sind permanent.

Die Idee hinter dieser Definition ist, daß temporäre Variablen möglichst vermieden werden sollen, da für sie Heap-Zellen allokiert werden. Heap-Speicher wird für eine Variable, falls notwendig, bei ihrem ersten Auftreten allokiert. Eine Variable, die zuerst im Regelkopf auftritt, wird entweder an bereits vorhandene Daten gebunden, oder die Speicherzelle für die Variable muß am Heap allokiert werden; es macht also keinen Sinn, eine Variable, die in einem Regelkopf das erste Mal vorkommt, als permanent zu klassifizieren. Ähnliches gilt für Variablen, die zuerst in einer Struktur auftreten, für sie ist die Allokation einer Heap-Zelle unbedingt erforderlich, weil der Funktor-Zelle am Heap die Zellen für die Argumente der Struktur folgen müssen. Variablen am Stack zu allokieren, die das erste Mal im letzten Ziel auftreten, ist ebenfalls unzweckmäßig, da sie sonst wegen LCO ohnehin „globalisiert" (d.h. durch **put_unsafe_value** am Heap erzeugt) werden würden.

In [Aït90] wird folgendes Beispiel für diese „optimierte" Klassifikation von Variablen gebracht:

```
a :-
     b(X,X),
     c.
```

```
a/0: allocate                   % a :-
     put_variable Y1, A1        %      b(X,
     put_unsafe_value Y1, A2    %         X
     call b/2, 0                %          ),
     deallocate                 %      c
     execute c/0                %       .
```

Die Variable X wird hier als permanent klassifiziert, für sie wird Y1 allokiert. Da Y1 mittels **put_variable** initialisiert wird, handelt es sich um eine unsichere

Variable, deren letztes Auftreten in Argument-Position (wegen Environment-Trimming) auf die Instruktion **put_unsafe_value** übersetzt wird. Da Y1 zu diesem Zeitpunkt noch eine ungebundene Stack-Variable ist (es ist ja noch kein Ziel aufgerufen worden), wird Y1 durch **put_unsafe_value** „globalisiert". Dieser WAM-Code ist also extrem ineffizient; obwohl zur Übersetzungszeit bekannt ist, daß für die Variable X eine Heap-Zelle erforderlich ist, wird dies erst zur Laufzeit durch **put_unsafe_value** festgestellt. Weiters wird eine Speicherzelle am Stack (für Y1) verschwendet.

Klassifiziert man X im obigen Beispiel hingegen wie bisher als temporär, erhält man für die Regel a/0 folgenden WAM-Code, der sowohl effizienter, kürzer, als auch besser verständlich ist:

```
a/0: allocate                % a :-
     put_variable A1, A2     %      b(X,X
     call b/2, 0             %           ),
     deallocate              %      c
     execute c/0             %       .
```

SICStus Prolog beispielsweise erzeugt ähnlichen WAM-Code, klassifiziert also die Variable X ebenfalls als temporär.

Hier soll die ursprüngliche Variablen-Klassifikation beibehalten werden:

> Eine Variable, die nur in einem Ziel einer Regel vorkommt (wobei der Regelkopf zusammen mit dem ersten Ziel betrachtet wird), heißt temporär. Alle anderen Variablen sind permanent.

2.5 Indexing

Normalerweise werden beim Backtracking alle möglichen Klauseln in der Reihenfolge probiert, in der sie im Prolog-Programm definiert sind. Falls alle Argumente eines Ziels freie Variablen sind, gibt es auch keine bessere Vorgangsweise. Wenn aber ein Argument bereits an einen Term gebunden ist, müssen jene Klauseln nicht versucht werden, die an der entsprechenden Stelle im Kopf einen Term stehen haben, der mit diesem Argument nicht unifizierbar ist. Dadurch erspart man sich oft viele Backtracking-Versuche, im Idealfall wird überhaupt kein Wahlpunkt erzeugt.

Die WAM verfügt über einige Befehle, die es ermöglichen, abhängig vom Wert des Argumentregisters A1 im WAM-Code an verschiedene Stellen zu verzweigen; diese Technik nennt man Indexing. Da ausschließlich das Argumentregister A1 für die Änderung des Kontrollflusses in Betracht gezogen wird, ermöglicht die WAM nur Indexing nach dem Hauptfunktor des ersten Arguments. Dies ist sinnvoll, da Prolog-Programmierer sehr häufig das erste Argument eines Prädikats zur Unterscheidung der Klauseln verwenden.

Bei folgendem Prädikat last/2, das v.a. zur Ermittlung des letzten Elements einer Liste eingesetzt wird, kann mit Hilfe von Indexing nach dem Hauptfunktor des ersten Arguments erreicht werden, daß überhaupt kein Wahlpunkt angelegt wird, falls beim Aufruf von last/2 das erste Argument keine Variable ist:

```
last([X|Xs], Last) :-
     last(Xs, Last, X).

last([], Last, Last).
last([X|Xs], Last, _X0) :-
     last(Xs, Last, X).
```

Die folgende, äquivalente Definition last2/2 hingegen erlaubt kein Indexing nach dem Hauptfunktor des ersten Arguments, weil beide Klauseln '.'/2 als Hauptfunktor des ersten Arguments haben:

```
last2([Last], Last).
last2([_X|Xs], Last) :-
     last2(Xs, Last).
```

Ein Prolog-Programmierstil, der die Generierung von Indexing-Code nach dem Hauptfunktor des ersten Arguments ermöglicht, wird in [OKe90] gelehrt, während in anderen Prolog-Lehrbüchern, wie beispielsweise in [SS86], kaum gezeigt wird, wie man saubere Prolog-Programme schreibt, welche auch auf effizienten WAM-Code übersetzt werden können.

Die folgenden drei neuen WAM-Instruktionen dienen dem Indexing:

switch_on_term V, C, L, S: Die Parameter V, C, L und S sind Markierungen (labels) von WAM-Instruktionen im Code-Bereich. Abhängig davon, ob der dereferenzierte Wert des Argumentregisters A1 eine Variable, eine Konstante, eine nicht leere Liste oder eine andere Struktur (keine Liste und keine Konstante) ist, wird zu der Instruktion mit der Markierung V, C, L bzw. S verzweigt.

switch_on_constant N, T: Der Parameter T dieses Befehls ist eine Hash-Tabelle der Größe N, deren Einträge Paare (c_i, L_{c_i}) $(1 \leq i \leq N)$ sind, wobei

c_i Konstanten und L_{c_i} Markierungen von Instruktionen darstellen. Dieser Befehl darf nur verwendet werden, wenn mittels **switch_on_term** zuvor sichergestellt worden ist, daß der dereferenzierte Wert des Argumentregisters A1 eine Konstante k ist. Kommt k in der Hash-Tabelle T vor, d.h. gilt $k = c_j$ $(1 \leq j \leq N)$, so wird zu dem Befehl mit der Markierung L_{c_j} gesprungen. Wird die Konstante k nicht in T gefunden, findet Backtracking statt.

switch_on_structure N, T: Wie bei der Instruktion **switch_on_constant** bezeichnet auch bei diesem WAM-Befehl der Parameter T eine Hash-Tabelle der Größe N, wobei die Einträge dieser Tabelle nun Funktoren (außer dem Listen-Konstruktor '.'/2) samt Arität ($\geq$ 1) und Markierungen im Code-Bereich der WAM sind. Vor Verwendung dieser WAM-Instruktion muß sichergestellt sein (durch **switch_on_term**), daß das Argumentregister A1 (dereferenziert) auf eine Struktur f/n verweist. Wird f/n in der Hash-Tabelle T gefunden, verzweigt der Kontrollfluß zu der f/n entsprechenden Markierung, andernfalls findet Backtracking statt.

Um Indexing-Code für ein Prädikat p zu generieren, das aus den Klauseln C_1,...,C_n $(n > 1)$ besteht, faßt man die Klauseln in Teilfolgen S_1,...,S_m $(1 \leq m \leq n)$ zusammen, sodaß S_i entweder aus einer einzigen Klausel besteht, deren erstes Argument eine Variable ist, oder aus möglichst vielen aufeinanderfolgenden Klauseln, deren erste Argumente keine Variablen sind. Klauseln, deren erstes Argument eine Variable ist, dürfen deshalb nicht mit anderen Klauseln zusammengefaßt werden, da sie auf jeden Fall (unabhängig vom dereferenzierten Wert des Argumentregisters A1) versucht werden müssen. Sofern $m > 1$ gilt, werden die Teilfolgen S_i mittels der WAM-Instruktionen **try_me_else**, **retry_me_else** (falls $m > 2$) und **trust_me** verkettet. Für Teilfolgen S_i, die nur aus einer Klausel bestehen, wird WAM-Code direkt erzeugt, sonst wird für S_i Indexing-Code generiert (es sei denn, alle Klauseln der Teilfolge S_i haben im ersten Argument denselben Hauptfunktor, dann ist durch Indexing kein Performace-Gewinn erzielbar).

Bei folgendem Prädikat p/1 erhält man durch Zusammenfassen von Klauseln C_i $(1 \leq i \leq 10)$ die Teilfolgen $S_1 = (C_1, C_2, C_3, C_4, C_5)$, $S_2 = (C_6)$, $S_3 = (C_7)$

und $S_4 = (C_8, C_9, C_{10})$.

```
p(a).                  % C1
p(f(_X)).              % C2
p([]).                 % C3
p([a]).                % C4
p(b).                  % C5
p(_X).                 % C6
p(X) :-                % C7
     q(X).
p(g(_X,_Y)).           % C8
p(h(_X,_Y,_Z)).        % C9
p(g(X,Y)) :-           % C10
     r(X,Y).
```

WAM-Code für das Prädikat p/1 wird folgendermaßen generiert:

```
p/1: try_me_else S2
     Indexing-Code für S1
S2:  retry_me_else S3
     proceed                   % p(_X).
S3:  retry_me_else S4
     execute q/1               % p(X) :- q(X).
S4:  trust_me
     Indexing-Code für S4
```

Indexing-Code für die Teilfolge S_1 sieht dann so aus:

```
      switch_on_term S11, C1, S1d, S1b
 C1:  switch_on_constant 3, {a: S1a,
                             b: S1e,
                             []: S1c}
 S11: try_me_else S12
 S1a: get_constant a, A1        % p(a
      proceed                   %    ).
 S12: retry_me_else S13
 S1b: get_structure f/1, A1     % p(f
      unify_void 1              %    (_X)
      proceed                   %        ).
 S13: retry_me_else S14
 S1c: get_constant [], A1       % p([]
      proceed                   %     ).
 S14: retry_me_else S15
 S1d: get_list A1               % p([
      unify_constant a          %    a
      unify_constant []         %     ]
      proceed                   %      ).
 S15: trust_me
 S1e: get_constant b, A1        % p(b
      proceed                   %    ).
```

Kommen in einer Teilfolge S_i mehrere Klauseln mit demselben Hauptfunktor im ersten Argument vor (wie beispielsweise in S_4 die beiden Klauseln C_8 und C_{10} mit dem Hauptfunktor g/2), werden drei neue WAM-Instruktionen benötigt, um die Verkettung dieser Klauseln zu realisieren: Die WAM-Befehle **try L**, **retry L** und **trust L** funktionieren fast genauso wie **try_me_else L**, **retry_me_else L** bzw. **trust_me**, jedoch sichern sie die Adresse des nächsten WAM-Befehls im Wahlpunkt (außer **trust**, denn diese Instruktion gibt den Wahlpunkt frei) und springen zu der Adresse L im Code-Bereich.

Mit Hilfe dieser neuen WAM-Befehle kann Indexing-Code für die Teilfolge S_4

aus obigem Beispiel folgendermaßen erzeugt werden:

```
      switch_on_term S41, fail, fail, F4
F4:   switch_on_structure 2, {g/2: F41,
                               h/3: S4b}
F41:  try S4a
      trust S4c
S41:  try_me_else S42
S4a   get_structure g/2, A1     % p(g
      unify_void 2              %    (_X,_Y)
      proceed                   %           ).
S42:  retry_me_else S43
S4b:  get_structure h/3, A1     % p(h
      unify_void 3              %    (_X,_Y,_Z)
      proceed                   %              ).
S43:  trust_me
S4c:  get_structure g/2, A1     % p(g
      unify_variable A1         %    (X,
      unify_variable A2         %       Y)) :-
      execute r/2               %              r(X,Y).
```

Man beachte, daß der Gesamt-Code für das Prädikat p/1 mit Indexing deutlich ineffizienter ist als ohne Indexing, falls beim Aufruf von p/1 der dereferenzierte Wert des Argumentregisters A1 eine Variable ist, denn in diesem Fall müssen alle Klauseln probiert werden. Die für jede Teilfolge S_i exekutierten Befehle **switch_on_term** stellen dann einen Overhead dar, weiters werden mehr Wahlpunkte am Stack angelegt (einer für die Verkettung der Teilfolgen S_i und einer in jeder Teilfolge, die aus mehreren Klauseln besteht, zur Verkettung der Klauseln). Im Fall, daß es in einer Teilfolge S_i mehrere Klauseln mit demselben Hauptfunktor im ersten Argument gibt, muß in jedem Fall (unabhängig davon, ob das Argumentregister A1 auf eine Variable verweist) ein weiterer Wahlpunkt für diese Teilfolge (mittels **try**) angelegt werden, was den Performance-Gewinn durch Indexing möglicherweise zunichte macht.

Abschließend soll noch der Indexing-Code für das Prädikat last/2 vom Anfang dieses Abschnitts gezeigt werden (man beachte dabei die geschickte Registerallokation, es werden keinerlei Put-Instruktionen benötigt):

```
last/2: get_list A1            % last([
        unify_variable A3      %       X|
        unify_variable A1      %         Xs],Last) :-
        execute last/3         %      last(Xs,Last,X).

last/3: switch_on_term L1, La, Lb, fail

L1:     try_me_else L2
La:     get_constant [], A1    % last([],
        get_value A2, A3       %         Last,Last
        proceed                %                  ).
L2:     trust_me
Lb:     get_list A1            % last([
        unify_variable A3      %       X|
        unify_variable A1      %         Xs],Last,_X0) :-
        execute last/3         %      last(Xs,Last,X).
```

2.6 Cut

In diesem Abschnitt wird die WAM erweitert, um das Cut zu unterstützen. Wenn das Cut ausgeführt wird, müssen alle Wahlpunkte vom Stack gelöscht werden, die erzeugt worden sind, seit das Prädikat aufgerufen worden ist, welches das Cut enthält. Falls beim Aufruf dieses Prädikats ebenfalls ein Wahlpunkt generiert worden ist, muß auch dieser vom Stack entfernt werden.

Man unterscheidet zwei Arten von Cut:

- Ein Cut, der vor dem ersten Ziel einer Klausel steht, heißt Neck-Cut. Die Klausel hat also entweder die Gestalt '*kopf* `:- !.`' oder die Form '*kopf* `:- !,` $ziel_1$`, ...,` $ziel_n$`.`', wobei $n \geq 1$ gilt.
- Kommen vor einem Cut ein oder mehrere Ziele, spricht man von einem tiefen Cut. Eine Klausel, die ein tiefes Cut enthält, hat folgende Gestalt: '*kopf* `:-` $ziel_1$`, ...,` $ziel_i$`, !, ...,` $ziel_n$`.`' ($1 \leq i \leq n$)

Um die Implementierung des Cuts zu ermöglichen, verwendet die WAM ein weiteres Register B0, welches durch die WAM-Instruktionen **call** und **execute** auf den Wert des WAM-Registers B (most recent choice point) gesetzt wird.

Wird eine Klausel exekutiert, die ein Neck-Cut enthält, gilt vor der Ausführung des Cuts entweder $B0 = B$ (falls wegen Indexing kein Wahlpunkt generiert worden ist) oder B0 zeigt auf den Wahlpunkt vor B ($B > B0$). Im Fall $B0 = B$ hat das Cut überhaupt keine Wirkung, da die Klausel ohnehin keine Alternativen bietet. Andernfalls gilt $B > B0$, und es muß der letzte Wahlpunkt, auf den B verweist, freigegeben werden. Dabei wird auch HB auf den Wert aus dem Wahlpunkt zurückgesetzt, wodurch es möglich wird, daß einige Variablen-Bindungen im Trail gelöscht werden dürfen (Heap-Variablen, deren Adressen nun größer als der neue Wert von HB sind, bzw. Stack-Variablen, die sich an Adressen größer als der neue Wert von B befinden). Beim Entfernen von Variablen-Bindungen aus dem Trail können „Löcher" im Trail entstehen, die durch „Zusammenschieben" der im Trail verbleibenden Bindungen entfernt werden müssen. Was soeben beschrieben worden ist, sind genau jene Aktionen, welche die neue WAM-Instruktion **neck_cut** durchführt.

WAM-Code für die Klauseln '`p :- !.`' und '`q :- !, r.`' sieht folgendermaßen aus (das Neck-Cut ist natürlich nur dann sinnvoll, wenn es mehrere Klauseln für p/0 bzw. für q/0 gibt):

```
p/0: neck_cut        % p :- !
     proceed         %       .

q/0: neck_cut        % q :- !,
     execute r/0     %        r.
```

Eine Klausel, die nur ein Cut ausführt, wird wie ein Fakt kompiliert, eine Regel, die zuerst ein Cut und dann ein einziges Ziel exekutiert, wird wie eine Kettenregel übersetzt.

Im Fall eines tiefen Cuts ist das WAM-Register B0 durch aufgerufene Ziele möglicherweise (falls die aufgerufenen Ziele Wahlpunkte erzeugt haben) überschrieben worden, weshalb der Wert von B0 bei der Allokation des Environments genauso wie das WAM-Register CP (continuation program counter) im Environment gesichert werden muß. Dies ist allerdings nur für Regeln notwendig, die ein tiefes Cut enthalten, weshalb es eine Verschwendung von Stack-Speicher (und von Rechenzeit) wäre, B0 genauso wie CP immer im Environment zu sichern. Deshalb wird ein tiefes Cut mit einer permanenten Variable Yn verbunden, die sonst nicht benötigt wird. Folgende zwei WAM-Befehle ermöglichen die Übersetzung von tiefen Cuts nach WAM-Code:

get_level Yn: Diese Instruktion muß unmittelbar auf den Befehl **allocate** folgen, sie speichert den Wert des WAM-Registers B0 im neuen Environment an der Stelle für Yn.

cut Y*n*: Diese Instruktion gibt alle Wahlpunkte frei, die nach jenem erzeugt worden sind, auf welchen Y*n* verweist. Ähnlich wie der Befehl **neck_cut** entfernt auch **cut** Bindungen aus dem Trail, die nach der Freigabe der Wahlpunkte nicht weiter im Trail benötigt werden, wobei enstehende „Löcher" im Trail durch Verschieben der verbleibenden Bindungen geschlossen werden müssen.

Die Verwendung der beiden WAM-Instruktionen **get_level** und **cut** wird nun anhand der Klausel '`p :- q, !.`' gezeigt, die ohne das Cut eine Kettenregel wäre:

```
p/0: allocate         % p
     get_level Y1     %   :-
     call q/0, 1      %      q,
     cut Y1           %        !
     deallocate       %
     proceed          %         .
```

Anstatt des WAM-Befehls **proceed** erzeugen einige Compiler, wie beispielsweise SICStus Prolog, die Instruktion **execute true/0**.

Obiges Beispiel zeigt deutlich, daß tiefe Cuts zu wesentlich ineffizienterem Code führen als Neck-Cuts.

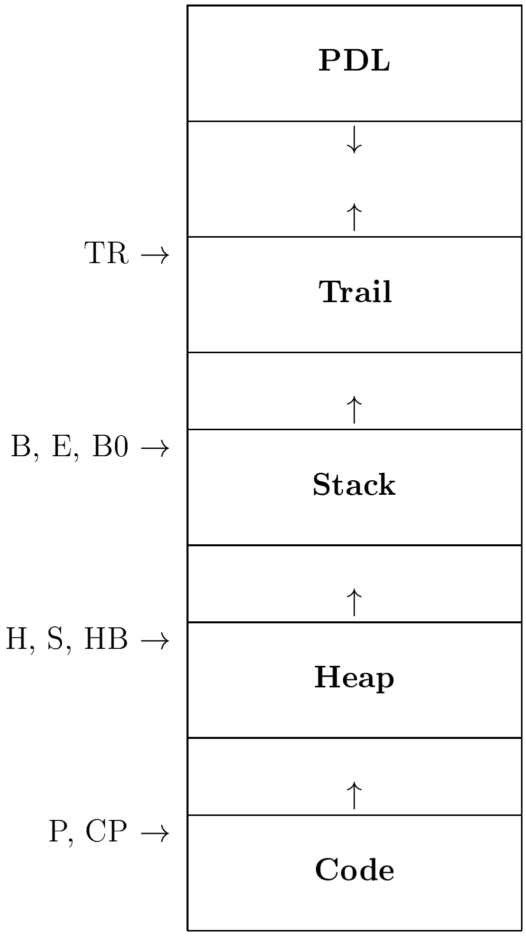
2.7 Speicherbereiche

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 dort 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:



3 SICStus Prolog

Da SICStus Prolog auf Universitäten sehr verbreitet ist und mit dem vollständigen Quell-Code (in C und in Prolog) ausgeliefert wird, eignet sich dieses Prolog-System besonders dafür, Erweiterungen einzubauen. Aus diesem Grund wird die interprozedurale Registerallokation für eine WAM mit fixem Registersatz, wie sie in den folgenden Abschnitten beschrieben wird, in SICStus Prolog integriert und getestet.

In diesem Abschnitt werden zunächst die Leistungsmerkmale von SICStus Prolog genannt, danach wird die SICStus WAM vorgestellt, wobei v.a. auf die Unterschiede zur WAM, wie sie im vorigen Abschnitt erklärt ist, eingegangen wird.

3.1 Merkmale

SICStus Prolog wird am Swedish Institute of Computer Science (SICS) seit 1986 entwickelt. Es bietet neben dem „Edinburg Standard" (der z.B. in [CM87] beschrieben ist) viele Erweiterungen, wie beispielsweise Coroutining (blockierte Ziele), Integerarithmetik mit beliebig großen Zahlen (indefinite precision integer arithmetic), eine Schnittstelle zur Programmiersprache C und seit kurzem einen integrierten Constraint Solver für endliche Bereiche (Finite Domains, CLP(N)). Weiters entspricht SICStus Prolog (abgesehen vom Modul-System) bereits dem neuen ISO-Standard, der in [CDE96] dokumentiert ist. SICStus Prolog ist portabel, die aktuelle Version 3 ist für Sun-Workstations, DEC-Workstations (auch für die DEC Alpha Architektur), Silicon Graphics Rechner, IBM RS/6000, HP9000, HPPA, m68k, Sequent Symmetry und Personal Computer mit den Intel Prozessoren ab 80386 erhältlich. Für einige Architekturen (Sun SparcCenter2000, Silicon Graphics, Sequent Symmetry) wird auch eine oder-parallele Version von SICStus Prolog (Muse) angeboten. In [BK96] wird SICStus Prolog (auf einem PC) mit anderen Prolog-Kompilern verglichen und erweist sich als eines der schnellsten derzeit erhältlichen Prolog-Systeme.

SICStus Prolog kann Prädikate sowohl interpretieren (was sehr langsam ist, aber für dynamische Prädikate, deren Klauseln mittels assert/1 und retract/1 zur Laufzeit geändert werden, benötigt wird) als auch kompilieren. Im Fall der Übersetzung von Prolog-Programmen wird Code für eine Variante der WAM, die SICStus WAM, erzeugt. Dieser WAM-Code wird dann üblicherweise interpretiert, für einige Hardware-Architekturen, wie z.B. für Sun- und DEC-Workstations (außer DEC Alpha) sowie für m68k, kann der WAM-Code auch auf Maschinencode weiterübersetzt werden (native code), wodurch meist eine Geschwindigkeitssteigerung bei der Programmausführung um den Faktor 2 bis 3 erreicht wird (die Übersetzung dauert dann allerdings doppelt bis dreimal so lange und der Code wird etwas länger).

Aus einem Prolog-Programm wird mittels pl_wam WAM-Code erzeugt. Soll Native Code für eine bestimmte Hardware-Architektur generiert werden, erzeugt zunächst wam_sam Code für die SICStus Abstract Machine (SAM), die eine General-Purpose Load/Store-Architektur mit beliebig vielen Registern darstellt. Für m68k wird aus dem SAM-Code mittels sam_68k und 68k_bin direkt Maschinencode generiert, für RISC-Prozessoren wird der SAM-Code zuerst durch sam_riss auf RISS-Code (RIScified Sam) übersetzt. RISS ist eine abstrakte Maschine, die eine Verallgemeinerung von RISC-Prozessoren darstellt. Mittels riss_sparc und sparc_bin wird Maschinencode für Sun-Workstations erzeugt, mit Hilfe von riss_mips und mips_bin generiert man Maschinencode für DEC-Workstations mit MIPS-Prozessoren. Die Native-Code-Übersetzung von SICStus Prolog wird genau in [Hay94] beschrieben.

3.2 SICStus WAM

In diesem Abschnitt werden die Instruktionen jener Variante der WAM, beschrieben, die in SICStus Prolog implementiert ist (SICStus WAM). Dabei wird v.a. auf Unterschiede zu der WAM eingegangen, wie sie bisher vorgestellt worden ist. Als Grundlage für diesen Abschnitt dienen sowohl der in Prolog geschriebene WAM-Code Generator von SICStus Prolog (pl_wam) als auch der Quellcode des WAM-Code Interpreters, welcher in der Programmiersprache C geschrieben ist, sowie zahlreiche Experimente mit dem WAM-Code Generator pl_wam (Blackbox Tests).

3.2.1 Register

Sowohl die temporären Register Xi (bzw. Argumentregister Ai) als auch die permanenten Variablen Yi werden nur durch ihre Nummer i repräsentiert. Ob das Argument i eines SICStus WAM-Befehls nun das Register Xi (Ai) oder Yi bezeichnet, geht aus dem Namen der Instruktion selbst hervor, es gibt getrennte Befehle, die entweder auf Registern Xi (beispielsweise **unify_x_value**) operieren oder auf Variablen Yi (z.B. **unify_y_value**).

Neben den temporären Registern Xi und den permanenten Variablen Yi kennt die SICStus WAM auch temporäre Register Ti, welche den Registern Xi entsprechen und nur zur Unifikation in „tiefen" Termen (z.B. in verschachtelten Strukturen) eingesetzt werden, wie beispielsweise in der Instruktion **unify_temp_value**.

Weiters beginnt die Numerierung der Register nicht wie bisher mit 1, sondern bei 0. Die Argumentregister 0 bis $n-1$ (bisher A1,...,An) enthalten beim Aufruf eines Prädikats der Arität n somit die Argumente 1 bis n, die dem Prädikat mittels Put-Instruktionen übergeben werden. Die permanenten Variablen werden ebenfalls mit 0 beginnend durchnumeriert.

Das Argumentregister 0 (bisher A1 genannt) wird von einigen Get-Befehlen als Optimierung (wegen Indexing nach dem Hauptfunktor des ersten Arguments) gesondert behandelt; es gibt dann spezielle Get-Instruktionen, wie beispielsweise **get_structure_x0**, denen die Registernummer nicht übergeben werden muß.

3.2.2 Unifikation

In diesem Abschnitt werden sämtliche Instruktionen der SICStus WAM vorgestellt, die der Unifikation dienen. Zu beachten ist, daß die SICStus WAM keine Set-Instruktionen kennt, statt ihnen werden Unify-Befehle verwendet, die, wie bereits weiter vorne angemerkt, im Schreib-Modus dasselbe bewirken wie entsprechende Set-Instruktionen.

Variablen

Put-Instruktionen

put_x_variable(n**,**i**):** Funktioniert analog zu **put_variable X**n'**, A**i'. Dabei gelten $n' = n + 1$ und $i' = i + 1$, da die Numerierung der Register der SICStus WAM bei 0 beginnt.

put_y_variable(n**,**i**):** Dieser Befehl darf nur vor dem ersten Aufruf in einer Klausel vorkommen, sonst ist die Instruktion **put_y_first_value(**n**,**i**)** zu verwenden. Entspricht **put_variable Y**n'**, A**i'.

put_y_first_value(n**,**i**):** Ähnlich wie **put_y_variable(**n**,**i**)**, jedoch ist dieser Befehl nach dem ersten Aufruf in einer Klausel zu verwenden.

put_x_value(n**,**i**):** Entspricht **put_value X**n'**, A**i'.

put_y_value(n**,**i**):** Funktioniert wie **put_value Y**n'**, A**i'.

put_x_unsafe_value(n**,**i**):** Dieser Befehl wird benötigt, falls das temporäre Register n einen unsicheren Wert enthält. Bei der WAM, wie sie bisher vorgestellt worden ist, war dies nie möglich, weshalb es auch keine Entsprechung für diese Instruktion gibt.

put_y_unsafe_value(n**,**i**):** Funktioniert analog zu **put_unsafe_value Y**n'**, A**i'.

Get-Instruktionen

get_x_variable(n,i): Entspricht **get_variable Xn', Ai'**.

get_y_variable(n,i): Funktioniert analog zu **get_variable Yn', Ai'**. Dieser Befehl darf nur vor dem ersten Aufruf in einer Klausel vorkommen, sonst ist stattdessen die Instruktion **get_y_first_value(n,i)** zu verwenden.

get_y_first_value(n,i): Ähnlich wie **get_y_variable(n,i)**, allerdings darf dieser Befehl nur nach dem ersten Aufruf in einer Klausel eingesetzt werden.

get_x_value(n,i): Funktioniert wie **get_value Xn', Ai'**.

get_y_value(n,i): Entspricht **get_value Yn', Ai'**.

Unify-Instruktionen

unify_x_variable(n): Entspricht **unify_variable Xn'**.

unify_y_variable(n): Dieser Befehl funktioniert analog zu **unify_variable Yn'**, darf aber nur vor dem ersten Aufruf in einer Klausel eingesetzt werden, sonst ist die Instruktion **unify_y_first_value(n)** zu verwenden.

unify_y_first_value(n): Ähnlich wie **unify_y_variable(n)**, allerdings darf dieser Befehl nur nach dem ersten Aufruf in einer Klausel generiert werden.

unify_x_value(n): Entspricht **unify_value Xn'**.

unify_y_value(n): Funktioniert genauso wie **unify_value Yn'**.

unify_x_local_value(n): Wie **unify_local_value Xn'**.

unify_y_local_value(n): Entspricht **unify_local_value Yn'**.

unify_temp_variable(n): Diese Instruktion entspricht **unify_variable Xn'**, sie wird jedoch nur zur Unifikation in „tiefen“ Termen (z.B. in verschachtelten Strukturen) verwendet.

unify_temp_value(n): Ähnlich wie **unify_value Xn'**, jedoch nur zur Unifikation in „tiefen“ Termen gedacht.

Konstanten Als Konstanten zählen für SICStus Prolog nicht nur Atome, sondern auch Zahlen bis zu einer gewissen Größe. Die SICStus WAM unterstützt auch beliebig große Zahlen mittels spezieller Instruktionen (siehe später).

Put-Instruktionen

put_constant(c,i): Funktioniert analog zu **put_constant c, A**i'.

Get-Instruktionen

get_constant(c,i): Entspricht **get_constant c, A**i'. Dieser Befehl wird nur für $i > 0$ (entsprechend den Argumentregistern A2,A3,...) verwendet, für $i = 0$ ist die Instruktion **get_constant_x0(c)** zu verwenden.

get_constant_x0(c): Funktioniert analog zu **get_constant c, A0**. Dieser Befehl wird zur Implementierung des Indexings nach dem Hauptfunktor des ersten Arguments benötigt.

Unify-Instruktionen

unify_constant(c): Entspricht **unify_constant c**, allerdings übernimmt diese Instruktion auch die Aufgabe von **set_constant c**.

Strukturen

Put-Instruktionen

put_structure(f,i): Entspricht **put_structure f, A**i'.

put_temp_structure(f,n): Wie **put_structure f, X**n', wird aber nur für verschachtelte Subterme verwendet.

Get-Instruktionen

get_structure(f,i): Funktioniert analog zu **get_structure f, A**i', wird jedoch ausschließlich für $i > 0$ (entsprechend den Argumentregistern A2,A3,...) verwendet, für $i = 0$ ist stattdessen der Befehl **get_structure_x0(f)** einzusetzen.

get_structure_x0(f): Entspricht **get_structure f, A0**, diese Instruktion wird wegen des Indexings nach dem Hauptfunktor des ersten Arguments benötigt.

get_temp_structure(f,n): Entspricht **get_structure f, X**n' und wird nur zur Unifikation von verschachtelten Subtermen benötigt.

Unify-Instruktionen

unify_structure(f): Dieser Befehl wird nicht unbedingt benötigt, er könnte durch eine Kombination der Instruktionen **unify_temp_variable(**n**)** und **get_structure(f,**n**)** ersetzt werden, allerdings werden durch die Instruktion **unify_structure(f)** ein Befehl und ein Register eingespart, nach Möglichkeit wird daher als Optimierung diese Instruktion generiert.

Listen

Put-Instruktionen

put_nil(i**):** Als Optimierung wird die leere Liste nicht mittels der Instruktionen für Konstanten behandelt. Dieser Befehl funktioniert ähnlich wie die Instruktion **put_constant** [], **A**i'.

put_list(i**):** Entspricht **put_list A**i'.

put_temp_list(n**):** Ähnlich wie **put_list X**n', diese Instruktion wird nur für verschachtelte Subterme gebraucht.

Get-Instruktionen

get_nil(i**):** Entspricht **get_constant** [], **A**i'. Dieser Befehl wird nur für $i > 0$ verwendet.

get_nil_x0: Ähnlich wie **get_constant** [], **A0**.

get_list(i**):** Entspricht **get_list A**i', darf nur für $i > 0$ eingesetzt werden.

get_list_x0: Entspricht **get_list A0**.

get_temp_list(n**):** Ähnlich wie **get_list X**n', wird nur für „tiefe“ Listen verwendet.

Unify-Instruktionen

unify_nil: Entspricht **unify_constant** [].

unify_list: Dieser Befehl ist nicht unbedingt erforderlich (ähnlich wie bei der Instruktion **unify_structure**), er ermöglicht aber die Generierung eines effizienteren Codes.

Große ganze Zahlen Zahlen ab einer bestimmten Größe können nicht mehr mittels der Instruktionen für Konstanten behandelt werden, SICStus Prolog bietet für große Zahlen eine eigene Heap-Repäsentation.

Put-Instruktionen

put_large(num,i): Wird für große Werte von num eingesetzt, ähnlich wie der Befehl **put_constant num, A**i'.

put_temp_large(num,n): Wird nur in verschachtelten Subtermen verwendet, ähnlich wie **put_constant num, X**n'.

Get-Instruktionen

get_large(num,i): Wie **get_constant num, A**i', nur für $i > 0$.

get_large_x0(num): Ähnlich wie **get_constant num, A0**.

get_temp_large(num,n): Wie **get_constant num, X**n', jedoch wird diese Instruktion nur für verschachtelte Subterme eingesetzt.

Unify-Instruktionen

unify_large(num): Dieser Instruktion ist nicht unbedingt erforderlich, es handelt sich bei diesem Befehl um eine Optimierung, ähnlich wie bei der Instruktion **unify_structure**.

Anonyme Variablen

Put-Instruktionen

put_x_void(i): Dieser Befehl wird als Optimierung an Stelle der Instruktion **put_x_variable**(i,i) generiert.

Unify-Instruktionen

unify_void: Entspricht **unify_void 1**.

3.2.3 Environment

Die folgenden Instruktionen werden für die Verwaltung der Environments benötigt:

allocate: Wie bisher erzeugt dieser Befehl ein neues Environment.

deallocate: Zur Freigabe des aktuellen Environments.

init(List): Wird vor der ersten call-Instruktion generiert.

3.2.4 Kontrollfluß

Die folgenden SICStus WAM-Befehle dienen dem Aufruf von Prädikaten sowie von eingebauten Funktionen, wie beispielsweise Arithmetik.

call(L,N): Dieser Befehl funktioniert analog zu **call L, N**, wobei L das Label der ersten Instruktion des WAM-Codes eines Prädikats ist und N die Anzahl der permanenten Variablen bezeichnet, die nach dem Aufruf noch benötigt werden.

execute(L): Entspricht **execute L**.

proceed: Das ist die letzte Instruktion im WAM-Code von Fakten (wie bisher), aber in manchen Fällen auch von Regeln, falls nur eingebaute Funktionen, wie z.B. `is/2` verwendet werden.

fail: Die Ausführung dieser Instruktion bewirkt Backtracking.

builtin(Goal,Else): Wird für die Kompilation von speziellen Prolog Funktionen benötigt, wie beispielsweise für die metalogischen Prädikate `var/1` und `nonvar/1`, oder für `functor/3`. Das Argument Goal dieser Instruktion beschreibt die auszuführende Operation, das zweite Argument Else kann `fail` heißen, falls Backtracking erwünscht ist, sofern Goal scheitert, oder `else` lauten, wenn statt Backtracking zu einer Adresse gesprungen werden soll, die im Code der Klausel gespeichert ist.

builtin_imm(Goal,Else): Funktioniert ähnlich wie **builtin(Goal,Else)**, wird z.B. für die Übersetzung von Vergleichsoperationen verwendet.

function(Expr,Value,N,EffAr): Diese Instruktion wird u.a. zur Kompilation von `arg/3`, `compare/3`, `if/3` und von arithmetischen Ausdrücken (`is/2`) eingesetzt. Expr bezeichnet die Operation, welche durch **function** exekutiert werden soll, Value bestimmt das Register, in dem das Ergebnis abgespeichert wird. Das Argument N beschreibt den Speicherplatz, der am Heap gebraucht wird, der Parameter EffAr bezeichnet die Anzahl der X-Register, deren Inhalt noch benötigt wird.

function_imm(Expr,Value,N,EffAr): Diese Instruktion wird, ähnlich wie der Befehl **function(Expr,Value,N,EffAr)**, zur Übersetzung von speziellen Operationen verwendet.

3.2.5 Indexing

SICStus Prolog speichert die Indexing-Information für jede Klausel im WAM-Code ab. So wird beispielsweise mittels einer Bit-Maske codiert, ob das erste Argument einer Klausel eine Nummer, ein Atom, eine Liste oder eine Struktur sein kann. Auf diese Weise werden auch metalogische Prädikate, wie beispielsweise `var/1` oder `nonvar/1`, direkt in die Indexing-Information übersetzt, sofern sie als erstes Ziel einer Klausel Einschränkungen für das erste Argument der Klausel festlegen. Informationen über Cuts in der Klausel werden ebenfalls in der Indexing-Information abgelegt. Der Hauptfunktor des ersten Arguments einer Klausel (sofern das erste Argument keine Variable ist) wird gesondert abgespeichert. Weiters wird die WAM-Instruktion **switch_on_term** zur Implementierung des Indexings benötigt.

3.2.6 Cut

In diesem Unterabschnitt werden jene SICStus WAM-Instruktionen vorgestellt, mit deren Hilfe das Cut implementiert wird. Zu beachten ist, daß der WAM-Code Generator von SICStus Prolog (pl_wam) nicht für alle Cuts im Prolog-Programm Cut-Befehle generiert, oft werden Neck-Cuts direkt in der Indexing-Information codiert, die für jede Klausel gespeichert wird.

choice_x(n)**:** Ähnlich wie **choice_y**(n) (siehe weiter unten), jedoch wird die Adresse des letzten Wahlpunkts im X-Register n gespeichert. Das Cut kann dann mittels **cutb_x**(n) ausgeführt werden.

choice_y(n)**:** Entspricht **get_level Y**n'.

cutb: Wie **neck_cut**.

cutb_x(n)**:** Verwirft alle Wahlpunkte bis zu jenem, auf den das X-Register n verweist.

cut_y(n)**:** Entspricht **cut Y**n'.

3.2.7 Blockierte Ziele

SICStus Prolog unterstützt Coroutining, d.h. der Programmierer kann Bedingungen für die Argumente eines Prädikats festlegen, die erfüllt sein müssen, damit der Code für dieses Prädikat exekutiert wird. So kann der Programmierer beispielsweise verlangen, daß das erste Argument eines Prädikats keine Variable sein soll. Wird diese Bedingung verletzt, führt das allerdings nicht zum Scheitern des Prädikats (wie bei der Verwendung von `nonvar/1`), sondern das Ziel, welches dieses Prädikat aufrufen wollte, wird solange verzögert, bis die Bedingung erfüllt ist.

Die folgende SICStus WAM-Instruktion ermöglicht Coroutining:

heapmargin_call(N,EffAr): Dieser Befehl prüft, ob es blockierte Ziele gibt, die „aufzuwecken" sind. Weiters erkennt diese Instruktion Speicherüberläufe am Heap und kann die Speicherbereinigung (Garbage Collection) auslösen. Das Argument N beschreibt den Speicherplatz, der am Heap gebraucht wird, der Parameter EffAr bezeichnet die Anzahl der X-Register, deren Inhalt noch benötigt wird.

Blockierte Ziele können auch durch die WAM-Instruktionen **call** und **execute** „aufgeweckt" werden, denn auch bei diesen Befehlen ist bekannt, welche Argumentregister noch benötigt werden (entsprechend den Argumenten des aufgerufenen Prädikats).

3.2.8 Profiling

Die SICStus WAM-Befehle **profile_point**, **bump_entry** und **bump_counter** werden vom WAM-Code Generator pl_wam nur dann erzeugt, wenn Profiling-Code generiert werden soll (für Performance-Messungen). Da die Profiling-Befehle den WAM-Code verändern, werden Benchmarks in den folgenden Abschnitten mittels geeigneter Stopp-Prädikate ohne Profiling durchgeführt.

4 SOURCE, SINK, VOID

Wie bereits im ersten Abschnitt erwähnt, kennt die WAM nur Caller Saved Register, d.h. beim Aufruf des WAM-Codes eines Prädikats muß der Aufrufer annehmen, daß der Inhalt sämtlicher Argumentregister während der Exekution des aufgerufenen Prädikats verloren geht. Werden Variablen für mehrere Ziele benötigt, so müssen sie im Environment gespeichert und bei Bedarf in die entsprechenden Argumentregister geladen werden.

Das Ziel dieser Diplomarbeit ist, eine interprozedurale Registerallokation für die WAM zu erreichen, sodaß unnötige Datenbewegungen vermieden werden. Dabei wird auch der Einsatz von Stack-Variablen minimiert, was ebenfalls zu einer Performance-Steigerung beiträgt, da den Stack-Variablen Speicherplätze in Environments auf dem Stack entsprechen, während Argumentregister im Idealfall direkt auf Maschinenregister des verwendeten Prozessors abgebildet werden können (im Fall von SICStus Prolog handelt es sich bei den Argumentregistern allerdings um Software-Register, d.h. diese Register befinden sind im Hauptspeicher, ein Zugriff auf ein Argumentregister ist deshalb kaum schneller als auf eine Stack-Variable).

Die neue interprozedurale Registerallokation soll möglichst lokal arbeiten, Prädikate sollen weiterhin völlig unabhängig voneinander übersetzt werden, um komplizierte und zeitaufwendige globale Datenfluß-Analysen zu vermeiden. Mittels einer fixen Aufteilung der Argumentregister in Caller Saved und Callee Saved Register und einer entsprechenden Allokation dieser Register (eventuell mittels einfacher Heuristiken) kann dieses Ziel erreicht werden.

Weiters soll die interprozedurale Registerallokation derartig in SICStus Prolog eingebaut werden, daß möglichst wenig Änderungen im Prolog-System vorgenommen werden müssen, vor allem darf die Variante der WAM, die in SICStus Prolog implementiert ist, nicht modifiziert werden, da sonst sowohl der WAM-Code Interpreter als auch der SAM-Code Generator wam_sam, der für die Native Code Kompilation benötigt wird, geändert werden müßten. Im Idealfall soll die neue Registerallokation durch eine Transformation von Prolog-Programmen realisiert werden. Das ist leider nicht direkt möglich, da SICStus Prolog über keinerlei Low-Level Befehle verfügt, mit deren Hilfe man die Registerallokation des WAM-Code Generators pl_wam beeinflussen könnte.

Daher muß eine Schnittstelle entwickelt werden, mit der man in einem Prolog-Programm jede beliebige Registerallokation beschreiben kann. Vor allem muß es möglich sein, auf Prolog-Ebene zu ereichen, daß manche Argumentregister wie Callee Saved Register verwendet werden. Diese Schnittstelle soll aus wenigen Low-Level Befehlen bestehen, die sich möglichst einfach im WAM-Code Generator pl_wam integrieren lassen. Die neuen Befehle beeinflussen die Registerallokation, aber es werden keinerlei neue SICStus WAM-Instruktionen benötigt, und die Semantik sämtlicher WAM-Befehle bleibt unverändert.

4.1 SOURCE, SINK

4.1.1 Syntax und Semantik der Befehle

In source stellt U. Neumerkel zwei neue Low-Level Instruktionen für Prolog vor (`'SOURCE'/n` und `'SINK'/n`), mit deren Hilfe er die EBC-Transformation, eine Programm-Transformationstechnik, welche er ursprünglich für Binäres Prolog entwickelt hat, auch für WAM-basierte Prolog-Systeme nutzbar macht. Dabei werden Prolog-Programme zunächst auf einen Zwischencode (Continuation Prolog) transformiert. Dieser Zwischencode entspricht binärem Prolog, d.h. jede Klausel ist eine Kettenregel, sie hat genau ein Ziel; ein zusätzliches Argument wird für die Continuation eingeführt. Auf die Zwischendarstellung Continuation Prolog wird die EBC-Transformation angewendet, es werden alternative Repräsentationen für die Continuations mittels Gleichungen definiert und in den Continuation Prolog-Code eingebaut. Durch diese Gleichungen werden oft Redundanzen sichtbar gemacht, der Code kann vereinfacht werden. Dieser optimierte Continuation Prolog-Code kann direkt auf der Binären WAM (BinWAM), einer Spezialisierung der WAM für Binäres Prolog, ausgeführt werden, oder mittels der neuen Low-Level Befehle `'SOURCE'/n` und `'SINK'/n` auf Low-Level Prolog-Code zurücktransformiert werden, der sich anschließend auf effizienteren WAM-Code übersetzen läßt, als das ursprüngliche Prolog-Programm. U. Neumerkel implementierte diese zwei neuen Low-Level Instruktionen im WAM-Code Generator pl_wam von SICStus Prolog, Benchmarks in [Neu95] zeigen, daß sich mit Hilfe der EBC-Transformation Geschwindigkeitssteigerungen bis zum Faktor 2.4 erzielen lassen (gemessen mit SICStus Prolog auf einem PC mit dem i486 Prozessor). In [Neu95] wird weiters gezeigt, wie man Definite Clause Grammars (DCGs), welche von Prolog-Programmierern sehr häufig verwendet werden, direkt, ohne den Umweg über Continuation Prolog und die EBC-Transformation, auf Low-Level Prolog übersetzen kann.

Die Semantik der beiden Low-Level Instruktionen `'SOURCE'/n` und `'SINK'/n` wird nun genau dargestellt, auf Implementierungsdetails wird hier allerdings nicht eingegangen. Die Modifikationen, die für die Implementierung dieser neuen Befehle notwendig waren, betreffen jedenfalls nur einen kleinen Teil des WAM-Code Generators pl_wam von SICStus Prolog.

Beide Low-Level Instruktionen können mit beliebig vielen Argumenten aufgerufen werden. Als Argument kann entweder `'-'` oder `'+'(X)` stehen, wobei mittels `'-'` angezeigt wird, daß das entsprechende Argument undefiniert ist, während `'+'(X)` bedeutet, daß das Argument der Wert von X sein soll.

'SOURCE'/n: Diese Instruktion sichert zu, daß für alle ihre Argumente `'+'(X)` jene Argumentregister verwendet werden, die ihren Argumentpositionen entsprechen. Der Low-Level Befehl `'SOURCE'/n` darf nur unmittelbar nach einem „gewöhnlichen" Prolog-Ziel (also nach einer call-Instruktion) verwendet werden. Der WAM-Code, welcher für `'SOURCE'/n` erzeugt wird, ähnelt jenem, der die Unifikation mit Argumenten aus Klauselköpfen (mittels Get-Instruktionen) durchführt.

Bei der Implementierung von `'SOUCE'/n` war eine komplexe Registerallokation für Klauselkörper nötig.

'SINK'/n: Dieser Low-Level Befehl sorgt dafür, daß sich die Argumente `'+'(X)` in den ihren Positionen entsprechenden Argumentregistern befinden. Die Instruktion `'SINK'/n` darf nur als letztes Ziel einer Klausel verwendet werden. Der WAM-Code, welcher für `'SINK'/n` generiert wird, ähnelt sehr jenem, der für „gewöhnliche" Ziele mittels Last Call Optimization (LCO) erzeugt wird.

Kommt `'SINK'/n` in einer Klausel als einziges Ziel vor, wird die WAM-Instruktion **proceed** generiert, andernfalls wird **execute('SINK'/n)** erzeugt (die Fakten `''SINK'(_).'`, `''SINK'(_,_).'`, usw. müssen definiert werden). Dieser Aufruf eines Prädikats mit der Arität n ist wichtig, da blockierte Ziele bei der WAM-Instruktion **execute** möglicherweise „aufgeweckt" werden und dabei nur auf die ersten n Argumentregister, die für die Exekution des augferufenen Prädikats erforderlich sind, Rücksicht genommen wird. Aus diesem Grund darf nicht **execute(true/0)** generiert werden, wie es der WAM-Code Generator pl_wam sonst macht, wenn das letzte Ziel einer Klausel ein Builtin-Prädikat ist.

4.1.2 Beispiel

Das folgende Beispiel soll die korrekte Verwendung der beiden Low-Level Befehle `'SOURCE'/n` und `'SINK'/n` demonstrieren. Das Prädikat `sum/2` wird verwendet, um die Summe eines beliebig geklammerten Ausdrucks der Gestalt $z_0+z_1+...+z_n$

($n \geq 0$) zu ermitteln, wobei alle z_i ($0 \leq i \leq n$) Zahlen sind:

```
sum(N,N) :-
     number(N).
sum(A+B,S) :-
     sum(A,SA),
     sum(B,SB),
     S is SA+SB.
```

Der WAM-Code Generator pl_wam von SICStus Prolog erzeugt für obiges Prädikat `sum/2` folgenden Code:

```
get_x_value(0,1)            % sum(N,N) :-
heapmargin_call(128,1)      %
builtin(number(0),fail)     %      number(N)
proceed                     %                .

get_structure_x0((+)/2)     % sum(A+B,
allocate                    %
get_y_variable(0,1)         %         S) :-
unify_x_variable(0)         %
unify_y_variable(3)         %
put_y_variable(1,1)         %
init([2])                   %
call(sum/2,4)               %      sum(A,SA),
put_y_value(3,0)            %
put_y_first_value(2,1)      %
call(sum/2,3)               %      sum(B,SB),
put_y_value(1,0)            %
put_y_value(2,1)            %
heapmargin_call(5,2)        %
function(0+1,0,0,2)         %      S is SA+SB
get_y_value(0,0)            %
deallocate                  %
execute(true/0)             %                 .
```

Der WAM-Code für die erste Regel von `sum/2` ist 4 Instruktionen lang, jener für die zweite mißt 18 Befehle und verwendet 4 Stack-Variablen. Wird die Summe von $n+1$ Zahlen $z_0 + z_1 + ... + z_n$ ($n \geq 0$) gebildet, wobei die Klammerung keine Rolle spielt, wird die erste Regel von `sum/2` (die Abbruchbedingung der Rekursion) genau $n+1$ Mal exekutiert und die rekursive Regel n Mal aufgerufen (für jedes Zeichen „+“ im Summenausdruck). Insgesamt müssen daher

$4(n+1)+18n=22n+4$ SICStus WAM-Befehle ausgeführt werden, wenn angenommen wird, daß durch den Indexing-Mechanismus von SICStus Prolog immer die passende Klausel gewählt wird. In Summe werden $4n$ Stack-Variablen angelegt.

Mit Hilfe von `'SOURCE'/n` und `'SINK'/n` läßt sich das Prädikat `sum/2` folgendermaßen umschreiben:

```
sum(Ns,S) :-
     sum(Ns),
     'SOURCE'(+S).

sum(N) :-
     number(N),
     'SINK'(+N).
sum(A+B) :-
     sum(A),
     'SOURCE'(+SA),
     sum(B),
     'SOURCE'(+SB),
     S is SA+SB,
     'SINK'(+S).
```

Das Prädikat `sum/2` ist ein sogenanntes Hüll-Prädikat, welches nach außen hin die gleiche Schnittstelle bietet wie die nicht optimierte Version. Nach Möglichkeit soll in Prolog-Programmen direkt `sum/1` verwendet werden, um den Overhead durch die Schnittstelle zu vermeiden.

Die von U. Neumerkel (siehe [Neu95]) modifizierte Version des WAM-Code Generators pl_wam erzeugt aus obigem Low-Level Prolog-Programm folgenden

Code:

```
allocate                     % sum(Ns,
get_y_variable(0,1)          %        S) :-
init([])                     %
call(sum/1,1)                %     sum(Ns),
get_y_value(0,0)             %     'SOURCE'(+S)
deallocate                   %
execute(true/0)              %                 .

                             % sum(N) :-
builtin(number(0),fail)      %     number(N),
proceed                      %     'SINK'(+N).

get_structure_x0((+)/2)      % sum(A+B) :-
unify_x_variable(0)          %
allocate                     %
unify_y_variable(1)          %
init([0])                    %
call(sum/1,2)                %     sum(A),
get_y_first_value(0,0)       %     'SOURCE'(+SA),
put_y_value(1,0)             %
call(sum/1,1)                %     sum(B),
put_y_value(0,4)             %
put_x_value(0,5)             %     'SOURCE'(+SB),
heapmargin_call(5,6)         %
function(4+5,4,0,6)          %     S is SA+SB,
put_x_value(4,0)             %     'SINK'(+S)
deallocate                   %
execute('SINK'/1)            %               .
```

Nun besteht der Code für die erste Regel von `sum/1` nur mehr aus 2 Instruktionen (statt 4 Befehlen für das nicht optimierte Prädikat), für die zweite Regel sind 16 Instruktionen erzeugt worden (gegenüber 18 für die nicht optimierte Version), wobei nur noch 2 Stack-Variablen erforderlich sind (anstatt 4). Allerdings muß das Hüll-Prädikat `sum/2` genau 1 Mal aufgerufen werden, dessen WAM-Code aus 6 Instruktionen besteht und 1 Stack-Variable benötigt. Da für einen längeren Ausdruck die erste Regel von `sum/1` oft aufgerufen wird (für jede Zahl im Ausdruck 1 Mal), müssen insgesamt wesentlich weniger WAM-Befehle exekutiert werden als im nicht optimierten Beispiel. Zudem werden deutlich weniger Stack-Variablen benötigt: Wird die Summe der $n+1$ Zahlen $z_0 + z_1 + ... + z_n$ ($n \geq 0$) mittels obiger optimierter Version von `sum/2` gebildet, müssen insgesamt

$2(n+1)+16n+6 = 18n+8$ SICStus WAM-Instruktionen exekutiert werden (gegenüber $22n+4$ Befehlen für das nicht optimierte Prädikat). Für $n = 0$ benötigt daher die optimierte Version (unter Verwendung des Hüll-Prädikats) mit 8 Instruktionen genau doppelt soviele Befehle wie die ursprüngliche Version, für $n = 1$ exekutieren beide Versionen 26 WAM-Instruktionen, für $n \geq 2$ ist die optimierte Version deutlich schneller (um den Faktor $\frac{22n+4}{18n+8}$), für große Werte von n wird in etwa eine Beschleunigung um den Faktor $22/18 \approx 1.22$ erreicht, ein Speedup um über 20% ist also möglich (diese Berechnungen gelten nur unter der Annahme, daß alle SICStus WAM-Instruktionen in etwa gleich viel Zeit benötigen). Benchmarks mit SICStus Prolog zeigen, daß in der Realität tatsächlich Speedups um 20% bis 25% erzielt werden. Eine exakte Messung der Maschinenbefehle[1], die auf einem Intel Pentium Prozessor bei der Berechnung der Summe der Zahlen von 1 bis 20 (ohne Klammerung) ausgeführt werden, ergibt einen Speedup um den Faktor 1.242 (d.h. um 24.2%). Außerdem werden für die optimierte Version nur noch $2n+1$ Stack-Variablen allokiert (anstatt $4n$).

Allerdings ist obiger optimierter WAM-Code nicht optimal, in der zweiten Regel von `sum/1` wird die Summe aus SA und SB (mittels **function**) zuerst im X-Register 4 abgespeichert und danach sofort in das X-Register 0 kopiert, obwohl **function** die Summe direkt im X-Register 0 ablegen könnte. Die von U. Neumerkel implementierte Erweiterung des WAM-Code Generators pl_wam (siehe [Neu95]) verfügt über einen experimentellen, „aggressiv“ optimierenden Modus, der eine WAM-Instruktion weniger erzeugt, allerdings wird im Fall dieser Klausel unkorrekter WAM-Code generiert.

Eine weitere Optimierung ist möglich, wenn man die Summe im zweiten Argument berechnet (das zweite Argument spielt dabei die Rolle eines „Akkumulators“):

```
sum(Ns,S) :-
    sum2(Ns,0),
    'SOURCE'(-,+S).

sum2(N,S0) :-
    number(N),
    S is S0+N,
    'SINK'(-,+S).
sum2(A+B,S0) :-
    sum2(A,S0),
    'SOURCE'(-,+S1),
    sum2(B,S1).
```

[1] Dieses Meßverfahren wird später noch genau beschrieben.

Der WAM-Code, den SICStus Prolog für diese Version von `sum/2` generiert, sieht folgendermaßen aus:

```
allocate                     % sum(Ns,
get_y_variable(0,1)          %        S) :-
put_constant(0,1)            %
init([])                     %
call(sum2/2,1)               %      sum2(Ns,0),
get_y_value(0,1)             %      'SOURCE'(-,+S)
deallocate                   %
execute(true/0)              %                     .

                             % sum2(N,S0) :-
builtin(number(0),fail)      %      number(N),
function(1+0,2,0,2)          %      S is S0+N,
put_x_value(2,1)             %      'SINK'(-,+S)
proceed                      %                   .

get_structure_x0((+)/2)      % sum2(A+B,S0) :-
unify_x_variable(0)          %
allocate                     %
unify_y_variable(0)          %
init([])                     %
call(sum2/2,1)               %      sum2(A,S0),
put_y_value(0,0)             %
deallocate                   %      'SOURCE'(-,+S1),
execute(sum2/2)              %      sum2(B,S1).
```

Die erste Regel von `sum2/2` exekutiert nun 4 WAM-Befehle, die rekursive Regel führt nur mehr 9 Instruktionen aus und allokiert dabei eine Stack-Variable. Das Hüll-Prädikat benötigt in dieser optimierten Version allerdings 8 Befehle und ebenfalls eine Stack-Variable. Für die Berechnung der Summe der $n + 1$ Zahlen $z_0 + z_1 + ... + z_n$ $(n \geq 0)$ werden jetzt $4(n + 1) + 9n + 8 = 13n + 12$ SICStus WAM-Instruktionen exekutiert. Für $n = 0$ benötigt diese Version mit 12 ausgeführten Befehlen 3 Mal so lange wie das ursprüngliche, nicht optimierte Prädikat, für $n \geq 1$ ist diese Version aber die schnellste, es wird eine Beschleunigung um den Faktor $\frac{22n+4}{13n+12}$ gegenüber der nicht optimierten Version erreicht. Für große Werte von n ist somit ein Speedup von zirka 69% (Faktor 22/13) erzielbar. Ein Speedup von 66% bis 72% läßt sich auch durch Benchmarks mit SICStus Prolog nachweisen. Eine genaue Messung der Maschineninstruktionen, die auf einem Intel Pentium Prozessor bei der Berechnung der Summe der Zahlen von 1 bis 20 (ohne Klammerung) ausgeführt werden, liefert einen Speedup um den Faktor 1.515 (d.h. um 51.5%). Außerdem werden jetzt nur noch $n+1$ Stack-Variablen allokiert.

Der WAM-Code für die erste, nicht rekursive Regel von `sum2/2` ist allerdings suboptimal, denn der WAM-Befehl **put_x_value(2,1)** könnte eingespart werden, wenn **function** für das Ergebnis das X-Register 1 statt des Registers 2 allokierte.

4.1.3 Definite Clause Grammar

Im folgenden wird, ähnlich wie in [Neu95], kurz erklärt, wie DCGs (Definite Clause Grammars, siehe z.B. [SS86] oder [OKe90]) direkt auf Low-Level Prolog-Programme abgebildet werden können. Dabei wird für ein Nonterminalsymbol der Arität n nicht wie in der bekannten DCG-Transformation ein Prädikat der Arität $n+2$ erzeugt, sondern eines mit der Stelligkeit $n+1$.

Hat das Nonterminal die Stelligkeit 0, wird für die Eingabe-Liste das erste Argument verwendet. Falls das Nonterminalsymbol eine Arität ≥ 1 aufweist, darf die Eingabe-Liste nicht auf das erste Argument abgebildet werden, weil sonst das Indexing nach dem Hauptfunktor des ersten Arguments keine Performance-Gewinne mehr brächte. Stattdessen wird in diesem Fall die Eingabe-Liste im zweiten Argument allokiert. Für die Ausgabe-Liste wird in jedem Fall das zweite Argument verwendet.

4.2 VOID

4.2.1 Syntax und Semantik des Befehls

Transformiert man Prolog-Programme derart, daß alle Prädikate dieselbe Arität haben, um eine fixe Aufteilung der Argumentregister in Caller Saved und Callee Saved Register zu ermöglichen, kommt es vor, daß beim Aufruf eines Prädikats die Argumente an manchen Positionen keine Rolle spielen, weil das aufgerufene Prädikat diese Argumente ohnehin ignoriert (in den Köpfen aller Klauseln dieses Prädikats stehen dann anonyme Variablen). Es liegt nahe, beim Aufruf des Prädikats an den entsprechenden Argument-Stellen ebenfalls anonyme Variablen zu verwenden, allerdings erzeugt SICStus Prolog in diesen Fällen die WAM-Instruktion **put_x_void**, obwohl der Inhalt der entsprechenden Argumentregister völlig egal ist. Es soll also überhaupt kein WAM-Befehl für diese Argument-Positionen generiert werden.

Da SICStus Prolog dem Programmierer keinen Befehl zur Verfügung stellt, mit dem er ausdrücken könnte, daß ein Argument in einem Ziel tatsächlich nicht benötigt wird (auch nicht vom aufgerufenen Prädikat), also eine echte Void-Variable ist, muß für diese Diplomarbeit eine adäquate Erweiterung in den WAM-Code Generator pl_wam von SICStus Prolog integriert werden.

Die folgende neue Low-Level Instruktion `'VOID'/1` erlaubt es, bestimmte Argument-Positionen in Zielen als echte Void-Variablen zu bezeichnen, damit für sie keinerlei Put-Befehle erzeugt werden:

`'VOID'(V)`: Diese neue Low-Level Instruktion darf nur als erstes Ziel in einer Klausel verwendet werden. Der Parameter V muß ein Variablenname sein. Mittels `'VOID'(V)` wird angezeigt, daß alle Vorkommnisse von V in der Klausel echte Void-Variablen bezeichnen, V darf dabei nur in Argument-Positionen auftreten, nicht jedoch in Strukturen oder Listen. Für alle mit V bezeichneten Argument-Stellen werden keine Put-Instruktionen generiert.

Auf den ersten Blick erscheint diese Lösung nicht besonders „sauber“, durch Weglassen des Ziels `'VOID'(V)` sollte der Code zwar ineffizienter, jedoch immer noch korrekt sein. Jedoch bei obiger Lösung werden in diesem Fall alle Void-Variablen aneinander gebunden. Das beeinträchtigt jedoch nicht die Korrektheit, da ja die Void-Variablen in den aufgerufenen Prädikaten nicht verwendet werden, also ungebunden bleiben (sonst wären es keine Void-Variablen). Eine alternative Art der Benennung von Void-Variablen wäre, dem Befehl `'VOID'/1` eine Liste von Void-Variablen zu übergeben bzw. eine Instruktion `'VOID'/n` einzuführen, deren Argumente die einzelnen Void-Variablen bezeichnen. Auch die mehrfache Verwendung von `'VOID'/1` zur Benennung mehrerer Void-Variablen wäre denkbar. Dann dürfte jede Void-Variable nur ein einziges Mal in der Klausel (abgesehen von ihrer Deklaration mittels `'VOID'`) verwendet werden, durch Weglassen von `'VOID'` würden die Void-Variablen zu anonymen Variablen, was etwas eleganter als obige Lösung wirkt. Weil aber die Verwendung einer einzigen Void-Variable für den Programmierer einfacher ist und auch zu einem besser lesbaren Low-Level Prolog-Code führt, habe ich den Befehl `'VOID'(V)` wie oben beschrieben in den WAM-Code Generator pl_wam eingebaut.

4.2.2 Implementierung des Befehls

Da das Low-Level Prädikat `'VOID'/1` für diese Diplomarbeit in den WAM-Code Generator pl_wam von SICStus Prolog zu integrieren war[2], wird hier auch auf Implementierungsdetails eingegangen.

Ersetzte man in einer Klausel sämtliche Void-Variablen durch anonyme Variablen oder irgendwelche Konstanten, so würde die neue Klausel weiterhin korrekt

[2] `'VOID'/1` wurde in die von U. Neumerkel um `'SOURCE'/n` und `'SINK'/n` erweiterte Version von pl_wam (siehe [Neu95]) eingebaut.

funktionieren, es würden aber unnötige Put-Instruktionen exekutiert werden. Die Idee bei der Implementierung von `'VOID'(V)` ist, daß zuerst sämtliche Vorkommnisse der Void-Variable V in einer Klausel durch unterschiedliche Konstanten ersetzt werden, die sonst nirgends in dieser Klausel auftreten. Dann wird der WAM-Code für die so transformierte Klausel erzeugt, im Anschluß daran sind alle Vorkommnisse der Instruktion **put_constant**, welche zuvor neu eingeführte Konstanten in die Argumentregister laden, aus dem WAM-Code zu löschen. Der daraus resultierende Code enthält somit keinerlei Put-Instruktionen für die Void-Variablen.

Das Prädikat `compile_all/4` wird mit jeder Klausel einer Prolog-Datei aufgerufen, die auf WAM-Code übersetzt werden soll. Noch vor der Erzeugung einer internen Klausel-Darstellung wird ein neues Prädikat aufgerufen, welches die Aufgabe hat, sämtliche Void-Variablen durch neue, verschiedene Konstanten zu ersetzen und diese in einer Liste zu sammeln (diese Liste wird später benötigt, um die entsprechenden Vorkommnisse der Instruktion **put_constant** aus dem WAM-Code zu entfernen). Zunächst wird eine Listen-Repräsentation der Klausel erzeugt, damit der Prolog-Code, welcher die Klausel transformiert, möglichst ohne Cuts auskommt, was wesentlich „eleganter" ist, als direkt auf der Und-Listen-Darstellung der Klausel zu operieren. Bei dieser Umwandlung der Klausel auf eine Listen-Repräsentation werden Ziele, die Variablen sind (Metacalls), auf den SICStus Prolog Befehl `call/1` abgebildet. Falls das erste Ziel in der Klausel kein Aufruf von `'VOID'/1` ist (dabei muß auch berücksichtigt werden, daß sich die zu transformierende Klausel in einem Modul befinden könnte, denn in diesem Fall wird `user:'VOID'/1` aufgerufen), ist keine Transformation vorzunehmen, es gibt in der Klausel keine Void-Variablen. Es wird die Listen-Darstellung der Klausel in eine Repräsentation zurückgewandelt, welche zu der ursprünglichen äquivalent ist. Ist das erste Ziel in der Klausel jedoch die Deklaration einer Void-Variable V, werden sämtliche Ziele der Klausel nach Vorkommnissen der Variable V durchsucht. Diese werden durch neue Variablen ersetzt, die auch in einer Variablen-Liste gesammelt werden (diese Variablen sind später an unterschiedliche, neue Konstanten zu binden). Dabei wird nicht geprüft, ob die Variable V in Argument-Position auftritt oder in einer Struktur bzw. in einer Liste, denn diese Überprüfung ergibt sich später automatisch. Weiters werden sämtliche positive ganze Zahlen, die in der Klausel vorkommen, in einer Zahlen-Liste gespeichert (später werden als neue Konstanten die kleinsten positiven ganzen Zahlen eingesetzt, die nicht in der ursprünglichen Klausel vorkommen). Dabei müssen unbedingt auch die positiven ganzzahligen Konstanten aus dem Klauselkopf berücksichtigt werden: Ersetzte man im ersten Ziel eine Void-Variable durch eine Konstante, die auch im Klauselkopf in Argument-Position vorkommt, so würde der WAM-Code Generator pl_wam von SICStus Prolog in vielen Fällen nicht die Instruktion **put_constant**, sondern **put_x_value** oder überhaupt keinen Befehl (falls sich die Konstante im ersten Ziel in der gleichen

Argument-Position wie im Klauselkopf befindet) erzeugen, da nach der Unifikation mit der Konstante im Klauselkopf (mittels **get_constant**) die Konstante über ein X-Register verfügbar ist. Somit könnte später das der Void-Variable entsprechende Vorkommnis der Instruktion **put_constant** im WAM-Code nicht gefunden und gelöscht werden, was in dieser Implementierung von `'VOID'/1` als Fehler angesehen wird. Die Zahlen-Liste wird sortiert, sodann werden die Variablen aus der Variablen-Liste an die jeweils nächste positive ganze Zahl gebunden, die nicht in der Zahlen-Liste vorkommt. Durch die Bindung der Elemente aus der Variablen-Liste werden auch die Void-Variablen in der Klausel an die entsprechenden Konstanten (Zahlen) gebunden. Die transformierte Klausel muß wieder in eine Repräsentation umgewandelt werden, die zur ursprünglichen Darstellung äquivalent ist (Und-Liste).

Im Prädikat `compile_all/4` wird die zuvor generierte Liste aller Void-Konstanten an das Prädikat `compile_internals/6` weitergereicht, welches um zwei Argumente für die Liste der Void-Konstanten erweitert worden ist, denn diese Liste wird als Listen-Differenz übergeben. Das Prädikat `compile_internals/6` übergibt die Listen-Differenz mit den Void-Konstanten an `compile_one_proc/9` (ebenfalls um 2 Argumente erweitert); dieses Prädikat löscht nach Erzeugung des WAM-Codes alle Vorkommnisse der Instruktion **put_constant**, welche die Void-Konstanten aus der Listen-Differenz in Argumentregister schreiben. Da eine Void-Konstante nur genau ein Mal vorkommen darf, wird sie nach ihrem Auftreten auch aus der Listen-Differenz entfernt. Die Darstellung der Void-Konstanten als Listen-Differenz ist deshalb notwendig, weil `compile_one_proc/9` indirekt rekursiv wieder `compile_internals/6` aufruft. Ist bereits der endgültige WAM-Code für die ganze Klausel erzeugt worden, verlangt `compile_internals/6`, daß die Listen-Differenz der Void-Konstanten die leere Liste beschreibt, was bedeutet, daß alle Void-Konstanten, welche für Void-Variablen substituiert worden waren, im WAM-Code gefunden und gelöscht worden sind. Sind noch Void-Konstanten in der Listen-Differenz übrig geblieben, so deutet das darauf hin, daß Void-Variablen falsch (also nicht in Argument-Position, sondern in einer Struktur oder in einer Liste) verwendet worden sind, da keine diesen Void-Konstanten entsprechenden Vorkommnisse von **put_constant** gefunden worden sind.

4.2.3 Beispiel

In diesem Abschnitt wird die korrekte Verwendung der Low-Level Instruktion `'VOID'/1` an Hand eines kurzen Beispiels gezeigt. Das folgende Prädikat `pn/2` beschreibt die transitive Hülle der Relation `p1/2`:

```
pn(A,B) :-
     p1(A,B).
pn(A,C) :-
     p1(A,B),
     pn(B,C).

p1(a,b).
p1(b,c).
```

Der WAM-Code Generator pl_wam von SICStus Prolog erzeugt für dieses Prolog-Programm folgenden Code:

```
                            % pn(A,B) :-
execute(p1/2)               %      p1(A,B).

allocate                    % pn(A,
get_y_variable(1,1)         %      C) :-
put_y_variable(0,1)         %
init([])                    %
call(p1/2,2)                %      p1(A,B),
put_y_unsafe_value(0,0)     %
put_y_value(1,1)            %
deallocate                  %
execute(pn/2)               %      pn(B,C).

get_constant_x0(a)          % p1(a,
get_constant(b,1)           %      b)
proceed                     %        .

get_constant_x0(b)          % p1(b,
get_constant(c,1)           %      c)
proceed                     %        .
```

Die beiden Fakten des Prädikats `p1/2` führen jeweils 3 Befehle aus. Die erste, nicht rekursive Kettenregel von `pn/2` besteht nur aus einer Instruktion, die rekursive Regel exekutiert 9 SICStus WAM-Befehle und allokiert 2 Stack-Variablen. Diese Regel soll nun optimiert werden.

Um unnötige Datenbewegungen zu vermeiden, werden die Fakten von `p1/2` so umgeschrieben, daß das zweite Argument vom Aufrufer wie ein Callee Saved Register behandelt werden kann:

```
pn(A,B) :-
      'VOID'(V),
      p1(A,V,B).
pn(A,C) :-
      p1(A,C,B),
      'SOURCE'(-,+C2,+B2),
      pn(B2,C2).

p1(a,X,b) :-
      'SINK'(+a,+X,+b).
p1(b,X,c) :-
      'SINK'(+b,+X,+c).
```

Wird das Prädikat `p1/2` noch an anderen Stellen im Programm aufgerufen, müssen Void-Variablen als zweite Argumente eingefügt werden, oder es muß folgendes Schnittstellen-Prädikat definiert werden:

```
p1(A,B) :-
      'VOID'(V),
      p1(A,V,B).
```

Der WAM-Code für die optimierte Version der Prädikate `p1/3` und `pn/2` sieht folgendermaßen aus:

```
                        % pn(A,B) :-
get_x_variable(2,1)     %       'VOID'(V),
execute(p1/3)           %       p1(A,V,B).

                        % pn(A,C) :-
put_x_void(2)           %
allocate                %
init([])                %
call(p1/3,0)            %       p1(A,C,B),
put_x_value(2,0)        %       'SOURCE'(-,+C2,+B2),
deallocate              %
execute(pn/2)           %       pn(B2,C2).

get_constant_x0(a)      % p1(a,X,
get_constant(b,2)       %         b) :-
proceed                 %       'SINK'(+a,+X,+b).

get_constant_x0(b)      % p1(b,X,
get_constant(c,2)       %         c) :-
proceed                 %       'SINK'(+b,+X,+c).
```

Der WAM-Code der beiden Fakten des Prädikats `p1/3` hat sich durch die Optimierung kaum geändert, statt auf das X-Register 1 wird nun auf das Register 2 zugegriffen. Die rekursive Regel von `pn/2` führt jetzt nur mehr 7 Instruktionen aus, also um 2 weniger als in der nicht optimierten Version. Außerdem allokiert diese Regel nun überhaupt keine Stack-Variable. Allerdings benötigt die erste Regel von `pn/2` nun 2 Befehle, um einen mehr als in der ursprünglichen Version. Insgesamt wird jedoch in den meisten Fällen die optimierte Version eine höhere Performance erzielen als das nicht optimierte Programm.

Dieses Beispiel läßt sich noch besser optimieren, indem man die Ausgabe-Argumente ähnlich wie im Summen-Beispiel behandelt. Allerdings wird dann `'VOID'/1` nicht benötigt, da die optimierten Prädikate die Arität 1 hätten. Aus diesem Grund, und weil obiges Beispiel bereits den Nutzen eines Callee Saved Registers zeigt (die Verwendung von Callee Saved Registern ist das Hauptthema des nächsten Abschnitts), ist hier nicht die bestmögliche Optimierung demonstriert worden.

5 Interprozedurale Registerallokation

Dieser Abschnitt beschreibt, wie man mit Hilfe der Low-Level Instruktionen `'SOURCE'/n`, `'SINK'/n` und `'VOID'/1` interprozedurale Registerallokationen erreichen kann. Es wird eine Transformation vorgestellt, die Prolog-Programme so umwandelt, daß alle Prädikate dieselbe Arität n haben. Dann kann nämlich eine Calling Convention eingeführt werden, die genau festlegt, welche der n Argumentregister Caller Saved und welche Callee Saved Register sind. Mittels einfacher Heuristiken wird entschieden, für welche Argumente Caller Saved Register bzw. Callee Saved Register zu allokieren sind, um eine bessere Performance zu erzielen.

5.1 Transformation

5.2 Beispiele

In diesem Abschnitt wird die zuvor beschriebene Transformation an einigen einfachen, kurzen Beispielen vorgezeigt, wobei jeweils auch die durch die Optimierung erreichten Performance-Gewinne ermittelt werden.

5.2.1 Verwandtschaftsbeziehungen

Mittels der Fakten `kind_von/2`, `gatte_gattin/2`, `männlich/1` und `weiblich/1` kann ein Familien-Stammbaum als Prolog-Datenbasis dargestellt werden. Das Prädikat `sohn_von/2` wird unter Verwendung von `kind_von/2` und `männlich/1` folgendermaßen definiert:

```
sohn_von(S,E) :-
     männlich(S),
     kind_von(S,E).

kind_von(max,anna).
kind_von(max,sepp).

männlich(max).
```

Der WAM-Code Generator pl_wam von SICStus Prolog erzeugt für obiges Prolog-Programm folgenden WAM-Code:

```
allocate                  % sohn_von(S,E) :-
get_y_variable(1,1)       %
get_y_variable(0,0)       %
init([])                  %
call(männlich/1,2)        %      männlich(S),
put_y_value(0,0)          %
put_y_value(1,1)          %
deallocate                %
execute(kind_von/2)       %      kind_von(S,E).

get_constant_x0(max)      % kind_von(max,
get_constant(anna,1)      %                anna)
proceed                   %                     .

get_constant_x0(max)      % kind_von(max,
get_constant(sepp,1)      %                sepp)
proceed                   %                     .

get_constant_x0(max)      % männlich(max)
proceed                   %                .
```

Der WAM-Code für die Prädikate `kind_von/2` und `männlich/1` ist optimal, die Regel von `sohn_von/2` jedoch bietet viele Optimierungsmöglichkeiten, denn sie exekutiert 9 WAM-Instruktionen und allokiert dabei 2 Stack-Variablen. Zum Beweisen des Ziels '`:- sohn_von(max,anna).`' werden daher beispielsweise $9 + 2 + 3 = 14$ Befehle ausgeführt.

Jetzt soll für obiges Beispiel eine interprozedurale Registerallokation erreicht werden. Es wird angenommen, daß eine WAM mit 2 Argumentregistern zur Verfügung steht, wobei beide Argumente Callee Saved Registern entsprechen. In der Praxis wird kaum eine derartige WAM verwendet werden, aber diese WAM mit 2 Callee Saved Registern reicht aus, um für dieses Beispiel optimalen Code zu erzeugen. Die drei Prädikate `sohn_von/2`, `kind_von/2` und `männlich/1` befolgen bereits die Calling Convention, dies muß aber durch die entsprechende Verwendung von `'SOURCE'/n` und `'SINK'/n` explizit gemacht werden, um eine bessere Code-Erzeugung zu bewirken. Weiters muß das Prädikat `männlich/1` um ein Argument erweitert werden, welches dem zweiten Callee Saved Register entspricht. Hüll-Prädikate sind in diesem Beispiel nicht erforderlich, weil die sich alle drei Prädikate an die Calling Convention halten. Im folgenden ist das transformierte

Prolog-Programm dargestellt:

```
sohn_von(S,E) :-
     männlich(S,E),
     'SOURCE'(+S2,+E2),
     kind_von(S2,E2).

kind_von(max,anna) :-
     'SINK'(+max,+anna).
kind_von(max,sepp) :-
     'SINK'(+max,+sepp).

männlich(max,S) :-
     'SINK'(+max,+S).
```

Der WAM-Code, den SICStus Prolog für die optimierte Version generiert, sieht folgendermaßen aus:

```
allocate                % sohn_von(S,E) :-
init([])                %
call(männlich/2,0)      %     männlich(S,E),
deallocate              %     'SOURCE'(+S2,+E2),
execute(kind_von/2)     %     kind_von(S2,E2).

get_constant_x0(max)    % kind_von(max,
get_constant(anna,1)    %              anna) :-
proceed                 %     'SINK'(+max,+anna).

get_constant_x0(max)    % kind_von(max,
get_constant(sepp,1)    %              sepp) :-
proceed                 %     'SINK'(+max,+sepp).

get_constant_x0(max)    % männlich(max,S) :-
proceed                 %     'SINK'(+max,+S).
```

Man beachte, daß sich der WAM-Code für die beiden Prädikate `kind_von/2` und `männlich/1` durch die Transformation überhaupt nicht geändert hat. Nach der interprozeduralen Registerallokation ist nun allerdings auch der Code für die Regel `sohn_von/2` optimal, es werden nur mehr 5 SICStus-WAM Befehle ausgeführt (gegenüber 9 Instruktionen für die nicht optimierte Version). Weiters wird keine einzige Stack-Variable benötigt (statt 2 Stack-Variablen). Nun werden zum Beweisen des Ziels '`:- sohn_von(max,anna).`' nur mehr $5+2+3=10$

Befehle ausgeführt, um 4 Instruktionen weniger als vor der interprozeduralen Registerallokation. Es ergibt sich für dieses Ziel daher ein Speedup um den Faktor $14/10 = 1.4$, d.h. die optimierte Version ist um 40% schneller, wenn angenommen wird, daß alle SICStus WAM-Befehle in etwa gleich viel Zeit für ihre Ausführung benötigen. Eine exakte Messung der Maschinenbefehle, die auf einem Intel Pentium Prozessor zum Beweisen des Ziels `':- sohn_von(max,anna).'` exekutiert werden, ergibt einen Speedup um den Faktor 1.141, d.h. das optimierte Programm ist nur um 14.1% schneller. Dieses relativ schlechte Ergebnis läßt sich dadurch erklären, daß der Code für den Aufruf des Ziels für beide Versionen identisch ist, aber in obiger Berechnung nicht berücksichtigt worden ist.

Es ist zu beachten, daß das ursprüngliche Prädikat `männlich/1` durch die Optimierung zum Prädikat `männlich/2` transformiert worden ist, daß also die Arität dieses Prädikats in der optimierten Version größer ist. Bei einem Aufruf der Gestalt `':- sohn_von(S,E).'` muß das Prädikat `männlich/2` wegen Backtracking i.A. mehrere Lösungen liefern, es werden also Wahlpunkte angelegt. Diese sind im optimierten Programm größer, es muß ein Argumentregister mehr im Wahlpunkt gespeichert werden, wodurch Performance-Gewinne durch die Optimierung teilweise zunichte gemacht werden. Eine genaue Analyse des Overheads, der bei nicht-deterministischen Prädikaten durch eine größere Zahl an Argumenten verursacht wird, findet sich im nächsten Abschnitt.

5.2.2 Komplementäre DNA-Sequenzen

Eine DNA-Sequenz ist eine beliebig lange Folge der 4 Basen Adenin (A), Thymin (T), Cytosin (C) und Guanin (G), wobei jeweils die Basen Adenin und Thymin sowie Cytosin und Guanin zueinander komplementär sind. Hier wird unter der komplementären DNA-Sequenz jene Folge verstanden, die durch Vertauschen der einzelnen Basen durch ihre Kompementär-Basen entsteht; die DNA-Sequenz selbst wird aber nicht umgedreht.

Das folgende Prädikat `cgats_gctas/2` beschreibt alle möglichen DNA-Sequenzen mit den dazugehörigen komplementären Sequenzen. Die 4 Fakten des Prädikats `cgat_gcta/2` beschreiben die 4 Basen mit ihren jeweiligen Komplementär-Basen:

```
cgats_gctas([],[]).
cgats_gctas([X|Xs],[Y|Ys]) :-
     cgat_gcta(X,Y),
     cgats_gctas(Xs,Ys).

cgat_gcta(c,g).
cgat_gcta(g,c).
cgat_gcta(a,t).
cgat_gcta(t,a).
```

Der WAM-Code Generator pl_wam von SICStus Prolog erzeugt für obige

Klauseln folgenden Code:

```
get_nil_x0                 % cgats_gctas([],
get_nil(1)                 %               [])
proceed                    %                 .

get_list_x0                % cgats_gctas([
unify_x_variable(0)        %             X|
allocate                   %
unify_y_variable(0)        %               Xs],
get_list(1)                %                   [
unify_x_variable(1)        %                    Y|
unify_y_variable(1)        %                      Ys]) :-
init([])                   %
call(cgat_gcta/2,2)        %      cgat_gcta(X,Y),
put_y_value(0,0)           %
put_y_value(1,1)           %
deallocate                 %
execute(cgats_gctas/2)     %      cgats_gctas(Xs,Ys).

get_constant_x0(c)         % cgat_gcta(c,
get_constant(g,1)          %             g)
proceed                    %               .

get_constant_x0(g)         % cgat_gcta(g,
get_constant(c,1)          %             c)
proceed                    %               .

get_constant_x0(a)         % cgat_gcta(a,
get_constant(t,1)          %             t)
proceed                    %               .

get_constant_x0(t)         % cgat_gcta(t,
get_constant(a,1)          %             a)
proceed                    %               .
```

Obiger WAM-Code für das Fakt des Prädikats `cgats_gctas/2` (bestehend aus 3 Befehlen) ist offensichtlich optimal, ebenso der Code für die 4 Fakten des Prädikats `cgat_gcta/2` (jeweils 3 Instruktionen). Jedoch der Code für die rekursive Regel von `cgats_gctas/2` läßt noch Optimierungen zu, denn es werden 2 Stack-Variablen verwendet. Im Idealfall können in diesem Prädikat durch eine bessere Registerallokation die beiden Vorkommnisse der WAM-Instruktion **put_y_value** eliminiert werden, weitere Optimierungsmöglichkeiten bietet diese Regel nicht. Wird das Prädikat `cgats_gctas/2` zur Umwandlung einer im ersten

Argument übergegebenen DNA-Sequenz der Länge n in ihre Komplementär-Folge eingesetzt, sind $(13+3)n+3 = 16n+3$ WAM-Instruktionen zu exekutieren, wobei $26n$ Stack-Variablen allokiert werden müssen. Dabei wird angenommen, daß wegen des Indexings nach dem Hauptfunktor des ersten Arguments immer sofort die passende Klausel ausgeführt wird.

Nun soll für diesen WAM-Code eine interprozedurale Registerallokation erreicht werden. Es wird angenommen, daß eine WAM mit 6 Argumentregistern zur Verfügung steht[3] Als Calling Convention wird festgelegt, daß die ersten drei Argumente Caller Saved Registern und die letzen drei Callee Saved Registern entsprechen. Innerhalb des Prädikats `cgats_gctasINT/6` wird diese Calling Convention nicht berücksichtigt, die Liste `[Y|Ys]` (ursprünglich das zweite Argument von `cgats_gctas/2`) wird auf das erste Callee Saved Register abgebildet (also auf das vierte Argument). Für die Liste `Xs` wird das zweite Callee Saved Register (Argument-Position 5) allokiert, um die Erzeugung einer Stack-Variable zu vermeiden. Da `cgats_gctasINT/6` die Calling Convention nicht befolgt (2 Callee Saved Register werden nicht gesichert), wird das Hüll-Prädikat `cgats_gctas/6` als Schnittstelle nach außen definiert. Der Name des Prädikats `cgats_gctasINT/6` soll darauf hinweisen, daß dieses Prädikat nur für „internen" Gebrauch gedacht ist, weil es die Calling Convention nicht berücksichtigt. Nach Möglichkeit soll dennoch `cgats_gctasINT/6` verwendet werden, um den Overhead des Schnittstellen-Prädikats `cgats_gctas/6` zu vermeiden, welches die ersten beiden Callee Saved Register (Argument-Positionen 4 und 5) sichert und

[3]Es würde auch eine WAM mit 4 Registern ausreichen, aber in diesem Beispiel soll auch gezeigt werden, daß die Verwendung von mehr Registern keinen Overhead im WAM-Code verursacht (abgesehen von größeren Wahlpunkten).

nach dem Aufruf von `cgats_gctasINT/6` wieder restauriert.

```
cgats_gctas(Xs,Ys,_,S1,S2,S3) :-
     'VOID'(V),
     cgats_gctasINT(Xs,V,V,Ys,V,S3),
     'SOURCE'(-,-,-,-,-,+S32),
     'SINK'(-,-,-,+S1,+S2,+S32).

cgats_gctasINT([],_,_,[],_,S3) :-
     'SINK'(-,-,-,-,-,+S3).
cgats_gctasINT([X|Xs],_,_,[Y|Ys],_,S3) :-
     'VOID'(V),
     cgat_gcta(X,Y,V,Ys,Xs,S3),
     'SOURCE'(-,-,-,+Ys2,+Xs2,+S32),
     cgats_gctasINT(Xs2,V,V,Ys2,V,S32).

cgat_gcta(c,g,_,S1,S2,S3) :-
     'SINK'(-,-,-,+S1,+S2,+S3).
cgat_gcta(g,c,_,S1,S2,S3) :-
     'SINK'(-,-,-,+S1,+S2,+S3).
cgat_gcta(a,t,_,S1,S2,S3) :-
     'SINK'(-,-,-,+S1,+S2,+S3).
cgat_gcta(t,a,_,S1,S2,S3) :-
     'SINK'(-,-,-,+S1,+S2,+S3).
```

Wird das Prädikat `cgats_gctas/6` auch von nicht transformiertem Prolog-Code aus aufgerufen, ist folgendes Schnittstellen-Prädikat zu definieren, dessen WAM-Code nur aus der Instruktion **execute(cgats_gctas/6)** besteht:

```
cgats_gctas(Xs,Ys) :-
     'VOID'(V),
     cgats_gctas(Xs,Ys,V,V,V,V).
```

Der WAM-Code für die 4 Klauseln des Prädikats `cgat_gcta/6` ist völlig identisch mit dem Code für `cgat_gcta/2` aus dem nicht optimierten Beispiel, es werden also weiterhin je 3 WAM-Instruktionen generiert. Der WAM-Code für die beiden Prädikate `cgats_cgtas/6` und `cgats_gctasINT/6` sieht folgendermaßen aus:

```
allocate                   % cgats_gctas(Xs,Ys,_,S1,S2,S3) :-
get_y_variable(1,4)        %
get_y_variable(0,3)        %
get_x_variable(3,1)        %
init([])                   %     'VOID'(V),
call(cgats_gctasINT/6,2)   %     cgats_gctasINT(Xs,V,V,Ys,V,S3),
put_y_value(0,3)           %
put_y_value(1,4)           %
deallocate                 %     'SOURCE'(-,-,-,-,-,+S32),
execute('SINK'/6)          %     'SINK'(-,-,-,+S1,+S2,+S32).

get_nil_x0                 % cgats_gctasINT([],_,_,
get_nil(3)                 %                      [],_,S3) :-
proceed                    %     'SINK'(-,-,-,-,-,+S3).

get_list_x0                % cgats_gctasINT([
unify_x_variable(0)        %                 X|
unify_x_variable(4)        %                   Xs],_,_,
get_list(3)                %                [
unify_x_variable(1)        %                 Y|
unify_x_variable(3)        %                   Ys],_,S3) :-
allocate                   %
init([])                   %     'VOID'(V),
call(cgat_gcta/6,0)        %     cgats_gcta(X,Y,V,Ys,Xs,S3),
get_x_variable(0,4)        %     'SOURCE'(-,-,-,+Ys2,+Xs2,+S32),
deallocate                 %
execute(cgats_gctasINT/6)  %     cgats_gctasINT(Xs2,V,V,Ys2,V,S32).
```

Die erste, nicht rekursive Regel von `cgats_gctasINT/6` exekutiert nach wie vor 3 SICStus WAM-Instruktionen, die rekursive Regel allerdings benötigt nur mehr 12 Befehle (gegenüber 13 Instruktionen vor der interprozeduralen Registerallokation) und allokiert keine Stack-Variable (in der nicht optimierten Version benötigt die rekursive Regel 2 Stack-Variablen). Allerdings ist zur Wahrung der Calling Convention nach außen hin das Hüll-Prädikat `cgats_gctas/6` notwendig, welches 10 WAM-Befehle ausführt und 2 Stack-Variablen verwendet. Zur Umwandlung einer im ersten Argument übergegebenen DNA-Sequenz der Länge n in ihre Komplementär-Folge werden nun unter Verwendung des Schnittstellen-Prädikats `cgats_gctas/6` $(12 + 3)n + 3 + 10 = 15n + 13$ WAM-Instruktionen exekutiert (statt $(13 + 3)n + 3 = 16n + 3$ Befehlen vor der Optimierung), wobei insgesamt nur 2 Stack-Variablen benötigt werden. Für $n < 10$ exekutiert die nicht optimierte Version weniger Instruktionen, für $n = 10$ führen beide Versionen gleich viele Befehle aus, und für $n > 10$ benötigt das optimierte Prädikat weniger Instruktionen. Für große Werte von n ergibt sich ein theoretischer Speedup um zirka 6.6%, wenn angenommen wird, daß alle WAM-Befehle gleich viel Zeit zur Ausführung erfordern. In der Praxis werden größere Performance-Gewinne erzielt, weil weniger Stack-Variablen allokiert werden müssen; so zeigen beispielsweise Benchmarks für $n = 200$ einen Speedup von 27% bis 30%, die Verwendung von noch größeren Werten für n führt zu keinen anderen Ergebnissen, bei $n = 200$ spielt also der Overhead durch das Hüll-Prädikat und das Iterations-Prädikat, das beim Benchmark aufgerufen wird[4], keine Rolle mehr. Für $n = 100$ wird immerhin ein Speedup von 23% bis 26% erreicht. Eine exakte Messung der Maschineninstruktionen, die auf einem Intel Pentium Prozessor für $n = 40$ ausgeführt werden, ergibt einen Speedup um den Faktor 1.113, d.h. das optimierte Programm ist um 11.3% schneller. Allerdings muß hier berücksichtigt werden, daß die Anzahl der Maschinenbefehle, die zum Generieren der 40-elementigen Liste exekutiert werden, bei beiden Versionen gleich groß ist und im Benchmark-Programm mitgemessen wird. Weiters kann oft direkt das Prädikat `cgats_gctasINT/6` verwendet werden, wenn man berücksichtigt, daß dieses Prädikat die ersten beiden Callee Saved Register nicht rettet, was weitere Geschwindigkeitsgewinne, v.a. für kleine Werte von n ermöglicht.

5.2.3 Aufteilen einer Liste

Das folgende Prädikat `split/5` kann verwendet werden, um eine Liste aufzuteilen, deren Elemente entweder den Hauptfunktor `'+'/1` oder `'-'/1` haben. Die Liste der mit `'+'/1` bzw. `'-'/1` markierten Elementen wird im erste Argument übergeben, das zweite und dritte Argument enthält die Listen-Differenz der mit `'+'/1` versehenen „positiven“ Elemente (aber ohne Markierungen), und das vierte und fünfte Argument repräsentieren die Listen-Differenz der mit `'-'/1` markierten „negativen“ Elemente (ebenfalls ohne Markierungen).

```
split([],Ps,Ps,Ns,Ns).
split([X|Xs],Ps0,Ps,Ns0,Ns) :-
     split2(X,Ps0,Ps1,Ns0,Ns1),
     split(Xs,Ps1,Ps,Ns1,Ns).

split2(+X,[X|Ps],Ps,Ns,Ns).
split2(-X,Ps,Ps,[X|Ns],Ns).
```

[4]Die Messungen wurden auf einem PC (Intel Pentium, 133MHz, 64MB RAM) unter Linux durchgeführt, wobei das Prädikat `statistics/2` von SICStus Prolog zur Ermittlung der Laufzeiten verwendet wurde. Da dieses Prädikat nur Zeit-Ergebnisse mit der Granularität 1ms liefert, waren Iterationen der Benchmarks notwendig, um eine Laufzeit $\gg$ 0ms zu erreichen.

Der WAM-Code Generator pl_wam von SICStus Prolog erzeugt für dieses Prolog-Programm folgenden ziemlich ineffizienten Code:

```
get_nil_x0                 % split([],
get_x_value(1,2)           %          Ps,Ps,
get_x_value(3,4)           %                Ns,Ns)
proceed                    %                      .

get_list_x0                % split([X|Xs],Ps0,Ps,Ns0,Ns) :-
allocate                   %
get_y_variable(4,4)        %
get_y_variable(2,2)        %
unify_x_variable(0)        %
unify_y_variable(0)        %
put_y_variable(1,2)        %
put_y_variable(3,4)        %
init([])                   %
call(split2/5,5)           %      split2(X,Ps0,Ps1,Ns0,Ns1),
put_y_value(0,0)           %
put_y_unsafe_value(1,1)    %
put_y_value(2,2)           %
put_y_unsafe_value(3,3)    %
put_y_value(4,4)           %
deallocate                 %
execute(split/5)           %      split(Xs,Ps1,Ps,Ns1,Ns).

get_structure_x0((+)/1)    % split2(+
unify_x_variable(5)        %        X,
get_list(1)                %          [
unify_x_value(5)           %           X|
unify_x_local_value(2)     %             Ps],Ps,
get_x_value(3,4)           %                    Ns,Ns)
proceed                    %                          .

get_structure_x0((-)/1)    % split2(-
unify_x_variable(5)        %        X,
get_x_value(1,2)           %          Ps,Ps,
get_list(3)                %                [
unify_x_value(5)           %                 X|
unify_x_local_value(4)     %                   Ns],Ns)
proceed                    %                          .
```

Der WAM-Code für das Fakt von `split/5` ist mit 4 Instruktionen bereits optimal, aber die rekursive Regel exekutiert 17 WAM-Befehle und allokiert 5

Stack-Variablen, es besteht bei dieser Regel also ein großes Optimierungspotential. Der Code der zwei Fakten des Prädikats `split2/5` ist akzeptabel, er besteht aus je 7 WAM-Befehlen und allokiert keine Stack-Variable. Folglich werden zum Aufteilen einer n Elemente langen Liste $(17+7)n+4 = 24n+4$ SICStus WAM-Instruktionen exekutiert und $5n$ Stack-Variablen verwendet.

Das Hauptproblem bei der rekursiven Regel des Prädikats `split/5` ist, daß die Argumente `Ps1` und `Ns1` für den Aufruf von `split2/5` im dritten bzw. vierten Argumentregister allokiert sind, beim rekursiven Aufruf jedoch im zweiten und dritten Argumentregister benötigt werden. Weiters werden die Argumente `Ps` und `Ns` im Environment gespeichert, was sich durch interprozedurale Registerallokation sicherlich vermeiden läßt.

Nun wird versucht, mit Hilfe einer interprozeduralen Registerallokation den WAM-Code für die rekursive Regel von `split/5` zu optimieren. Es wird angenommen, daß eine WAM mit 6 Argumentregistern zur Verfügung steht, wobei die ersten 5 Argumente Caller Saved Registern und das letze Argument einem Callee Saved Register entsprechen. Das modifizierte Prädikat `split2INT/6` läßt das dritte, fünfte und sechste Argumentregister unverändert, also neben dem echten Callee Saved Register (Argument-Position 6) werden auch das dritte und fünfte Caller Saved Register wie Callee Saved Register behandelt. Für das ursprüngliche dritte und fünfte Argument von `split2/5` (Ausgabe-Argumente der Listen-Differenzen) werden das zweite und das vierte Argumentregister allokiert. Dadurch kann die rekursive Regel von `splitINT/6` die Variablen `Ps` und `Ns` in der dritten bzw. fünften Argument-Position halten, während die zweite bzw. vierte Argument-Stelle sowohl für `Ps0` und `Ps1` bzw. für `Ns0` und `Ns1` verwendet wird. Durch diese Allokation werden Datenbewegungen weitgehend vermieden, nur die Variable `Xs` kann nicht immer dieselbe Argument-Position verwenden. Für `Xs` wird das Callee Saved Register eingesetzt, um die Allokation einer Stack-Variable zu verhindern. Weil das Prädikat `splitINT/6` die Calling Convention nicht befolgt (das Callee Saved Register wird verwendet, ohne es zu sichern), wird ein Hüll-Prädikat `split/6` als Schnittstelle nach außen definiert. Die Verwendung des Schnittstellen-Prädikats ist möglichst zu vermeiden, da es den Performance-

Gewinn, der durch die Optimierung erzielt wird, teilweise zunichte macht.

```
split(Xs,Ps0,Ps,Ns0,Ns,S1) :-
      'VOID'(V),
      splitINT(Xs,Ps0,Ps,Ns0,Ns,V),
      'SINK'(-,-,-,-,-,+S1).

splitINT([],Ps,Ps,Ns,Ns,_).
splitINT([X|Xs],Ps0,Ps,Ns0,Ns,_) :-
      'VOID'(V),
      split2INT(X,Ps0,Ps,Ns0,Ns,Xs),
      'SOURCE'(-,+Ps1,+Ps2,+Ns1,+Ns2,+Xs2),
      splitINT(Xs2,Ps1,Ps2,Ns1,Ns2,V).

split2INT(+X,[X|Ps],S1,Ns,S2,S3) :-
      'SINK'(-,+Ps,+S1,+Ns,+S2,+S3).
split2INT(-X,Ps,S1,[X|Ns],S2,S3) :-
      'SINK'(-,+Ps,+S1,+Ns,+S2,+S3).
```

Wird das Prädikat `split2INT/6` direkt verwendet (nicht nur als Hilfsprädikat), muß folgendes Hüll-Prädikat `split2/6` definiert werden, um die Bindung der Argument 2 und 3 bzw. 4 und 5 aneinander zu erreichen:

```
split2(X,Ps0,Ps,Ns0,Ns,S1) :-
      split2INT(X,Ps0,Ps,Ns0,Ns,S1),
      'SOURCE'(-,+Ps2,+Ps2,+Ns2,+Ns2,+S2),
      'SINK'(-,-,-,-,-,+S2).
```

Der WAM-Code für das optimierte Programm sieht folgendermaßen aus:

```
allocate                 % split(Xs,Ps0,Ps,Ns0,Ns,S1) :-
get_y_variable(0,5)      %
init([])                 %    'VOID'(V),
call(splitINT/6,1)       %    splitINT(Xs,Ps0,Ps,Ns0,Ns,V),
put_y_value(0,5)         %    'SINK'(-,-,-,-,-,+S1)
deallocate               %
execute('SINK'/6)        %                          .

get_nil_x0               % splitINT([],
get_x_value(1,2)         %            Ps,Ps,
get_x_value(3,4)         %                  Ns,Ns,_)
proceed                  %                          .

get_list_x0              % splitINT([
unify_x_variable(0)      %          X|
unify_x_variable(5)      %            Xs],Ps0,Ps,Ns0,Ns,_) :-
allocate                 %
init([])                 %    'VOID'(V),
call(split2INT/6,0)      %    split2INT(X,Ps0,Ps,Ns0,Ns,Xs),
get_x_variable(0,5)      %    'SOURCE'(-,+Ps1,+Ps2,+Ns1,+Ns2,+Xs2),
deallocate               %
execute(splitINT/6)      %    splitINT(Xs2,Ps1,Ps2,Ns1,Ns2,V).

get_structure_x0((+)/1) % split2INT(+
unify_x_variable(6)      %           X,
get_list(1)              %             [
unify_x_value(6)         %              X|
unify_x_variable(6)      %                Ps],S1,Ns,S2,S3) :-
put_x_value(6,1)         %    'SINK'(-,+Ps,+S1,+Ns,+S2,+S3)
proceed                  %                                  .

get_structure_x0((-)/1) % split2INT(-
unify_x_variable(6)      %           X,Ps,S1,
get_list(3)              %                   [
unify_x_value(6)         %                    X|
unify_x_variable(6)      %                      Ns],S2,S3) :-
put_x_value(6,3)         %    'SINK'(-,+Ps,+S1,+Ns,+S2,+S3)
proceed                  %                                  .
```

Das Schnittstellen-Prädikat `split2INT/6` wird vermutlich nicht benötigt, der WAM-Code ist hier trotzdem abgebildet:

```
allocate               % split2(X,Ps0,Ps,Ns0,Ns,S1) :-
init([])               %
call(split2INT/6,0)    %    split2INT(X,Ps0,Ps,Ns0,Ns,S1),
get_x_value(1,2)       %    'SOURCE'(-,+Ps2,+Ps2,+Ns2,+Ns2,+S2),
get_x_value(3,4)       %
deallocate             %
execute('SINK'/6)      %    'SINK'(-,-,-,-,-,+S2).
```

Die erste Klausel des Prädikats `splitINT/6` führt auch in der optimierten Version 4 SICStus WAM-Befehle aus, die rekursive Regel jedoch benötigt nach der interprozeduralen Registerallokation nur mehr 9 Instruktionen (gegenüber 17 Befehlen für die nicht optimierte Version) und überhaupt keine Stack-Variable (anstatt 5 Stack-Variablen vor der interprozeduralen Registerallokation). Allerdings exekutiert das Schnittstellen-Prädikat `split/6` 7 Instruktionen und allokiert eine Stack-Variable, oft kann jedoch direkt das Prädikat `splitINT/6` eingesetzt werden, wenn man berücksichtigt, daß das Callee Saved Register nicht gerettet wird.

Die beiden Klauseln von `split2INT/6` führen weiterhin jeweils 7 Befehle aus, allerdings ist der WAM-Code für dieses Prädikat nicht optimal, denn anstatt das Register 6 für die Variable `Ps` (bzw. für termNs in der zweiten Klausel) zu allokieren, könnte man direkt das Register 1 (bzw. 3) verwenden und somit je eine Instruktion **put_x_value** einsparen.

Zum Aufteilen einer n Elemente langen Liste werden nach der interprozeduralen Registerallokation nur mehr $(9 + 7)n + 4 + 7 = 16n + 11$ SICStus WAM-Instruktionen exekutiert (statt $24n + 4$ Befehlen) und eine einzige Stack-Variable im Hüll-Prädikat allokiert (statt $5n$ Stack-Variablen). Es ergibt sich daher für große Werte von n ein theoretischer Speedup um den Faktor $24/16 = 1.5$, d.h. das optimierte Programm läuft um 50% schneller (unter der Annahme, daß alle WAM-Instruktionen gleich viel Zeit zur Ausführung benötigen). Benchmarks zeigen in der Praxis eine ähnliche Beschleunigung von 50% bis zu 60% (unter Verwendung des Hüll-Prädikats `split/6`. Eine genaue Messung der Maschineninstruktionen, die auf einem Intel Pentium Prozessor zum Aufteilen einer 80-elementigen Liste ausgeführt werden, zeigt einen Speedup um den Faktor 1.509, d.h. das optimierte Programm ist um 50.9% schneller. Bei dieser Messung sind die Maschinenbefehle inkludiert, die zum Generieren der Liste (für beide Versionen identisch) benötigt werden.

6 Benchmarks

In diesem Abschnitt werden Daten aus Benchmarks vorgestellt, die eine genaue Einschätzung der Performance-Gewinne ermöglichen, welche durch eine Calling Convention mit Caller Saved und Callee Saved Registern erzielbar sind. Dabei werden teilweise auch zusätzliche Optimierungen mittels `'SOURCE'/n` und `'SINK'/n` verwendet, die über die Calling Convention hinausgehen.

6.1 Benchmark-Programme

Es folgt eine kurze Beschreibung der Benchmark-Programme, die in dieser Diplomarbeit verwendet werden:

cgat: An Hand des Prädikats `cgats_gctas/2` ist in dieser Diplomarbeit die interprozedurale Registerallokation demonstriert worden. Cgat ist ein deterministisches Benchmark-Programm.

chat: Dieses Programm stammt aus einer Reihe von Benchmarks für den Aquarius Prolog Kompiler; es ermöglicht, englische Sätze zu analysieren. Hier ist unter dem Namen Chat nur der Chat-Parser zu verstehen, nicht das komplette Chat-System, das Anfragen an eine geographische Datenbank in natürlicher Sprache versteht. Dieser Benchmark ist einer der größten, der in dieser Diplomarbeit verwendet wird. Chat ist ein nicht-deterministischer Benchmark, d.h. es wird während der Programm-Exekution Backtracking ausgeführt. Dieser Benchmark analysiert 16 englische Sätze mit Hilfe des Chat-Parsers.

crypt: Dieses Programm ist ebenfalls ein Aquarius-Benchmark, er löst ein einfaches arithmetisches Puzzle. Crypt ist ein nicht-deterministischer Benchmark.

derive: Hierbei handelt es sich um einen Aquarius-Benchmark, der 3 arithmetische Ausdrücke symbolisch differenziert. Das Hauptprädikat dieses Benchmarks verhält sich nicht-deterministisch.

mu: Dieser Aquarius-Benchmark beweist ein Theorem aus Hofstadters „Mu-Math"'. Hierbei handelt es sich um ein nicht-deterministisches Programm.

qsort: Der Quicksort-Algorithmus wird von diesem Aquarius-Benchmark auf eine 50-elementige Liste angewendet. Das Prädikat `qsort/3` verwendet Listen-Differenzen zur Beschreibung der sortierten Liste. Die Verwendung der Listen-Differenzen ist allerdings ungewöhnlich, denn es wird das Ende der Liste vor dem Anfang berechnet. Dadurch kann das mittlere Element mittels der WAM-Instruktion **put_list** (statt durch **get_list**) erzeugt werden, wodurch erreicht wird, daß keine Trail-Überprüfungen im Körper der der Regel `qsort/3` notwendig sind. Das Prädikat `partition/4` ist nicht-deterministisch, obwohl der Algorithmus selbst leicht deterministisch implementiert werden kann.

qsort*: Hierbei handelt es sich um eine umgeschriebene Version des Benchmarks Qsort, um bessere Optimierungen zu ermöglichen. Diese Implementierung des Quicksort-Algorithmus' ist deterministisch. Dieser Benchmark sortiert die gleiche Liste wie Qsort.

queens: Dieser Aquarius-Benchmark löst das sogenannte „8 Königinnen Problem", bei dem 8 Königinnen so auf einem Schachbrett aufzustellen sind, daß sie sich nicht bedrohen. Für dieses Problem gibt es 92 Lösungen, die alle mittels Backtracking ermittelt werden, d.h. Queens ist ein nicht-deterministischer Benchmark.

query: Dieses Programm ist ebenfalls ein Aquarius-Benchmark, der in einer Datenbank, welche die Population und Fläche veschiedener Länder enthält, jene Länder mit in etwa gleicher Bevölkerungsdichte sucht. Hierbei handelt es sich um ein nicht-deterministisches Programm.

serial: Dieser Aquarius-Benchmark berechnet für jedes verschiedene Element in einer 25-elementigen Liste eine eindeutige Nummer. Dieses Programm verhält sich nicht-deterministisch.

sohn: An Hand dieses besonders einfachen Prädikats ist die interprozedurale Registerallokation in dieser Diplomarbeit vorgestellt worden. Das Prädikat `kind_von/2` ist nicht-deterministisch.

split: Das Prädikat `split/5` ist ebenfalls im vorigen Abschnitt zur Demonstration der interprozeduralen Registerallokation verwendet worden. Es handelt sich hierbei um einen deterministischen Benchmark.

sum: Das ist das Beispiel, welches in dieser Diplomarbeit verwendet worden ist, um den Einsatz der beiden Low-Level Prädikate `'SOURCE'/1` und `'SINK'/1` zu demonstrieren. Der Algorithmus zur Berechnung der Summe eines gegebenen Ausdrucks ist eigentlich deterministisch, allerdings wird das von SICStus Prolog nicht erkannt, weshalb sich dieses Benchmark-Programm nicht-deterministisch verhält.

tak-: Auch hierbei handelt es sich um einen Aquarius-Benchmark, der die Takeuchi Funktion mittels rekursiver Arithmetik berechnet. Da die Berechnung von tak(18,12,6) zu lange dauert (die Meßmethode ist sehr ausfwendig, siehe nächster Abschnitt), wird hier nur tak(9,6,3) berechnet. Der Algorithmus ist eigentlich deterministisch, SICStus Prolog erkennt das jedoch leider nicht, weshalb Wahlpunkte angegelgt werden.

tak*: Das ist eine leicht modifizierte, deterministische Version des Benchmarks Tak. Es ist nur ein Cut eingefügt worden, um den Determinismus für SICStus Prolog sichtbar zu machen. Dieses Benchmark-Programm berechnet tak(18,12,6), wie auch die Originalversion des Benchmarks.

ten: Dieser Benchmark dient dazu, das Worst-Case Verhalten abzuschätzen, wenn ein fixer Registersatz verwendet wird. Ten führt Backtracking aus, ohne irgendwelche „sinnvollen" Berechnungen zu machen. Der Prolog-Code für diesen Benchmark sieht folgendermaßen aus:

```
top :-
    ten,
    fail.
top.

ten. ten. ten. ten. ten.
ten. ten. ten. ten. ten.
```

6.2 Meßverfahren

Hier wird beschrieben, auf welche Weise die Daten aus den folgenden Tabellen ermittelt worden sind.

Um die Ausführungszeit von Prolog-Programmen mit SICStus Prolog zu messen, bedient man sich üblicherweise des eingebauten Prädikats `statistics/2`, mit dessen Hilfe man u.a. die Laufzeit seit dem letzten Aufruf dieses Prädikats ermitteln kann. Allerdings sind die Zeitmessungen, die auf diese Art durchgeführt werden, sehr ungenau, denn das Prädikat `statistics/2` liefert die Laufzeit in Millisekunden, die Granularität ist also 1ms. Sehr viele Benchmark-Programme sind allerdings so kurz, daß ihre Ausführung kaum länger als 1ms bis 10ms dauert. Mehrfach durchgeführte Messungen liefern Laufzeit-Unterschiede von bis zu ±100%, derart ermittelte Daten haben keinerlei Aussagekraft. Auf diese Weise können also keine brauchbaren Daten ermittelt werden, es müssen die Benchmarks daher wiederholt exekutiert werden, um Laufzeiten zu erreichen, die viel größer als 1ms sind. Allerdings wird dann der Overhead durch die Wiederholungen mitgemessen, man erhält verfälschte Resultate. Außerdem lassen sich kleine Laufzeit-Unterschiede auf diese Weise nicht ermitteln, da die Meßgenauigkeit einer derartigen Meßmethode zirka bei ±5% liegt. Die Meßergebnisse sind nicht reproduzierbar, sie lassen höchstens Schlüsse auf die Größenordnung der

Performance-Gewinne zu, die durch eine bestimmte Optimierung erzielt werden. Für eine wissenschaftliche Arbeit sind solcherart ermittelte Daten völlig unbrauchbar.

Die Daten für diese Diplomarbeit sind mit einer exakten Meßmethode ermittelt worden, die von U. Neumerkel entwickelt worden war. Dabei wird Emulator von SICStus Prolog nicht direkt ausgeführt, sondern es wird zuerst der GNU-Debugger gdb gestartet, der dann die Exekution von SICStus Prolog überwacht und die genaue Anzahl der ausgeführten Maschinenbefehle ermittelt. Die Benchmarks müssen daher nicht in Schleifen wiederholt ausgeführt werden, man kann mit dieser neuen Meßmethode auch minimale Unterschiede in den verschiedenen Benchmark-Programmen feststellen. Meßdaten, die auf diese Art gewonnen werden, sind exakt reproduzierbar, sofern die verschiedenen Messungen mit derselben Version von SICStus Prolog auf der gleichen Plattform durchgeführt werden. Weiters sollte SICStus Prolog vor jeder Messung neu gestartet werden, um sicherzustellen, daß sich vor jeder Messung das Prolog-System im gleichen Anfangszustand befindet, sonst könnten in Folge von Speicherbereinigungen (Garbage Collection) oder Überlauf-Prüfungen die Meßergebnisse leicht voneinander abweichen. Diese Meßmethode hat allerdings den Nachteil, daß nicht direkt die Ausführungszeit eines Prolog-Programms, sondern die Anzahl der exekutierten Maschineninstruktionen gemessen wird. Ein Rückschluß auf die Ausführungszeit ist nur unter der Annahme möglich, daß alle Maschinenbefehle in etwa gleich viel Zeit benötigen, was bei modernen RISC-Prozessoren meist der Fall ist. Ein weiterer Nachteil diese Meßmethode ist die benötigte Zeit, denn die Ausführung eines Programms durch den GNU-Debugger gdb ist in etwa um den Faktor 20000 langsamer als die direkte Exekution des Programms. Es können auf diese Art also nur kleine Benchmark-Programme ausgeführt werden.

Die Daten in den folgenden Tabellen sind mit SICStus Prolog 3.5p8 (Version 3.5, Patchlevel 8), das unter Linux für einen Intel Pentium Prozessor optimiert kompiliert worden war, ermittelt worden. Auf einem Intel Pentium mit 133MHz Taktfrequenz hat es fast zwei Wochen gedauert, bis alle Daten berechnet waren.

6.3 Performance-Gewinne durch Optimierungen

In diesem Abschnitt wird genau untersucht, welche Performance-Gewinne durch die interprozedurale Registerallokation möglich sind. Bei folgenden Benchmark-Programmen sind Optimierungen durchgeführt worden:

cgat: Die Optimierungen, die an diesem Benchmark durchgeführt worden sind, sind im Detail weiter vorne in dieser Diplomarbeit zu finden.

derive: Hier wurde das zweite Argument des Prädikats `d/3` wie ein Callee Saved Register behandelt.

qsort*: An der deterministischen Version des Quicksort-Algorithmus' ist die Listen-Differenz optimiert worden (Optimierung eines Output-Arguments, siehe [Neu95]).

queens: Bei diesem Benchmark ist ein Callee Saved Register eingeführt worden, um die Allokation von Stack-Variablen im Prädikat `queens/3` aus der ursprünglichen Version zu vermeiden. Nach der Optimierung haben alle Prädikate in diesem Benchmark-Programm die Arität 4.

query: Hier sind die beiden Prädikate `pop/2` und `area/2` optimiert worden (siehe [Neu95]).

serial: Bei diesem Benchmark ist das Prädikat `numbered/3` optimiert worden (siehe [Neu95]). Weiters ist das Prädikat `before/2` um 4 Argumente erweitert worden, die von `split/4` als Callee Saved Register verwendet werden, sodaß keine Stack-Variablen allokiert werden müssen.

sohn: Dieser Benchmark wird zusammen mit den durchgeführten Optimierungen weiter vorne in dieser Diplomarbeit im Detail beschrieben.

split: Die Optimierungen, die an diesem Benchmark durchgeführt worden sind, sind im Detail weiter vorne in dieser Diplomarbeit zu finden.

sum: Dieser Benchmark wird zusammen mit den durchgeführten Optimierungen weiter vorne in dieser Diplomarbeit im Detail beschrieben. Hier wir die ursprüngliche Version des Prädikats `sum/2` mit der ersten optimierten Version verglichen.

sum+: Die Optimierungen, die an diesem Benchmark durchgeführt worden sind, sind im Detail weiter vorne in dieser Diplomarbeit zu finden. Die ursprüngliche Version des Prädikats `sum/2` wird hier mit der zweiten optimierten Version (Akkumulator-Version) verglichen.

tak*: In der deterministischen Version des Benchmark-Programms Tak ist ein Output-Argument optimiert worden.

Die folgende Tabelle zeigt für jeden dieser Benchmarks die Anzahl der Maschineninstruktionen, die das nicht optimierte Programm exekutiert (Spalte „Instr.“), die Anzahl der nach der Optimierung ausgeführten Maschinenbefehle (Spalte „Instr. opt.“), sowie den durch die Optimierung erreichten Speedup:

Benchmark	Instr.	Instr. opt.	Speedup
cgat	10777	9685	1.113
derive	14219	13911	1.022
qsort*	64579	61429	1.051
queens	8333410	7792525	1.069
query	523174	417096	1.254
serial	63452	60410	1.050
sohn	339	297	1.141
split	32223	21355	1.509
sum	13652	10995	1.242
sum+	13652	9012	1.515
tak*	14693660	13055851	1.125

Durch die interprozedurale Registerallokation sind also Beschleunigungen von über 50% möglich; einige Benchmarks, wie beispielsweise das Programm Derive, müßten jedoch umgeschrieben werden, um bessere Resultate zu erzielen.

6.4 Overhead durch fixen Registersatz

Bei deterministischen Prädikaten ist klar, daß die Performance-Gewinne unabhängig von der Anzahl der verwendeten Argumentregister sind, da bei der Exekution dieser Prädikate keine Wahlpunkte erzeugt werden. Im Fall von nicht-deterministischen Prädikaten müssen alle Argumentregister im Wahlpunkt gesichert und beim Backtracking wieder aus Wahlpunkt zurückgeladen werden. Je größer also die Arität eines nicht-deterministischen Prädikats ist, desto mehr Aufwand entsteht beim Backtracking. Während bei deterministischen Prädikaten beliebig viele Argumentregister als Callee Saved Register verwenden werden können, ohne daß dadurch ein Mehraufwand entsteht (auch wenn diese Callee Saved Register nicht ausgenützt werden), muß man bei nicht-deterministischen Prädikaten sehr aufpassen, daß der Overhead beim Backtracking nicht den Performance-Gewinn, den die Verwendung von Callee Saved Registern ermöglicht, völlig zunichte macht, oder noch schlimmer, daß die „Optimierungen“ zu einer Verlangsamung des Programms führen.

In den folgenden Tabellen wird an Hand einiger nicht-deterministischer Benchmarks gezeigt, welchen Overhead die Verwendung von mehr Argumentregistern mit sich bringt. Diese zusätzlichen Register werden nicht verwendet, sie werden in Zielen als Void-Variablen angegeben, in den Klauselköpfen entsprechen die zusätzlichen Argumente anonymen Variablen.

In Spalte „min. Reg.“ wird angegeben, wieviele Argumentregister mindestens verwendet werden sollen, d.h. daß alle Prädikate mit kleinerer Arität um zusätzliche Void-Argumente erweitert worden sind. Bei dieser Erweiterung muß man besonders darauf aufpassen, daß Prädikate, die sich vor der Transformation nur in der Stelligkeit (und nicht im Prädikatsnamen) unterschieden haben, nach der Erweiterung um zusätzliche Argumente nicht mehr voneinander unterscheidbar sind, es müssen also eventuell vor der Transformation Prädikate umbenannt werden. Diese Transformation ist automatisch durch ein Prolog-Programm durchgeführt worden. In der ersten Zeile („min. Reg.“=0) sind die Meßwerte für die unveränderten Benchmark-Programme zu finden.

In den ersten beiden Tabellen sind die ausgeführten Maschineninstruktionen angegeben, in der dritten und vierten Tabelle sind die Verlangsamungen gegenüber der unveränderten Version als Faktor ausgedrückt. Bei den Meßwerten für das Benchmark-Programm Chat ist zu berücksichtigen, daß diese Werte (im Gegensatz zu allen anderen Mewerten) nicht exakt reproduzierbar sind, was wahrscheinlich an Interrupts liegt, die SICStus Prolog verwendet. So läßt sich auch erklären, daß der Meßwert bei Chat in der Zeile für 5 Register besser ist als in jener für 4 Register, was normalerweise nicht vorkommen dürfte. Die deterministische Version Qsort* ist ebenfalls in einer Tabelle aufgenommen worden, um zu veranschaulichen, daß bei deterministischen Prädikaten tatsächlich keine Verlangsamung durch die Verwendung zusätzlicher Argumente auftreten kann. Sieht man sich diese Tabellen spaltenweise von oben nach unten an, so erkennt man auch die Arität jenes nicht-deterministischen Prädikats im jeweiligen Benchmark-Programm, das die kleinste Stelligkeit besitzt, denn sobald mehr Argumente verwendet werden, als dieses nicht-deterministische Prädikat in der unveränderten Originalversion verlangt, entsteht ein Mehraufwand beim Backtracking. Beim Benchmark Queens beispielsweise tritt eine Verlangsamung erst ab 4 Registern auf, d.h. das nicht-deterministische Prädikat mit der kleinsten Stelligkeit hat die Arität 3 (das ist das Prädikat `select/3`).

In der Spalte für Ten kann man den Worst-Case an Verlangsamung durch zusätzliche Argumente ablesen: Bei 5 Registern läuft das ursprüngliche Programm um 23% schneller, bei 8 Registern um 57%, bei 10 Registern um 80%, bei 16 Registern um 148% und bei der Verwendung von 32 Argumentregistern ist das unveränderte Programm um 329% schneller. Der Einsatz von mehr als 8 Argumentregistern scheint also kaum sinnvoll zu sein, da hier die meisten nicht-deterministischen Benchmarks bereits deutliche Performance-Verluste zeigen.

Wird ein fixer Registersatz mit n Argumentregistern verwendet, könnten durch eine geeignete Optimierung des Emulators von SICStus Prolog, sodaß die n Argumentregister nicht in einer Schleife, sondern direkt im Wahlpunkt gespeichert bzw. restauriert werden, die Performance-Verluste deutlich reduziert werden. Die folgenden Messungen sind mit der unveränderten Version 3.5p8 von SICStus Prolog durchgeführt worden, d.h. es sind im Emulator keinerlei Optimierungen vorgenommen worden, um das Sichern und Zurücklesen von Argumentregistern zu beschleunigen.

min. Reg.	chat	crypt	derive	mu	qsort	qsort*	queens
0	20463216	465840	14219	128828	95417	64579	8333410
1	20463229	465840	14219	128828	95417	64579	8333410
2	20465638	466280	14219	128828	95417	64579	8333410
3	20468596	466720	14219	132860	95417	64579	8333410
4	20617641	467160	14563	136044	95417	64579	8378274
5	20576777	470800	14952	146233	99047	64579	8507243
6	21065139	473192	15323	153620	101945	64579	8602570
7	21574531	475584	15694	161007	104843	64579	8697897
8	22155150	477976	16065	168394	107741	64579	8793224
9	22737453	480368	16436	175781	110639	64579	8888551
10	23331465	482760	16807	183168	113537	64579	8983878
11	23933403	485152	17178	190555	116435	64579	9079205
12	24542033	487544	17549	197942	119333	64579	9174532
13	25153415	489936	17920	205329	122231	64579	9269859
14	25764473	492328	18291	212716	125129	64579	9365186
15	26381030	494720	18662	220103	128027	64579	9460513
16	26997587	497112	19033	227490	130925	64579	9555840
17	27614144	499504	19404	234877	133823	64579	9651167
18	28230701	501896	19775	242264	136721	64579	9746494
19	28847258	504288	20146	249651	139619	64579	9841821
20	29463815	506680	20517	257038	142517	64579	9937148
21	30080372	509072	20888	264425	145415	64579	10032475
22	30696929	511464	21259	271812	148313	64579	10127802
23	31313486	513856	21630	279199	151211	64579	10223129
24	31930043	516248	22001	286586	154109	64579	10318456
25	32546600	518640	22372	293973	157007	64579	10413783
26	33163157	521032	22743	301360	159905	64579	10509110
27	33779714	523424	23114	308747	162803	64579	10604437
28	34396271	525816	23485	316134	165701	64579	10699764
29	35012828	528208	23856	323521	168599	64579	10795091
30	35629385	530600	24227	330908	171497	64579	10890418
31	36245942	532992	24598	338295	174395	64579	10985745
32	36862499	535384	24969	345682	177293	64579	11081072

min. Reg.	query	serial	sohn	sum	tak-	ten
0	523174	63452	339	13652	92477	784
1	523187	63452	339	13652	92477	797
2	523195	63452	339	13652	92477	805
3	523411	63452	347	13804	92477	813
4	523627	63452	355	13956	92477	821
5	533218	65278	363	14393	95916	964
6	539059	66588	371	14716	98917	1053
7	544900	67898	379	15039	101918	1142
8	550741	69208	387	15362	104919	1231
9	556582	70518	395	15685	107920	1320
10	562423	71828	403	16008	110921	1409
11	568264	73138	411	16331	113922	1498
12	574105	74448	419	16654	116923	1587
13	579946	75758	427	16977	119924	1676
14	585787	77068	435	17300	122925	1765
15	591628	78378	443	17623	125926	1854
16	597469	79688	451	17946	128927	1943
17	603310	80998	459	18269	131928	2032
18	609151	82308	467	18592	134929	2121
19	614992	83618	475	18915	137930	2210
20	620833	84928	483	19238	140931	2299
21	626674	86238	491	19561	143932	2388
22	632515	87548	499	19884	146933	2477
23	638356	88858	507	20207	149934	2566
24	644197	90168	515	20530	152935	2655
25	650038	91478	523	20853	155936	2744
26	655879	92788	531	21176	158937	2833
27	661720	94098	539	21499	161938	2922
28	667561	95408	547	21822	229145	3011
29	673402	96718	555	22145	231684	3100
30	679243	98028	563	22468	234451	3189
31	685084	99338	571	22791	236852	3278
32	690925	100648	579	23114	239493	3367

min. Reg.	chat	crypt	derive	mu	qsort	qsort*	queens
0	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.000	0.999	1.000	1.000	1.000	1.000	1.000
3	1.000	0.998	1.000	0.970	1.000	1.000	1.000
4	0.992	0.997	0.976	0.947	1.000	1.000	0.995
5	0.994	0.989	0.951	0.881	0.963	1.000	0.980
6	0.971	0.984	0.928	0.839	0.936	1.000	0.969
7	0.948	0.980	0.906	0.800	0.910	1.000	0.958
8	0.924	0.975	0.885	0.765	0.886	1.000	0.948
9	0.900	0.970	0.865	0.733	0.862	1.000	0.938
10	0.877	0.965	0.846	0.703	0.840	1.000	0.928
11	0.855	0.960	0.828	0.676	0.819	1.000	0.918
12	0.834	0.955	0.810	0.651	0.800	1.000	0.908
13	0.813	0.951	0.793	0.627	0.781	1.000	0.899
14	0.794	0.946	0.777	0.606	0.763	1.000	0.890
15	0.776	0.942	0.762	0.585	0.745	1.000	0.881
16	0.758	0.937	0.747	0.566	0.729	1.000	0.872
17	0.741	0.933	0.733	0.548	0.713	1.000	0.863
18	0.725	0.928	0.719	0.532	0.698	1.000	0.855
19	0.709	0.924	0.706	0.516	0.683	1.000	0.847
20	0.694	0.919	0.693	0.501	0.670	1.000	0.839
21	0.680	0.915	0.681	0.487	0.656	1.000	0.831
22	0.667	0.911	0.669	0.474	0.643	1.000	0.823
23	0.653	0.907	0.657	0.461	0.631	1.000	0.815
24	0.641	0.902	0.646	0.450	0.619	1.000	0.808
25	0.629	0.898	0.636	0.438	0.608	1.000	0.800
26	0.617	0.894	0.625	0.427	0.597	1.000	0.793
27	0.606	0.890	0.615	0.417	0.586	1.000	0.786
28	0.595	0.886	0.605	0.408	0.576	1.000	0.779
29	0.584	0.882	0.596	0.398	0.566	1.000	0.772
30	0.574	0.878	0.587	0.389	0.556	1.000	0.765
31	0.565	0.874	0.578	0.381	0.547	1.000	0.759
32	0.555	0.870	0.569	0.373	0.538	1.000	0.752

min. Reg.	query	serial	sohn	sum	tak-	ten
0	1.000	1.000	1.000	1.000	1.000	1.000
1	1.000	1.000	1.000	1.000	1.000	0.984
2	1.000	1.000	1.000	1.000	1.000	0.974
3	1.000	1.000	0.977	0.989	1.000	0.964
4	0.999	1.000	0.955	0.978	1.000	0.955
5	0.981	0.972	0.934	0.949	0.964	0.813
6	0.971	0.953	0.914	0.928	0.935	0.745
7	0.960	0.935	0.894	0.908	0.907	0.687
8	0.950	0.917	0.876	0.889	0.881	0.637
9	0.940	0.900	0.858	0.870	0.857	0.594
10	0.930	0.883	0.841	0.853	0.834	0.556
11	0.921	0.868	0.825	0.836	0.812	0.523
12	0.911	0.852	0.809	0.820	0.791	0.494
13	0.902	0.838	0.794	0.804	0.771	0.468
14	0.893	0.823	0.779	0.789	0.752	0.444
15	0.884	0.810	0.765	0.775	0.734	0.423
16	0.876	0.796	0.752	0.761	0.717	0.403
17	0.867	0.783	0.739	0.747	0.701	0.386
18	0.859	0.771	0.726	0.734	0.685	0.370
19	0.851	0.759	0.714	0.722	0.670	0.355
20	0.843	0.747	0.702	0.710	0.656	0.341
21	0.835	0.736	0.690	0.698	0.643	0.328
22	0.827	0.725	0.679	0.687	0.629	0.317
23	0.820	0.714	0.669	0.676	0.617	0.306
24	0.812	0.704	0.658	0.665	0.605	0.295
25	0.805	0.694	0.648	0.655	0.593	0.286
26	0.798	0.684	0.638	0.645	0.582	0.277
27	0.791	0.674	0.629	0.635	0.571	0.268
28	0.784	0.665	0.620	0.626	0.404	0.260
29	0.777	0.656	0.611	0.616	0.399	0.253
30	0.770	0.647	0.602	0.608	0.394	0.246
31	0.764	0.639	0.594	0.599	0.390	0.239
32	0.757	0.630	0.585	0.591	0.386	0.233

6.5 Vergleich Performance-Gewinne — Overhead

In diesem Abschnitt wird der Overhead, welcher durch den Einsatz einer größeren Zahl an Argumentregistern entsteht, mit den Performance-Gewinnen durch Optimierungen verglichen, die v.a. durch die Verwendung von Callee Saved Registern ermöglicht werden.

In den folgenden Tabellen sind für jene Benchmarks, die im Rahmen dieser Diplomarbeit optimiert worden sind (siehe weiter vorne), die Beschleunigungen bzw. Verlangsamungen gegenüber der Originalversion angegeben, wobei jedes Prädiakt derartig transformiert worden sind, daß die Stelligkeit mindestens so groß ist, wie in der Spalte „min. Reg.“ angegeben. In der ersten Zeile („min. Reg.“=0) sind die Beschleunigungen durch die Optimierungen gegenüber der Originalversion angegeben, wobei keine zusätzlichen Argumente eingeführt worden sind (diese Werte sind bereits weiter oben angegeben worden).

An Hand der Tabellen ist leicht zu erkennen, welche Benchmarks sich deterministisch verhalten, also kein Backtracking durchführen, denn bei diesen Programmen tritt keine Verschlechterung durch den Einsatz einer größeren Zahl an Argumentregistern ein.

Man sieht an diesen Tabellen, daß die Verwendung eines fixen Registersatzes mit mehr als 10 Argumentregistern kaum sinnvoll ist, da hier bei den meisten nicht-deterministischen Benchmarks trotz Optimierungen bereits deutliche Verschlechterungen auftreten. Die optimale Registersatz-Größe scheint bei zirka 8 Argumentregistern zu liegen, da sich bei dieser Register-Anzahl die Performance-Verluste bei nicht-deterministischen Prädikaten in Grenzen halten und weil beim Einsatz von bis zu 8 Argumentregister die zusätzlichen Register oft als Callee Saved Register verwendet werden können, um Datenbewegungen zwischen Registern sowie die Allokation von Stack-Variablen zu minimieren.

Zu bedenken ist noch, daß folgende Messungen mit der unveränderten Originalversion SICStus Prolog 3.5p8 durchgeführt worden sind. Durch entsprechende Optimierungen im Emulator von SICStus Prolog ließe sich der Mehraufwand beim Backtracking durch die zusätzlichen Argumentregister sicherlich reduzieren, sodaß auch ein fixer Registersatz mit mehr als 8 Argumentregistern sinnvoll erscheinen könnte.

min. Reg.	cgat	derive	qsort*	queens	query	serial
0	1.113	1.022	1.051	1.069	1.254	1.050
1	1.113	1.022	1.051	1.069	1.251	1.050
2	1.113	1.022	1.051	1.069	1.251	1.050
3	1.113	1.022	1.051	1.069	1.250	1.050
4	1.113	0.997	1.051	1.069	1.250	1.050
5	1.113	0.971	1.051	1.052	1.222	1.015
6	1.113	0.947	1.051	1.039	1.205	0.996
7	1.113	0.924	1.051	1.027	1.189	0.976
8	1.113	0.902	1.051	1.015	1.174	0.957
9	1.113	0.882	1.051	1.004	1.158	0.938
10	1.113	0.862	1.051	0.992	1.144	0.920
11	1.113	0.843	1.051	0.981	1.129	0.903
12	1.113	0.825	1.051	0.970	1.115	0.887
13	1.113	0.807	1.051	0.960	1.101	0.871
14	1.113	0.791	1.051	0.949	1.088	0.855
15	1.113	0.775	1.051	0.939	1.075	0.841
16	1.113	0.759	1.051	0.929	1.062	0.826

min. Reg.	sohn	split	sum	sum+	tak*
0	1.141	1.509	1.242	1.515	1.125
1	1.141	1.509	1.242	1.515	1.125
2	1.141	1.509	1.225	1.515	1.125
3	1.111	1.509	1.208	1.490	1.125
4	1.083	1.509	1.192	1.465	1.125
5	1.056	1.509	1.148	1.400	1.125
6	1.030	1.509	1.118	1.355	1.125
7	1.006	1.509	1.089	1.313	1.125
8	0.983	1.509	1.062	1.273	1.125
9	0.960	1.509	1.036	1.236	1.125
10	0.939	1.509	1.011	1.201	1.125
11	0.919	1.509	0.987	1.168	1.125
12	0.899	1.509	0.965	1.136	1.125
13	0.881	1.509	0.943	1.107	1.125
14	0.863	1.509	0.923	1.078	1.125
15	0.845	1.509	0.903	1.052	1.125
16	0.829	1.509	0.884	1.026	1.125

7 Andere Optimierungsverfahren

Literatur

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

[BK96] Beckstein, Clemens und Klausner, Joachim. Noch einmal mit Logik. *c't magazin für computer technik*, Verlag Heinz Heise, Deutschland, Oktober 1996.

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

[CDE96] Cervoni, L., Deransart, P. und Ed-Dbali, A. A. *Prolog: The Standard.* Berlin 1996.

[CM87] Clocksin, W. F. und Mellish, C. S. *Programming in Prolog.* Berlin 1987.

[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.

[Hay94] Haygood, Ralph Clarke. Native Code Compilation in SICStus Prolog. *11. ICLP 1994*, MIT Press.

[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.

[Neu95] Neumerkel, Ulrich. Interprocedural register allocation for the WAM based on source-to-source transformations. *JICSLP 95*, MIT Press.

[OKe90] O'Keefe, R. A. *The Craft of Prolog.* The MIT Press, Cambridge, Massachusetts, 1990.

[SS86] Sterling, L. und Shapiro, E. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

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