Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Fachgebiet Wissensbasierte Systeme

Diplomarbeit

Entwicklung eines Compilers für eine Prolog-Erweiterung zur Programmierung paralleler Algorithmen

Sebastian Bogan

Matr.-Nr.: 181894

<borstel@cs.tu-berlin.de>

24.Juli 2003

Betreuer :	Prof. Dr. Erhard Konrad
2.-Gutachter :	Dr. Peter Geibel

Was den Leser erwartet

Prolog ist eine logische Programmiersprache. Verschiedene Aufgabenstellungen lassen sich mit Prolog einfach und elegant lösen. Kommt es allerdings auf Ausführungsgeschwindigkeit an, so sind Lösungen in Prolog oft deutlich langsamer als Umsetzungen in systemnahen Sprachen. Seit langem wird daher versucht, die Ausführung logischer Programme zu parallelisieren. Die Lösungsuche von Prolog eignet sich zwar zum Parallelisieren, ist aber in ihrer Umsetzung nicht trivial und erfordert mehrere Zwischenschritte. Ein bestehendes Prologsystem so zu erweitern, dass damit parallele Algorithmen programmiert werden können, ist nicht ohne tiefgreifendes Verständnis der Interna möglich. In der vorliegenden Arbeit wird daher nach den Grundlagen (Abschnitt 1) ein neuer Pseudointerpreter auf der Basis der Warren Abstract Machine entwickelt (Abschnitt 2, 3 und 4). Der Pseudointerpreter und die von ihm verstandene Prolog-ähnliche Sprache werden so weit verfeinert bis das Strategiespiel 4-Gewinnt beschrieben und günstige Züge berechnet werden können (Abschnitt 5 und 6). Das gewonnene Wissen bildet dann den Ausgangspunkt, die Sprache und den Pseudointerpreter um explizite ODER-Parallelität zu erweitern und die Programmierung paralleler Algorithmen zu ermöglichen (Abschnitt 7). Die Implementation von 4-Gewinnt wird diesbezüglich angepasst. In einer anschliessenden Testreihe auf einem Mehrrechnersystem werden sowohl Lauffähigkeit des entwickelten Systems als auch der erzielte Performancegewinn demonstriert. Die Interpretation der Ergebnisse und der Vergleich des entwickelten Systems mit bestehenden Prolog-Systemen bildet den Abschluss dieser Arbeit (Abschnitt 8).

Der Anhang enthält eine Liste der entwickelten Built-In-Prädikate, den Prolog-Quelltext des Spiels 4-Gewinnt und das Literaturverzeichnis.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Beispiele

1 Grundlagen

1.1 Prolog

Prolog, ein Akronym für PROgrammation en LOGique, ist eine Programmiersprache, welche sich besonders im Umfeld der Künstlichen Intelligenz (KI) etabliert hat. Erfunden und erstmals umgesetzt wurde Prolog durch Alain Colmerauer und Phillipe Roussel an der University of Aix-Marseille um 1971. Der heutige Standard, die Edinburgh Syntax, basiert auf den Arbeiten von David H.D.Warren an der University of Edingburgh. Das von ihm und seinen Kollegen entwickelte DEC-10 Prolog System und dessen weiter verfeinerte Prinzipien sind heute bekannt als Warren Abstract Machine (WAM) und bilden die Grundlage vieler Prolog Implementationen.

Die Programmierung in Prolog unterscheidet sich deutlich von imperativen Sprachen wie C oder Pascal. Ein Prolog-Programm ähnelt einer Datenbank; einer Sammlung von Fakten und Anfragen. Prolog wird als deskriptive Programmiersprache bezeichnet. In einem Prologprogramm wird ein Modell in Form von Sätzen, den Klauseln, beschrieben. Jede dieser Klauseln ist entweder eine Regel oder ein Fakt. Nach der Beschreibung können Fragen, eine dritte Form von Sätzen, an das System gerichtet werden und das System versucht diese im Rahmen der gegebenen Beschreibung zu beantworten.

Jeder Prolog-Fakt oder jede Prolog-Regel ist Teil einer Relation (Prädikat) und beschreibt einen bestimmten Aspekt dieser Relation bzw. setzt sie in Beziehung zu anderen.

Wie jede Programmiersprache folgt auch Prolog einer bestimmten Syntax, d.h. Regeln, Fakten und Fragen müssen nach bestimmten Vorgaben gebildet werden; einer Grammatik folgen. Die Grammatik einer Programmiersprache wird häufig in der Backus-Nauer-Form (BNF) angegeben. In einer solchen BNF sind Regeln definiert, nach denen komplexe Symbole (Nonterminale) durch einfachere Symbole oder Zeichenketten (Terminale) gebildet werden können.[1] Entspricht ein Programm

[1] Genau genommen ist in einer BNF der umgekehrte Weg, also wie aus komplexen Symbolen einfachere Symbole gebildet werden können, beschrieben.

der Grammatik der Sprache, ist es syntaktisch korrekt, so gibt es einen Weg, den Quelltext auf ein einziges Symbol, das Startsymbol, zu reduzieren. Jedes der Symbole (Token) hat einen Namen, einen Begriff, der das entsprechende Konstrukt der Programmiersprache bezeichnet. Da in der Prolog-Literatur zum Teil unterschiedliche Begriffe verwendet werden[2], soll an einem Beispielprogramm (Beispiel 1) die in dieser Arbeit verwendete Terminologie erläutert und zugleich ein Blick auf die Programmiersprache Prolog geworfen werden.

```
studiert(sophie).
studiert(thomas).
weiblich(sophie).
studentin(X) :- studiert(X),weiblich(X).
```

Beispiel 1: Ein typisches Prolog-Programm

Beispiel 1 ist ein vollständiges Prolog-Programm. Es definiert die Fakten "Sophie studiert.", "Thomas studiert." und "Sophie ist weiblich.". Ausserdem ist die Regel "Jeder, der sowohl studiert als auch weiblich ist, ist eine Studentin." enthalten.

Jede Klausel des Programms wird durch einen Punkt terminiert. Die einzelnen Klauseln definieren die Prädikate `studiert`, `weiblich` und `studentin`. `sophie` ist eine Konstante und `X` eine Variable.

In dem Fakt `studiert(sophie).`, ist `studiert` der Funktor und `sophie` das erste Argument. Hätte das Prädikat `studiert` mehr als ein Argument, dann müssten die folgenden Argumente, durch Kommata getrennt, innerhalb der runden Klammern angegeben werden. Ein Fakt kann, wie in anderen Programmiersprachen üblich, als eine Art Funktion mit dem Funktor als Funktionsnamen und den Argumenten als Parameter betrachtet werden.

Unter dem Begriff Struktur versteht man ein Gebilde, bestehend aus einem Funktor und den in Klammern folgenden Argumenten. Strukturen sind rekursiv definiert und können als Argument wiederum Strukturen enthalten.

Als Term bezeichnet man eine einfache Variable oder Konstante aber auch komplexere Gebilde, wie Strukturen.

[2]Insbesondere die Bedeutung des Symbols `Atom` unterscheidet sich stark (vgl. z. B. [15, 4] vs. [6, 5]).

Eine Prologregel wird aus zwei oder mehr Atomen gebildet. Das erste Atom bezeichnet man als Kopf der Regel und alle folgenden bilden den Rumpf. In der Regel `studentin(X) :- studiert(X),weiblich(X).` ist das Atom `studentin(X)` der Kopf und der Rumpf wird durch die Atome `studiert(X)` und `weiblich(X)` konstruiert. Dabei sind Atome häufig Strukturen, können aber auch einfache Konstanten sein. Innerhalb eines Prolog-Programms gilt der Kopf einer Regel genau dann, wenn sein Rumpf gilt. Diese Beziehung ist durch das stilisierte Implikationssymbol `:-` angedeutet.

In einem Interpreter oder als Teil eines compilierten Programms können Fragen an das System gestellt werden. Gibt es einen Weg, Programm (Datenbasis) und Anfrage (Query) durch Substitution in Übereinstimmung zu bringen, zu unifizieren, antwortet das System mit einer positiven Antwort.

Eine interaktive Prolog-Sitzung könnte wie Beispiel 2 aussehen.

```
?- consult(beispiel.1.1).
   yes
?- studiert(sophie).
   yes
?- studiert(peter).
   no
?- studiert(X).
   X=sophie
   yes
?- studiert(X).
   X=sophie;
   X=thomas;
   no
?- studentin(sophie).
   yes
```

Beispiel 2: Eine typische Prolog-Sitzung

Die erste Zeile in Beispiel 2 lädt das Programm aus Beispiel 1 und bestätigt den Erfolg mit `yes`. Prolog beantwortet die zweite Anfrage mit `yes`, da die Anfrage direkt mit einem Fakt der Datenbasis unifizierbar ist; sogar übereinstimmt. Die dritte Anfrage hingegen wird mit `no` beantwortet, da es keinen derartigen Eintrag in der Datenbasis gibt und Prolog davon ausgeht, dass alles was nicht unifizierbar ist,

falsch ist (negation by failure). Die vierte und fünfte Anfrage sind Ergänzungsfragen. Prolog wird aufgefordert, mögliche Bindungen für die Variable `X` zu finden und auszugeben. Durch das `;` nach dem ersten Ergebnis der fünften Anfrage wird der Interpreter aufgefordert nach weiteren Lösungen für `studiert(X).` zu suchen. Das Prolog-System benutzt das Backtrackingverfahren und liefert als zweites Ergebnis `X=thomas`. Eine erneute Suche nach Alternativen wird dann verneint. Die letzte Anfrage, die die Fragen "studiert Sophie" und "ist Sophie weiblich" impliziert, wird wie erwartet mit `yes` beantwortet.

1.2 WAM

Die Idee der Programmiersprache Prolog ist, Sachverhalte in der Sprache der Prädikatenlogik zu beschreiben und dabei das Resultionsverfahren zu verwenden, um zu prüfen, ob eine Aussage, die Anfrage, aus dem Programm folgt.

Aït-Kaci hat die Ideen von David H.D.Warren gesammelt und beschreibt in [5] ein Verfahren, ein Prolog-Programm in eine Sequenz von Befehlen (WAM-Instruktionen) und Parametern umzusetzen. Diese Sequenz ist eine auf das Resolutionsverfahren abgebildete Repräsentation des Prolog-Programms. Die Warren Abstract Machine (WAM) ist eine virtuelle Maschine (Virtual Machine), die durch die WAM-Instruktionen gesteuert wird. Aufgabe der WAM ist es, die Daten des Prolog-Programs in einer baumartigen Struktur zu repräsentieren und das Resolutionsverfaheren darauf anzuwenden. Der Parser für eine WAM bildet jede syntaktisch korrekte Eingabe in eine eindeutige Folge von WAM-Instruktionen ab.

Eine Implementation der WAM organisiert ihren eigenen Speicher in verschiedene kontinuierliche Speicherbereiche, globale Variablen und Register. Zentral für jede WAM sind die Speicher `HEAP` (Heap, Heap-Segment), `CODE` (Code-Segment) und `STACK` (Environment-Stack, Stack-Segment). Der Heap enthält die Repräsentation der Daten. Im Code-Segment sind die Programm-Befehle gespeichert und der Environment-Stack sichert den jeweiligen Kontext.

Jede Anfrage an das System hat eine Wanderung durch das Code-Segment zur Folge. Dabei verändert jeder Sprung den Environment-Stack und jeder Befehl vergleicht bzw. modifiziert die Daten im Heap.

Die Menge der WAM-Instruktionen lässt sich in zwei grosse Klassen unterteilen. Die erste schreibt Daten in den Heap und die zweite vergleicht mit dem Heap. Befehle der ersten Klasse tragen Namen wie `put_variable` oder `put_structure` und die der zweiten Klasse heissen `unify_variable` oder `get_structure`. Obwohl beide den

Heap verändern können, bietet es sich an, sie zunächst als 'schreibend' und 'vergleichend' zu betrachten. Jedes Prolog-Atom wird in eine Folge von Befehlen einer dieser Klassen übersetzt. So werden z.B. Anfragen in 'schreibende' und die Fakten eines Programms in 'vergleichende' Befehlsfolgen übersetzt und abgearbeitet. Wenn eine Anfrage an ein System gestellt wird, werden zunächst die Daten der Anfrage in den Heap geschrieben und danach durch die aus den Fakten des Programms erzeugte Befehlsfolge verglichen; es wird versucht Anfrage und Fakt zu unifizieren.

1.3 Speicher-Repräsentation

In den folgenden Abschnitten wird immer wieder von der Speicher- oder Heap-Repräsentation der Daten gesprochen. Da diese zentral für das Verständnis ist, soll hier ein einfaches Beispiel erläutert werden.

Jedes Prolog-Programm lässt sich als Graph (Abbildung 1) darstellen, wobei jeder Endknoten entweder eine Variable oder Konstante und jede Verzweigung einen komplexeren Term repräsentiert. Für Konstanten gilt, dass sie als Strukturen, bestehend nur aus einem Funktor, betrachtet werden können und daher eigentlich Verzweigungen ohne 'Kinder-Knoten' sind.

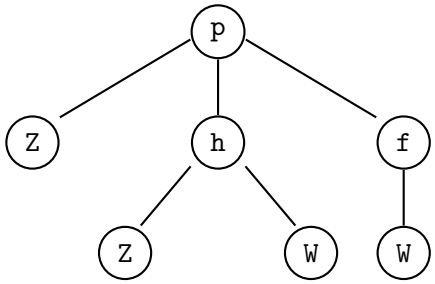

Abbildung 1: Graph für den Term `p(Z,h(Z,W),f(W)).`

Ein Graph wie in Abbildung 1 kann durch eine bestimmte Folge von Speicherzellen abgebildet werden (Tabelle 1). Jede der Zellen ist vom Typ `STR`, `FNC` oder `REF`.[3]

Eine Struktur mit n Parametern wird im Heap durch $n + 2$ Zellen definiert und durch eine einzige Zelle referenziert. Die Definition wird eingeleitet durch einen `STR`-Zelle, welche einen Zeiger auf die folgende `FNC`-Zelle enthält. Diese `FNC`-Zelle enthält Funktor und Anzahl der Parameter (Arity) der Struktur und wird gefolgt

[3] `FNC`-Zellen sind nur der Übersichtlichkeit halber eingeführt und werden in [5] nicht beschrieben.

von n Parameter-Zellen.[4] Im einfachsten Fall ist jede der Parameter-Zellen vom Typ `REF`. Enthält die Struktur jedoch weitere Strukturen als Argumente, so werden diese durch eine einfache `STR`-Zelle, die auf die `FNC`-Zelle der zuvor definierten 'Sub-Struktur' zeigt, repräsentiert. D. h., dass die äusserste Struktur zuletzt und die innerste zuerst definiert werden muss. Jede `FNC`-Zelle einer 'Sub-Struktur' wird dadurch von mindestens zwei `STR`-Zelle referenziert. (Tab. 1 Adressen 0 und 10 bzw. Adressen 4 und 11)

Token	**Adr.**	**Wert**
	0	STR : 1
h(	1	FNC : h/2
Z,	2	REF : 2
W)	3	REF : 3
	4	STR : 5
f(	5	FNC : f/1
W)	6	REF : 3
	7	STR : 8
p(	8	FNC : p/3
Z,	9	REF : 2
h,	10	STR : 1
f).	11	STR : 5

Tabelle 1: Heap aufgebaut durch L0-Query `p(Z,h(Z,W),f(W)).`

Variablen werden durch `REF`-Zellen repräsentiert und zeigen entweder auf sich selbst oder ein vorheriges Auftreten. Konstanten werden als nullstellige Strukturen betrachtet und erhalten ihre eigene `STR`-, `FNC`-Kombination sowie `STR`-Referenz.

[4] Eigentlich enthält die FNC-Zelle nur einen Zeiger auf einen String in einem anderen Speicherbereich.

2 Die Sprachen L0 bis L3

Im letzten Abschnitt wurde die WAM als Maschine, die durch WAM-Instruktionen gesteuert wird, vorgestellt. In diesem Abschnitt sollen, Hassan Aït-Kaci folgend [5], die Sprachen L0, L1, L2 und L3 untersucht werden. Ausgehend von der sehr einfachen Sprache L0 werden sukzessive verschiedene Merkmale von Prolog aufgenommen und so L0 zu einer 'vollständigen' Programmiersprache ausgebaut. Jede Erweiterung des Sprachumfangs soll zunächst in Form einer BNF definiert werden. Die dadurch notwendigen Erweiterungen der WAM-Instruktionen bzw. der WAM werden im Anschluss eingeführt. Anhand von einem jeweils zusammenhängenden Beispiel für 'schreibende' und 'vergleichende' Terme soll die Arbeitsweise der neuen WAM-Instruktionen demonstriert werden. Die Überlegungen in diesem Abschnitt sind nur theoretischer Natur, bilden aber die Grundlage für die Implementation des Parsers (Abschnitt 3) und der Virtuellen Maschine (Abschnitt 4).

2.1 L0

L0 ist eine sehr einfache Sprache. In L0 bestehen Programm und Anfrage aus jeweils nur einem einzigen Term. Die Syntax der Sprache L0 folgt der BNF aus Beispiel 3.

```
program ::=   term .
query ::=     term .
term ::=      variable | functor ( subterms ) | constant
subterms ::=  term | term , subterms
variable ::=  [A-Z][a-zA-Z0-9]*
functor ::=   [a-z][a-zA-Z0-9]*
constant ::=  [a-z][a-zA-Z0-9]*
```

Beispiel 3: BNF der Sprache L0

In der BNF von L0 sind '`(`', '`)`', '`.`' und '`,`' terminale Symbole. `term`, `variable`, `functor`, `constant` und `subterms` sind Nonterminale. '`::=`' und '`|`' sowie die regulären Ausdrücke '`[A-Z][a-zA-Z0-9]*`' und '`[a-z][a-zA-Z0-9]*`' sind Elemente

der Metasprache. '::=' steht für die Definition eines Begriffs und '|' für Alternativen. Worte, die von den regulären Ausdrücken erkannt werden, gehören zu den Terminalen.

In L0 ist ein Term entweder eine Konstante (`constant`), eine Variable (`variable`) oder eine Struktur, bestehend aus Funktor (`funktor`) und einer Menge von Parametern (`subterms`), die selbst wieder Terme sind.

2.1.1 L0-Anfragen

Um die Speicher-Repräsentation der Daten einer L0-Anfrage zu erzeugen, wird der gesamte Term zunächst in seine Bestandteile zerlegt. Dieser Vorgang wird als Registerzuweisung (Register-Allocation) bezeichnet. Dabei werden für Variablen, Konstanten und Strukturen die Register `X1`,...,`Xn` angefordert (alloziert) und den Bestandteilen zugeordnet. Diese Zuordnung erfolgt von links nach rechts und von aussen nach innen, mehrfaches Auftreten derselben Variable werden durch einen Registernamen repräsentiert und damit die Semantik der Anfrage erhalten (Beispiel 4).

```
X1 = p(X2,X3,X4).
X2 = Z
X3 = h(X2,X5)
X4 = f(X5)
X5 = W
```

Beispiel 4: Register-Allocation für die L0-Anfrage `p(Z,h(Z,W),f(W)).`

Eine Anforderung an die Speicher-Repräsentation ist, Subterme zuerst zu definieren und später zu referenzieren. Um dies zu erfüllen, müssen die allozierten Register geordnet werden. Dieser Schritt wird als Linearisieren oder Sequenzialisieren bezeichnet und erfolgt wieder von links nach rechts, jedoch anders als beim Allozieren von innen nach aussen. Die Zuordnung von Registern für Variablen wird in der sequenzialisierten Form nicht mehr explizit angegeben (Beispiel 5).

Aus einem so sequenzialisierten Term lässt sich eine Folge von WAM-Instruktionen ableiten, die eine korrekte Heaprepräsentation erzeugt. Dabei wird die Sequenz in Token zerlegt, und diese in Befehle der WAM übersetzt. Für L0-Anfragen stehen die WAM-Calls `put_structure()`, `set_variable()` und `set_value()` zur Verfügung. Jeder dieser Befehle verändert den Heap, setzt die globale Variable `H` auf die nächste

```
X3 = h(X2,X5)
X4 = f(X5)
X1 = p(X2,X3,X4).
```

Beispiel 5: Sequenz L0-Anfrage `p(Z,h(Z,W),f(W)).`

freie Speicherzelle, und speichert Adressen in den entsprechenden Registern (Tabelle 2).

`put_structure (Name,Arity,Register)`
: Erzeugt eine neue `STR`-Zelle und die direkt darauf folgende `FNC`-Zelle. Die `STR`-Zelle enthält einen Zeiger auf die folgende `FNC`-Zelle und die `FNC`-Zelle enthält Namen und Arity der Struktur. Zusätzlich wird ein Zeiger auf die `STR`-Zelle in dem entsprechenden Register gesichert.

`set_variable (Register)`
: Erzeugt eine neue `REF`-Zelle, die einen Zeiger auf sich selbst enthält (ungebundene Variable) und sichert einen Zeiger in dem entsprechende Register.

`set_value (Register)`
: Erzeugt eine neue `REF`-Zelle, die den Zeiger aus dem mitgelieferten Register enthält.

2.1.2 L0-Programme

Nachdem aus der Anfrage die Heap-Repräsentation der Daten aufgebaut wurde, kann versucht werden, den Programm-Term zu unifizieren. Dafür werden, wie bei einer L0-Anfrage, den Bestandteilen des Terms Register zugeordnet und daraus WAM-Instruktionen abgeleitet. Im Fall der Unifizierbarkeit sind die korrespondierenden Strukturen bereits im Heap etabliert, oder dieser kann entsprechend erweitert werden. D.h. die WAM-Instruktionen überprüfen, ob sich die Strukturen des Programm-Termes und die Register mit denen der Anfrage im Heap in Übereinstimmung bringen lassen. Ein Umsortieren der Register entfällt und die Befehlsfolge lässt sich direkt aus der Register-Zuordnung gewinnen.

Als Pendants zu den aus der Anfrage abgeleiteten WAM-Instruktionen werden die Token eines L0-Programms durch die Befehle `get_structur()`, `unify_variable()`

Token	Befehl	Adr.	Wert	Reg.	H
					0
X3 = h(	put_structure (h/2,X3)	0	STR : 1	X3 = 1	
		1	FNC : h/2		2
X2,	set_variable (X2)	2	REF : 2	X2 = 1	3
X5)	set_variable (X5)	3	REF : 3	X5 = 3	4
X4 = f(	put_structure (f/1,X4)	4	STR : 5	X4 = 4	
		5	FNC : f/1		6
X5)	set_value (X5)	6	REF : 3		7
X1 = p(	put_structure (p/3,X1)	7	STR : 8	X1 = 7	
		8	FNC : q/3		9
X2,	set_value (X2)	9	REF : 2		10
X3,	set_value (X3)	10	STR : 1		11
X4).	set_value (X4)	11	STR : 5		12

Tabelle 2: L0-Anfragen und WAM-Umsetzung `p(Z,h(Z,W),f(W)).`

```
X1 = p(X2,X3,X4).
X2 = f(X5)
X3 = h(X4,X6)
X6 = f(X7)
X7 = a
```

Beispiel 6: Register-Allocation für L0-Programm `p(f(X),h(Y,f(a)),Y).`

und `unify_value()` übersetzt. Jeder dieser Befehle hat einen `READ`-Mode und einen `WRITE`-Mode. Der Mode wird durch den Befehl `get_structur` bestimmt und auf `WRITE` gesetzt, wenn eine bis dahin ungebundene Variable durch eine neue Struktur instantiiert werden muss.

`get_structur (Name,Arity,Register)`

Vergleicht den übergebenen Namen und Arity mit den Einträgen in den entsprechenden (u.U. dereferenzierten) `STR`- und `FNC`-Heapzellen.

Trifft die Funktion auf eine durch die Anfrage ungebundene `REF`-Zelle, kann diese mit der im Programm an dieser Stelle vorgesehenen Struktur unifiziert, d. h. der Heap durch die entsprechende Struktur erweitert und der Zeiger der `REF`-Zelle auf die neue Struktur umgebogen werden.[5] Zusätzlich werden in diesem Fall die Funktionen `unify_variable` und `unify_value` vom `READ` auf den `WRITE`-Modus umgestellt.

`unify_variable (Register)`

`READ`: Setzt den Wert des übergebenen Registers auf die Adresse der nächsten zu untersuchenden Heap-Zelle (`S`).
`WRITE`: wie `set_variable (Register)`

`unify_value (Register)`

`READ`: Versucht Register und Heap zu unifizieren
`WRITE`: wie `set_value (Register)`

Die Ausführung der Befehlsfolge eines L0-Programms beginnt im `READ`-Mode. In `X1` steht eine Adresse, die den Einstieg in den Heap bildet, die `STR`-Zelle der äussersten Struktur. Für den Fall, dass in den `WRITE`-Mode gewechselt werden muss, enthält `H` weiterhin die Adresse der jeweils nächsten freien Heap-Zelle. Die neue globale Variable `S` enthält zu jedem Zeitpunkt die Adresse der nächsten zu untersuchenden Zelle. `S` wird durch `unify_variable()` und `unify_value()` incrementiert und von `get_structur()` gesetzt (Tabelle 3).

Die Abarbeitung der WAM-Instruktionen wird beendet, falls sich die Anfrage und das Programm an einer Stelle nicht in Übereinstimmung bringen lassen. Die WAM kehrt in diesem Fall mit einem `no` zurück.

[5] Das Umbiegen der `REF`-Zelle wird durch die WAM-Interne Operation `bind()` erledigt. Die hier verwendete Operation wird später verfeinert, indem die Indirektion über die zweite Zelle vermieden wird. Stattdessen erfolgt die Bindung direkt. Z.B. wird aus `3 = REF : 3` nicht `3 = REF : 15` sondern direkt `3 = STR : 16`.

Token	Befehl	Adr.	Wert	Reg	S	H	Mode
		0-11	s.o.	X1 = 7		12	READ
X1 = p(	get_structure(p/3,X1)				9		
X2,	unify_variable(X2)			X2 = 9	10		
X3,	unify_variable(X3)			X3 = 10	11		
X4).	unify_variable(X4)			X4 = 11	12		
X2 = f(	get_structure(f/1,X2)	12	STR : 13				
		13	FNC : f/1				
		2	REF : 12			14	WRITE
X5)	unify_variable(X5)	14	REF : 14	X5 = 14	13	15	
X3 = h(	get_structure(h/2,X3)				2		READ
X4,	unify_value(X4)	14	REF : 3		3		
X6)	unify_variable(X6)			X6 = 3	4		
X6 = f(	get_structure(f/1,X6)	15	STR : 16				
		16	FNC : f/1				
		3	REF : 15			17	WRITE
X7)	unify_variable(X7)	17	REF : 17	X7 = 17	5	18	
X7 = a	get_structure(a/0,X7)	18	STR : 19				
		19	FNC : a/0				
		17	REF : 18			20	

Tabelle 3: L0-Programm und WAM-Umsetzung `p(f(X),h(Y,f(a)),Y).`

Nachdem Anfrage und Programm erfolgreich unifiziert worden sind, lassen sich die Substitutionen am Heap ablesen. Die Anfrage `p(Z,h(Z,W),f(W)).` und das Programm `p(f(X),h(Y,f(a)),Y).` können z. B. durch Substitution von `Z = f(f(a))` und `W = f(a)` bzw. `X = f(a)` und `Y = f(f(a))` unifiziert werden. Ausgehend von `p = (X2,X3,X4)` und `X2 = 9` lässt sich das Auflösen der Substitution für `Z` leicht nachvollziehen (Tabelle 4).

Adr.	Wert	Term
9	REF : 2	
2	REF : 12	
12	STR : 13	
13	FNC : f/1	f(
14	REF : 3	
3	REF : 15	
15	STR : 16	
16	FNC : f/1	f(
17	REF : 18	
18	STR : 19	
19	FNC : a/0	a))

Tabelle 4: Substitution für `Z`

2.2 L1

L1 ist eine Erweiterung zu L0. In L1 werden Atome und Argument von Atomen eingeführt. In L1 sollen die 'äussersten' Strukturen als Atome und Strukturen in Argument-Positionen als Terme bezeichnet werden.[6] Ein L1-Programm besteht aus einer Menge von Atomen; einer Menge von Fakten jeweils abgeschlossen durch einen Punkt. Innerhalb eines L1-Programms gibt es keine zwei Atome mit gleichem Namen und Stelligkeit, d.h. jedes Prädikat wird durch genau einen Fakt definiert. Eine Anfrage in der Sprache L1 besteht aus einem einzigen Atom (Beispiel 7).

In L0 war der Umgang mit den WAM-Instruktionen eines Programms relativ trivial,

[6] Obwohl der Begriff Term eigentlich Atome einschliesst, soll er in dieser Arbeit auf Strukturen innerhalb von Atomen begrenzt werden.

```
program :==     atoms .
query :==       atom .
atoms :==       atom | atom , atoms
atom :==        functor ( arguments )
arguments :==   term | term , arguments
term :==        variable | functor ( subterms ) | constant
subterms :==    term | term , subterms
variable :==    [A-Z][a-zA-Z0-9]*
functor :==     [a-z][a-zA-Z0-9]*
constant :==    [a-z][a-zA-Z0-9]*
```

Beispiel 7: BNF der Sprache L1

da es genau einen Term und damit eine Befehlsfolge gab. In L1 jedoch besteht ein Programm aus einer Menge von Atomen, also auch einer Reihe von Befehlsfolgen. In einer entsprechenden WAM werden daher alle Befehlsfolgen in einem gemeinsamen Speicherbereich, dem Code-Segment, abgelegt. Jeder der Befehle ist dort in einer besonderen internen Repräsentation (Opcode) und den dazugehörigen Argumenten (Operanden) gespeichert.[7]

Eine globale Variable `P` (Programmzeiger) enthält zu jeder Zeit die Adresse des nächsten auszuführenden Befehls. Normalerweise bedeutet das, dass `P` in jedem Schritt um die Grösse (Opcode + Argumente) des aktuell auszuführenden Befehls incrementiert wird.

Einige Befehle modifizieren `P` in anderer Weise. Der neue Befehl `call()` setzt `P` auf die Adresse der ersten Instruktion des korrespondierenden Atoms / Prädikats. Funktorname und Arity der Anfrage werden dazu mit den Atomen des Programms verglichen und `P` entsprechend gesetzt. Die Aufgabe Anfrage und Prädikatsnamen zu 'matchen' wird `call()` überlassen. Dazu werden die Prädikatsnamen im Code-Segment gesichert und die Repräsentation Heap entfällt. Die Argumente der Atome werden damit zu den 'äussersten', im Heap gespeicherten, Strukturen. Zugleich verändert sich die Registerzuordnung und der besonderen Bedeutung von Argumenten wird durch die speziellen Register (`A1, ... ,An`) Rechnung getragen. Variablen, die als Argumente auftauchen, bekommen sowohl ein `A-` als auch ein `X`-Register

[7] Als Opcode wird in der Regel eine Zahl, die genau einer WAM-Instruktion zugeordnet ist, verwendet.

zugeordnet. Ein Prädikat mit `i` Argumenten hat hat dann auf jeden Fall die Registerzuordnungen `A1, ... ,Ai` und `Xi+1, ... ,Xn`. (Beispiel 8)

Ausser dem Befehl `call()` wird noch `proceed()` eingeführt, welcher in L1 das Ende der Befehlsfolge des entsprechenden Prädikates anzeigt. Erreicht die Lösungssuche von L1 diesen Punkt, dann gab es einen Weg, Anfrage und Programm zu unifizieren.

2.2.1 L1-Anfragen

Da in der Sprache L1 neben Strukturen auch Variablen in der äussersten Ebene liegen können, werden die bestehenden WAM-Instruktionen um die zwei neuen Befehle `put_variable()` und `put_value()` erweitert. Die aus L1-Anfragen übersetzten WAM-Instruktionen für Variablen in Argument-Position setzen damit auch die entsprechenden `A`-Register (Tabelle 5).

```
X4,A1 = Z
   A2 = h(X4,X5)
   A3 = f(X5)
```

Beispiel 8: Token L1-Anfrage `p(Z,h(Z,W),f(W)).`

`put_variable (X-Register,A-Register)`
: Erzeugt eine neue `REF`-Zelle, die einen Zeiger auf sich selbst enthält (ungebundene Variable) und sichert einen Zeiger in den entsprechenden `X`- und `A`-Registern.

`put_value (X-Register,A-Register)`
: Kopiert den Zeiger aus dem `X`-Register in das `A`-Register.

2.2.2 L1-Programme

Analog zu L1-Anfragen muss die Register-Allocation für L1-Programme angepasst werden (Beispiel 9). Die Menge der zur Verfügung stehenden WAM-Instruktionen wird durch `get_variable()` und `get_value()` erweitert.

Die neuen Befehle `get_variable()` und `get_value()` kümmern sich um Variablen in Argument-Position. Anders als bei `unify_variable()` und `unify_value`, die 'normale' Variablen behandeln, werden hier die `A`-Register berücksichtigt.

Token	Befehl	Adr.	Wert	Reg.
X4,A1=Z	put_variable(X4,A1)	0	REF:0	X4=0 A1=0
A2=h(	put_structure(h/2,A2)	1 2	STR:2 FNC:h/2	A2=1
X4,	set_value(X4)	3	REF:0	
X5)	set_variable(X5)	4	REF:4	X5=4
A3=f(	put_structure(f/1,A3)	5 6	STR:6 FNC:f/1	A3=5
X5)	set_value(X5)	7	REF:4	
	call(p/3)			

Tabelle 5: L1-Anfrage und WAM-Umsetzung `p(Z,h(Z,W),f(W)).`

```
A1 = f(X4)
A2 = h(X5,X6)
A3 = Y
X6 = f(X7)
X7 = a
```

Beispiel 9: Token L1-Programm `p(f(x),h(Y,f(a)),Y).`

`get_variable (X-Register,A-Register)`

Kopiert den Zeiger aus dem `A`-Register in das `X`-Register.

`get_value (X-Register,A-Register)`

Versucht `X`- und `A`-Register zu unifizieren.

Wie in L0 folgt die Abarbeitung des L1-Programms dem Heap-Aufbau der Anfrage (Tabelle 6). Der Heap wird in diesem Beispiel modifiziert und es ergeben sich die notwendigen Substitutionen. Tabelle 7 zeigt den resultierenden Heap.

Token	**Befehl**	**Adr.**	**Wert**	**Reg**	**S**	**H**	**Mode**
	p/3 :	0-7	s.o.	A1 = 0		8	READ
				A2 = 1			
				A3 = 5			
A1 = f(	get_structure (f/1,A1)	8	STR : 9				
		9	FNC : f/1				
		0	REF : 8			10	WRITE
X4),	unify_variable (X4)	10	REF : 10	X4 = 10		11	
A2 = h(	get_structure (h/2,A2)				3		READ
X5,	unify_variable (X5)			X5 = 3	4		
X6),	unify_variable (X6)			X6 = 4	5		
A3 = Y	get_value (X5,A3)	10	REF : 4				
X6 = f(	get_structure (f/1,X6)	11	STR : 12				
		12	FNC : f/1				
		4	REF : 11			13	WRITE
X7)	unify_variable (X7)	13	REF : 13	X7 = 13	6	14	
X7 = a	get_structure (a/0,X7)	14	STR : 15				
		15	FNC : a/0				
		13	REF : 14			16	
.	proceed						

Tabelle 6: L1-Programm und WAM-Umsetzung `p(f(X),h(Y,f(a)),Y).`

Adr.	Wert
0	REF : 8
1	STR : 2
2	FNC : h/2
3	REF : 0
4	REF : 11
5	STR : 6
6	FNC : f/1
7	REF : 4
8	STR : 9
9	FNC : f/1
10	REF : 4
11	STR : 12
12	FNC : f/1
13	REF : 14
14	STR : 15
15	FNC : a/0

Tabelle 7: L1-Heap durch `p(Z,h(Z,W),f(W)).` und `p(f(X),h(Y,f(a)),Y).`

2.3 L2

L2 erweitert die Sprache L1 und ist ein weiterer Schritt in Richtung Prolog. In einem L2-Programm ist es möglich, neben Fakten auch Regeln zu definieren. Eine solche Regel besteht aus einem Regelkopf (Kopf) und einem Regelkörper (Rumpf). Der Regelkopf ist ein Atom, welches genau dann gilt (`:-`), wenn sein Rumpf gilt. Der Rumpf ist eine Liste von Atomen (Teilziele), die von links nach rechts untersucht werden. Beginnend mit L2 wird jeder Fakt oder jede Regel eines Programms als Klausel (`clause`) bezeichnet.

In der BNF von L2 wird die Menge der Nonterminale um `clauses` und `clause` erweitert und das terminale Symbol `:-` eingeführt.

```
program :==    clauses
query :==      atom .
clauses :==    clause .  | clause . clauses
clause :==     atom | atom :- atom | atom :- atoms
atoms :==      atom | atom , atoms
atom :==       functor ( arguments )
arguments :==  term | term , arguments
term :==       variable | functor(subterms) | constant
subterms :==   term | term , subterms
variable :==   [A-Z][a-zA-Z0-9]*
functor :==    [a-z][a-zA-Z0-9]*
constant :==   [a-z][a-zA-Z0-9]*
```

Beispiel 10: BNF der Sprach L2

Die Teilziele einer Regel sind eigentlich eine Verkettung von Anfragen. D. h. passen Regelkopf und ursprüngliche Anfrage zusammen, wird diese temporär durch die einzelnen Teilziele ersetzt und abgearbeitet.

Ähnlich wie in L1 besteht in L2 die Einschränkung, dass es keine zwei Klauseln mit gleichem Namen und gleicher Stelligkeit geben darf. Damit kann es zu jeder Anfrage maximal eine Klausel geben, deren Funktor und Stelligkeit mit denen der Anfrage übereinstimmen. Dadurch ist Backtracking (siehe S. 22) implizit unterbunden und die komplette Anfrage scheitert, falls ein Teilziel nicht erreicht werden kann. Terminieren jedoch alle Teilziele erfolgreich, so sind die Variablen der Anfrage durch die Teilziele gebunden. Dabei kann eine Regel nur terminieren, wenn für jedes Teilziel

wiederum eine passende Regel oder letztlich ein Fakt existiert. Regeln die nur ein Teilziel im Regelkörper besitzen werden als Kettenregeln (chain-rules) bezeichnet.

Der Gültigkeitsbereich von Variablen ist auf Klauseln beschränkt, erstreckt sich aber über alle Teilziele. Variablen, die in mehr als einem Atom vorkommen, werden als permanente Variablen bezeichnet. Sie werden anders als normale Variablen statt in `X`- in `Y`-Registern gespeichert. Die besondere Speicherung ist erforderlich, da die Bindung einer permanenten Variable über die Ausführung eines einzelnen Teilzieles hinaus erhalten bleiben muss. Z. B. könnte die Aufgabe sein, eine Variable im ersten Teilziel einer Regel zu binden und in einem weiteren Teilziele zu verifizieren. Steht die Variable in den beiden Teilzielen an unterschiedlichen Argument-Positionen, so wäre es ohne `Y`-Register unmöglich, die Bindung zu überprüfen.

Da bei der Ausführung einer Regel (Prozedur) weitere Regeln aufgerufen werden können, ist es erforderlich, den jeweils aktuellen Kontext (Register sowie verschiedene WAM-interne Variablen und Zeiger) zu sichern. Eine zu L2 passende WAM verwendet für diese Aufgabe den Speicherbereich `STACK` (Environment-Stack). Der Kontext wird hier zu Beginn einer Regel gespeichert (`allocate()`) und am Ende wieder rekonstruiert (`deallocate()`). Der Ausschnitt des Stacks, der durch einen einzelnen Aufruf von `allocate()` beschrieben wird, heisst Environment-Frame. Ein Environment-Frame mit der Basis-Adresse `E` dient als Arbeits-Fenster für eine Prozedur und schützt ihren Kontext vor Veränderungen durch weitere Aufrufe. Regeln können damit gefahrlos geschachtelt werden. Für jeden Aufruf wird einfach ein neues Arbeits-Fenster angelegt. Kehrt eine Prozedur zurück, wird der zuletzt angelegte Environment-Frame vom Stack entfernt und es gilt wieder der alte Kontext.

Jeder neue Environment-Frame muss die Basis-Adresse des vorhergehenden Arbeits-Fensters (Continuation-Environment, `CE`) und den alten Programmzeiger (Continuation-Pointer, `CP`) sichern. Die `Y`-Register sind, anders als die `X`- und `A`-Register, Teil eines Arbeits-Fensters.

Kehrt eine Prozedur zurück, werden durch den Befehl `deallocate()` die Variablen `E` und `P` aus `CE` bzw. `CP` restauriert und damit der alte Environment-Frame bzw. Kontext wieder hergestellt.

In L2 wird innerhalb einer Regel, statt `proceed()` und `deallocate()` aufzurufen, nur der Befehl `deallocate()` verwendet. Zur Wiederherstellung des alten Programmzeigers bleibt der Aufruf von `proceed()` auf Fakten beschränkt.

`allocate (Number-Of-Permanent-Variables)`

Erzeugt einen neuen Environment-Frame, initialisiert diesen mit den Werten

für `CE`, `CP` und alloziert Platz für die permanenten Variablen.

`deallocate`
: Rekonstruiert die Variablen `E` und `P` und gibt den Speicherplatz des Environment-Frames frei.

`call (Name,Arity)`
: Setzt `CP` auf die Adresse des nächsten Teilziels (`CP = P + Grösse(P)`) und ruft die, dem Teilziel entsprechende, Regel auf (`P = @(p/n)`).

2.3.1 L2-Anfragen

Für L2-Anfragen ergeben sich vom Standpunkt der zu erzeugenden WAM-Instruktionen keine Veränderungen (vgl. BNF 7 und BNF 10). Um jedoch auch für L2 ein zusammenhängendes Beispiel angeben zu können, soll die Anfrage aus Tabelle 8 verwendet werden.

Token	**Befehl**	**Adr.**	**Wert**	**Reg.**
`A1 = a`	`put_structure (a/0,A1)`	`0`	`STR : 1`	`A1 = 1`
		`1`	`FNC : a/0`	

Tabelle 8: L2-Query-Calls `p(a).`

2.3.2 L2-Programme

In L2 werden permanente Variablen durch `Y`-Register repräsentiert und die Register-Allocation muss zuerst für sämtliche Atome einer Regel durchgeführt werden. Der Kopf einer Regel und das erste Teilziel werden dabei als ein Atom betrachtet. D. h. sie verwenden die gleichen Register für lokale und permanente Variablen (`X`- und `Y`-Register) und individuelle Argument-Register (`A`-Register). Alle anderen Atome sind bis auf die gemeinsamen permanenten Variablen unabhängig voneinander (Beispiel 11).

Regeln verwenden in L2 einen eigenen Environment-Frame (`allocate`, `deallocate`) und jedes Teilziel wird von einem `call`-Befehl aufgerufen. Der Kopf einer Regel wird mit dem durch die Anfrage aufgebauten Heap verglichen. Stimmen Funktorname

```
X2,A1 = Z

X2,A1 = Z
Y1,A2 = T

Y1,A1 = T

   A1 = a
   A2 = b

   A1 = b
```

Beispiel 11: Token L2-Programm `p(Z):-q(Z,T),r(T).,q(a,b).,r(b).`

und Stelligkeit überein, wird der Heap durch die Atome des Rumpfes erweitert und die entsprechenden Prozeduren aufgerufen (Tabelle 9).

2.4 L3

L3 erweitert die Sprache L2 durch die Möglichkeit, alternative Lösungswege zu verfolgen. Die BNF verändert sich dadurch nicht. In einem L3-Programm können nun Regeln mit gleichem Namen und gleicher Stelligkeit, aber unterschiedlichem Regelkörper, definiert werden. Das Prolog-System hat dadurch die Möglichkeit, Ziele durch unterschiedliche Regeln zu erfüllen. Scheitert eine Anfrage in einer Regel, wird versucht, eine weitere passende Regel zu finden und dadurch doch noch erfolgreich zu terminieren (Backtracking) . Backtracking eignet sich auch dazu, alternative Antworten zu einer Anfrage zu finden, d. h. obwohl eine Anfrage bereits erfolgreich beantwortet worden ist, nach weiteren Lösungswegen und Bindungen zu suchen.

Ein L3-Programm wird von oben nach unten untersucht.

Wird ein Prädikat, welches durch mehr als eine Klausel definiert ist, aufgerufen, so wird in L3 ein besonderer Environment-Frame, ein Choice-Frame, angelegt und von der globalen Variable `B` referenziert.[8] Ein Choice-Frame enthält einen Zeiger auf die jeweils nächste, noch nicht untersuchte Klausel des Prädikates. Wird zu einem

[8]Die Bezeichnung Choice-Frame wird verwendet, um das Überladen des Begriffs Choice-Point zu vermeiden.

Token	Befehl	Adr.	Wert	Reg	S	H	E	Mode
	p/1:	0-1	s.o.	A1 = 1		2	0	READ
	allocate (1)						1	
X2,A1 = Z	get_variable (X2,A1)			X2 = 0				
X2,A1 = Z	put_value (X2,A1)			A1 = 0				
Y1,A2 = T	put_variable (Y1,A2)	2	REF : 2	Y1 = 2		3		
				A2 = 2				
	call (q/2)							
	q/2:							
A1 = a	get_structure (a/0,A1)				1			
A2 = b	get_structure (b/0,A2)	3	STR : 4					
		4	FNC : b/0					
		2	REF : 3		5			WRITE
	proceed							READ
Y1,A1 = T	put_value (Y1,A1)			A2 = 2				
	call (r/1)							
	r/1:							
A1 = b	get_structure (b/0,A1)				3			
	proceed							
	deallocate						0	

Tabelle 9: L2-Programm-Calls`p(Z):-q(Z,T),r(T).,q(a,b).,r(b).`

späteren Zeitpunkt durch einen Fehler Backtracking ausgelöst, wird der zuletzt erzeugte Choice-Frame angesprungen und die nächste Alternative identifiziert. Der Programmzeiger `P` wird angepasst und die Lösungssuche kann fortgesetzt werden. Sind bis auf eine alle alternativen Klauseln ausgeschöpft, dann wird aus dem Choice-Frame ein normaler Environment-Frame und `B` auf den nächst älteren Choice-Frame zurückgesetzt.

Ein Problem ergibt sich, wenn eine Anfrage in einem komplett anderen Teilbaum scheitert, wenn also der Aufruf eines Teilziels erfolgreich terminiert aber ein späteres Teilziel scheitert. In diesem Fall müssen auch die alternativen Lösungen des ersten Teilziels in Betracht gezogen werden. Das bedeutet, dass das bisherige Verfahren, den Environment-Stack umgehend abzubauen, eingeschränkt werden muss. Erst wenn sämtliche Lösungswege an einem Choice-Point untersucht sind, darf der Stack reduziert werden. Bis zu diesem Zeitpunkt verhält sich `deallocate()` mehr als Markierung der Environment-Frames (Beispiel 12 und Abbildung 2).

```
a :- b(X),c(X).
b(X):-d(X).
d(i).
d(j).
c(j).
?- a.
```

Beispiel 12: Choice-Point Beispiel

Im Beispiel 12 und Abbildung 2 wird `X` zunächst an `i` gebunden (2.) und muss später durch `j` ersetzt werden (5.). Wäre der Stack nach der Unifikation von `X` an `i` abgebaut worden, dann wäre die Anfrage `?-a.` gescheitert. Da aber der Choice-Frame den Abbau verhindert hat, kann die Anfrage doch noch erfolgreich beantwortet werden (3. und 4.).

Nach dem Erzeugen eines Choice-Frames werden u. U. zuvor ungebundene Variablen gebunden. Während des Backtrackings müssen diese Bindungen wieder aufgehoben werden. D. h. es muss vermerkt werden, welche Variablen oder `REF`-Zellen verändert wurden. Das Speichern der Veränderungen der Variablenbindungen wird durch die WAM-interne Operation `trail()` erledigt. `trail()` speichert die Adressen jeder veränderten Variablen in dem neuen Speicherbereich `TRAIL` (Trail). Der Trail ist als Stack organisiert und das Register `TR` enthält die Basis-Adresse (in Trail) aller Veränderungen, die dem entsprechenden Choice-Point folgen. Sollen beim Back-

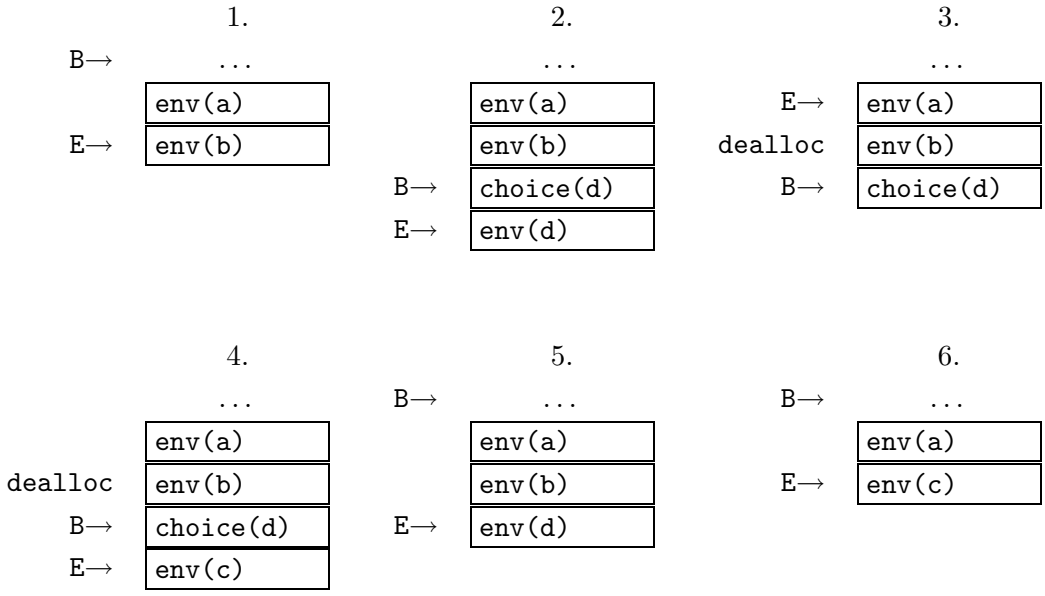

Abbildung 2: Stack-Entwicklung für Choice-Point Beispiel

tracking diese Bindungen wieder aufgehoben werden, werden von `TR` ausgehend, alle vermerkten Zellen zurückgesetzt. Dabei müssen nur Variablen beachtet werden, die auch schon vor dem Erzeugen des Choice-Frames existierten, d. h. deren Heap-Adresse kleiner als `H` zum Zeitpunkt Choice-Frame-Erzeugung ist.

Da ein Choice-Frame Ausgangspunkt für die alternativen Klauseln ist, müssen die Argument-Register (`A`-Register), die aktuelle Environment-Adresse (`E`), der Continuation-Point (`CP`), der vorhergehende Choice-Point (`B`), die Adresse der nächsten alternativen Regel (`L`), das Trail-Register (`TR`) und die Adresse der nächsten unbenutzten Heap-Zelle (`H`) festgehalten werden.

Die neuen WAM-Instruktionen `try_me_else()`, `retry_me_else()` und `trust_me()` verwenden den Choice-Frame als Ausgangspunkt für die Untersuchung der alternativen Klauseln.

`try_me_else (L)`
: Erzeugt einen neuen Choice-Frame und sichert die Werte sowie die Adresse (im Code-Segment) der nächsten Alternative (`L`).

`retry_me_else (L)`
: Restauriert den Inhalt der Register und Variablen und aktualisiert den Wert der nächste Alternative (`L`).

`trust_me`

Restauriert den Inhalt der Register und Variablen und setzt `B` auf den vorhergehenden Choice-Frame.

Neben diesen neuen Befehlen muss `allocate()` so erweitert werden, die unterschiedlichen Offsets von einem vorhergehenden Choice-Frame bzw. regulären Environment-Frame zu berücksichtigen.

In einem L3-Programm sind drei Fälle von Prädikat-Definitionen hinsichtlich der zu erzeugenden Befehlsfolge zu unterscheiden. Gibt es nur eine einzige Klausel für ein Prädikat, verändert sich nichts an der generierten Befehlsfolge gegenüber L2. Gehören zwei Definitionen zu einem Prädikat, wird die Befehlsfolge der ersten Regel durch `try_me_else (L)` eingeleitet und durch `trust_me` mit der zweiten verbunden. Gibt es weitere Alternativen, so werden alle 'Zwischen-Regeln' durch `retry_me_else (L)` verbunden. (Beispiel 13)

```
p/2 : try_me_else L1
      get_variable X3,A1      # X
      get_structure a/0,A2    # a
      procceed
p/2 : retry_me_else L2
      get_structure b/0,A1    # b
      get_variable X3,A2      # X
      procceed
p/2 : trust_me
      allocate 1
      get_variable X3,A1      # X
      get_variable Y1,A2      # Y
      put_value X3,A1         # X
      put_structure a/0,A2    # a
      call p/2
      put_structure b/0,A1    # b
      put_value Y1,A2         # Y
      call p/2
      deallocate
```

Beispiel 13: L3-Befehle `p(X,a).,p(b,X).,p(X,Y):-p(X,a),p(b,Y).`

3 L3 nach WAM Compiler

Nachdem der letzte Abschnitt die Entwicklung der Sprachen L0 bis L3 beschrieben hat, wird in diesem ein entsprechender Compiler besprochen. Dieser Compiler übersetzt ein L3-Programm wie in Beispiel 14, in eine Folge von WAM-Instruktionen, die entweder ausgegeben oder innerhalb der WAM interpretiert werden können (Beispiel 16).[9]

```
p(f(X,a,r(X)),b,Y):-r(X),r(Y),p(X,Y).
p(X,Y,Z).
p(t,U).
p(U,V,W):-r(U).
r(t).
?-p(t,t).
```

Beispiel 14: L3-Programm

Da die zu erzeugenden WAM-Instruktionen nicht nur von den einzelnen Klauseln eines Prolog-Programms, sondern von dem Programm als Ganzes beeinflusst wird, erfolgt die Übersetzung in zwei Schritten. Der erste Schritt umfasst die lexikalische und syntaktische Analyse des kompletten Programms. Durch diese Analyse wird im Speicher eine entsprechende Programm-Repräsentation aufgebaut, die im zweiten Schritt als WAM-Instruktionen ausgegeben werden kann.

3.1 Lexikalische Analyse

Die lexikalische Analyse, der erste Teilschritt auf dem Weg, eine Eingabe in die entsprechenden WAM-Instruktionen zu übersetzen, wird durch ein, aus einem flex-script[10] erzeugtes, Programm erledigt. Die einzelnen Zeichen der Eingabe werden

[9] Beginnend mit diesem Abschnitt soll die Sprache L3 auch als Prolog bezeichnet werden.

[10] flex der 'fast lexical analyzer generator' ist ein GNU-Werkzeug zur Generierung von Programmen die verschiedene Pattern erkennen sollen.

in der lexikalischen Analyse zu elementaren Symbolen (Token und Tokenwerten) zusammengefasst, die dann in der Syntaxanalyse weiterverarbeitet werden können. Die Token sind dabei als reguläre Ausdrücke beschrieben. Überflüssige Zeichen wie Leerzeichen oder Zeilenumbrüche werden direkt gefiltert. (Programm `lex.l`)

3.2 Programm-Repräsentation und Syntaktische Analyse

Es wurde bereits demonstriert, dass sich ein Prolog-Programm als Baum repräsentieren lässt (siehe S. 5). Der in diesem Abschnitt beschriebene Compiler nutzt dies und organisiert die eingelesenen Programm-Teile in einer hirarchischen Speicher-Struktur.

3.2.1 Programm-Repräsentation

In der Programm-Repräsentation, der Speicher-Struktur, besteht ein Programm (`PROGRAM`) aus verschiedenen Prädikaten (`CLAUSEL_DEF`), ein Prädikat aus Klauseln (`CLAUSE`) und eine Klausel aus einem Kopf-Atom (`HEAD`) und den optionalen Rumpf-Atomen (`BODY1` und `BODY`). Jedes Atom ist der Elternknoten zu seinen Argumenten (`ARGUMENT`) und Argumente sind vom Typ Term (`TERM`). Terme schliesslich sind entweder Elternknoten für weitere Terme oder die Blätter des Baumes (`VARIABLE` oder `CONSTANT`) (Abbildung 3).

Jedes Blatt oder Knoten der Struktur ist im Speicher durch ein `t_list`-Element repräsentiert. Alle Kinder eines Knotens sind durch den Zeiger `rn` zu einer Liste verknüpft. Jedes Kind ist durch den Zeiger `parent` mit seinem Elternknoten verbunden und ein Elternknoten zeigt auf das erste seiner Kinderknoten mit dem `child`-Zeiger.

Ausser diesen Zeigern sind in einem `t_list`-Element auch die Anzahl der Kinder (`arity`), der Typ des Knoten (`typ`), der externe Name (`ext_name`), die Register-Namen (`x_register`, `y_register` und `a_register`) und ein Referenz-Zeiger (`ref`) gespeichert (Abbildung 4 und Abbildung 5).

3.2.2 Syntaktische Analyse

Ein von der bereits vorgestellten L3-BNF nur leicht abgewandeltes bison-script[11] übernimmt die Aufgabe der syntaktischen Analyse. Ausgehend von der Reihenfolge

[11] bison ist ein yacc-ähnlicher parser generator aus der Reihe der GNU-Werkzeuge.

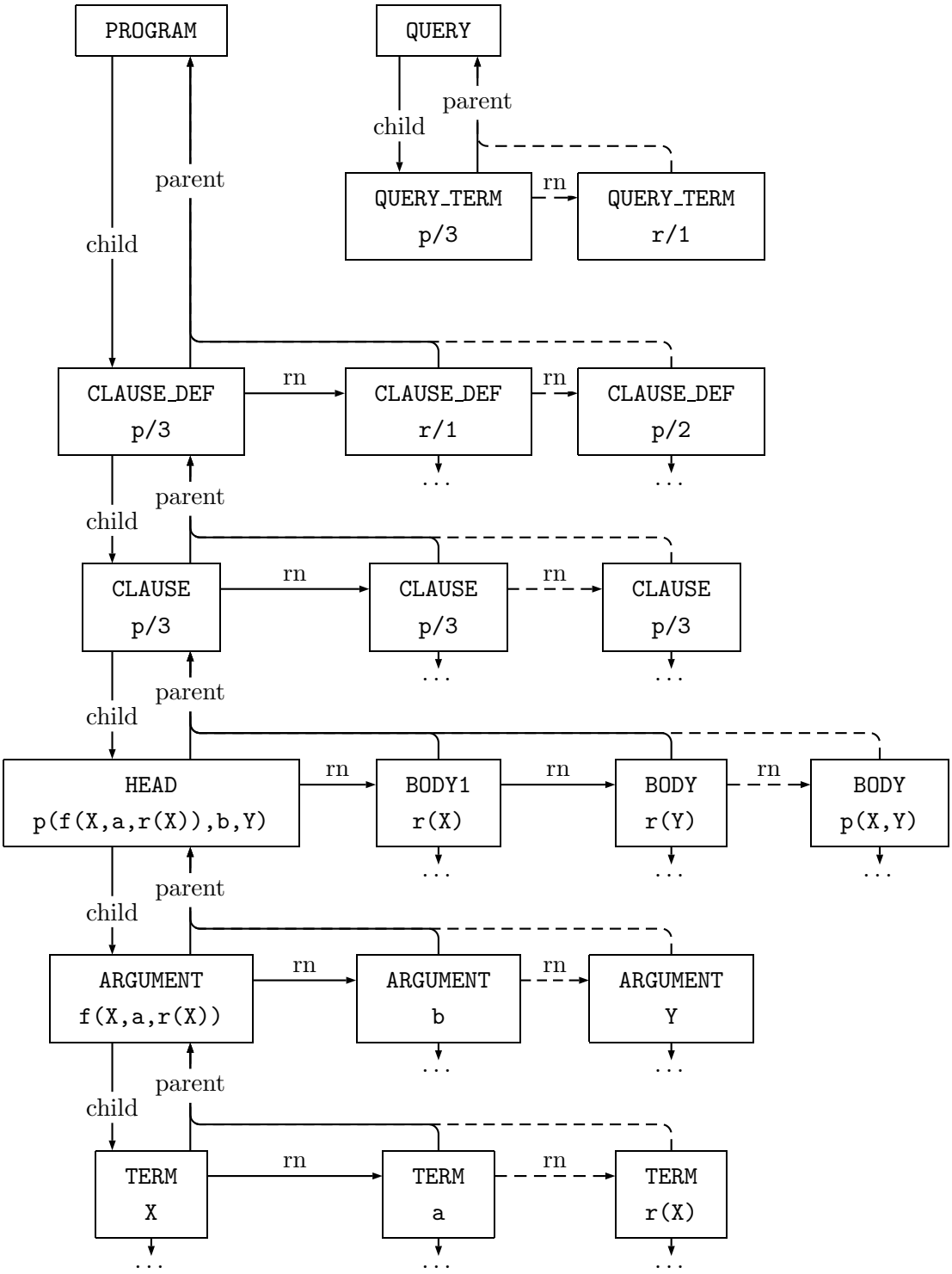

Abbildung 3: L3 to WAM Compiler Speicher für Programm 14

```
typedef struct d_list {
  struct d_list *rn;
  int type;
  char *ext_name;
  int x_register;
  int a_register;
  int y_register;
  int arity;
  struct d_list *child;
  struct d_list *parent;
  struct d_list *ref;
} t_list;
```

Abbildung 4: die Daten-Struktur `t_list`

der identifizierten Token, werden die Funktionen `begin_term()`, `end_term()` und `new_item()` von `yyparse()` aufgerufen und die Programm-Repräsentation erzeugt.

`begin_term (int type)`
: Fügt einen neuen Elternknoten der aktuellen Ebene hinzu. Die im folgenden erwarteten Kinderknoten werden zunächst als Nachbarn (`rn`) angehängt und später eine Ebene nach unten verschoben. Der erwartete Typ (`type`) ist z.B. `CLAUSE` oder `ATOM_BODY1`, Elemente also, die Kinderknoten besitzen und damit Elternknoten einer neuen Ebene sind.[12]

`end_term (int type)`
: Beendet die Liste der Kinderknoten, indem das erste der Kinder vom `rn`-Zeiger auf den `child`-Zeiger umgehängt und der Elternknoten wieder zum aktuellen Knoten erklärt wird. `end_term()` ist Gegenstück oder Abschlussfunktion zu `begin_term()` und erwartet die gleichen Typen (`type`).

`new_item (int type)`
: Erzeugt einen neuen Endknoten in der aktuellen Ebene. Diese Funktion wird

[12] Klauseln werden zu diesem Zeitpunkt nur in einer einfachen Liste verkettet. Erst wenn sämtliche Klauseln eingelesen sind, werden sie durch die Funktion `sort_clauses()` nach Prädikaten sortiert.

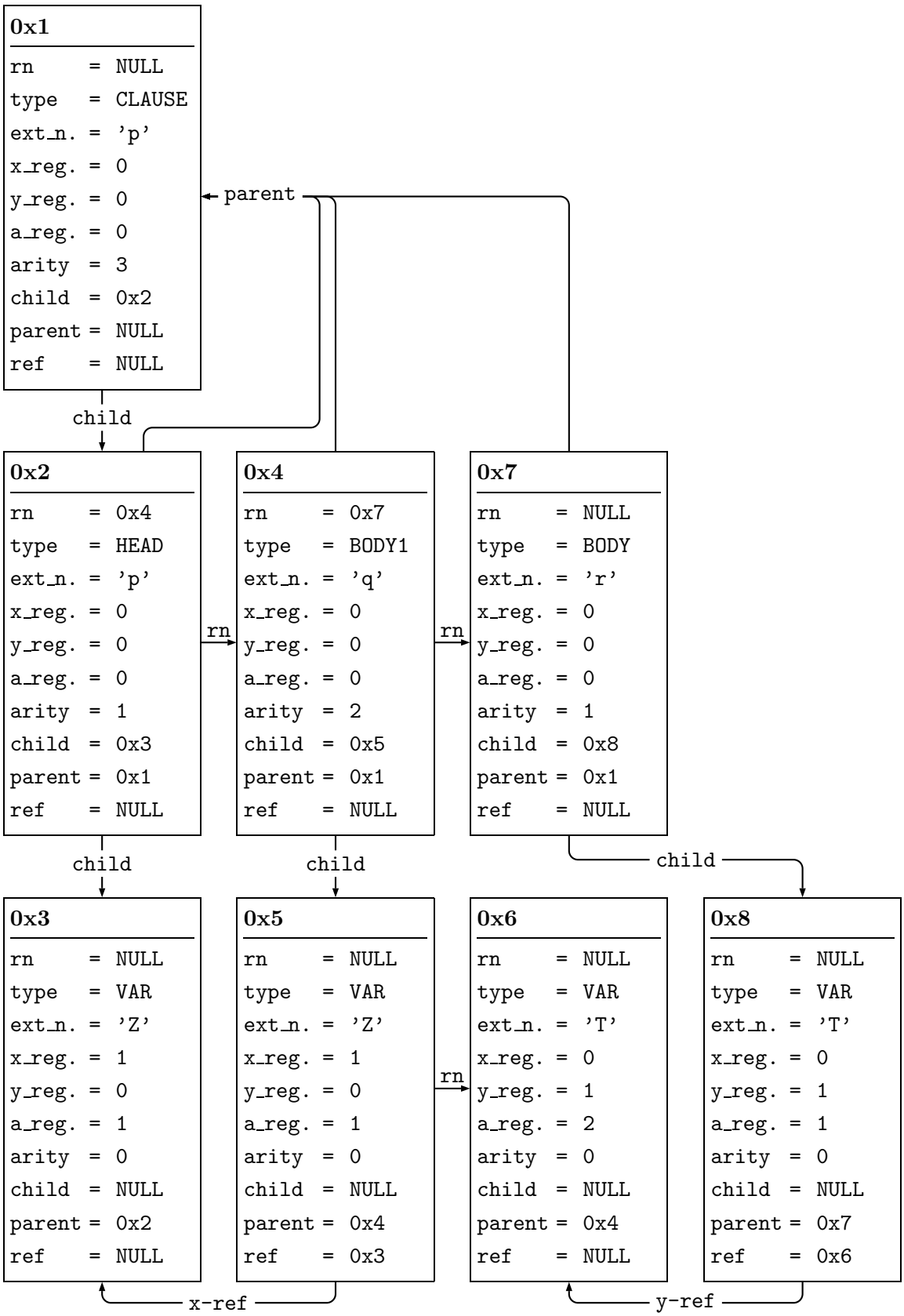

Abbildung 5: `t_list`-Repräsentation für `p(Z):-q(Z,T),r(T).`

verwendet, um die `t_list`-Elemente für einfache Variablen und Konstanten zu erzeugen; Knoten die keine Kinder besitzen.

Für die in Abbildung 5 dargestellte Klausel würde das aus dem bison-script erzeugte Programm die Funktionen `begin_term()`, `end_term()` bzw. `new_item()` in der in Beispiel 15 gezeigten Reihenfolge aufrufen.

```
begin_term(CLAUSE)      # p
 begin_term(HEAD)       #  p/1
  new_item(VARIABLE)    #   Z
 end_term(HEAD)         #  p/1
 begin_term(BODY1)      #  q/2
  new_item(VARIABLE)    #   Z
  new_item(VARIABLE)    #   T
 end_term(BODY1)        #  q/2
 begin_term(BODY)       #  r/1
  new_item(VARIABLE)    #   T
 end_term(BODY)         #  r/1
end_tem(CLAUSE)         # p
```

Beispiel 15: Aufrufe zu Abbildung 5

Nachdem die Repräsentation einer Klausel aufgebaut worden ist, werden die notwendigen Register durch die Funktion `clause_name()` bestimmt und in den `register`-Feldern der `t_list`-Elemente gespeichert. `clause_name()` erledigt dies in zwei Durchläufen. Im ersten Durchlauf (von links nach rechts und von innen nach aussen) werden mehrfach vorkommende Variablen identifiziert und durch `ref`-Zeiger verknüpft. Im zweiten Durchlauf (von links nach rechts, jedoch von aussen nach innen) werden die Registernamen zugeordnet.

Nachdem sämtliche Fakten, Regeln und Anfragen eingelesen und die entsprechenden Registernamen zugeordnet sind, übernimmt es die Funktion `sort_clauses()`, sie nach Prädikaten zu ordnen. Es wird dabei unterschieden, ob es sich um die erste Anfrage, eine weitere Anfrage, die erste Klausel eines neuen Prädikats oder eine weitere Klausel zu einem bereits bestehenden Prädikat handelt. Die von `yyparse()` aufgebaute Speicher-Struktur wird durch `sort_clauses()` vervollständigt (Abbildung 3).

3.3 Ausgabe

Die Ausgabe ist der letzte Schritt des 'L3 to WAM Compilers'. Hier wird die Programm-Repräsentation traversiert und die WAM-Instruktionen für Regeln, Fakten und Anfragen ausgegeben. `output_all()` übersetzt zunächst sämtliche Programm-Prädikate und anschliessend die Anfragen. Für jedes Prädikat werden die Befehlsfolgen der einzelnen Klauseln durch die Ausgabe der WAM-Instruktionen `try_me_else()`, `retry_me_else()` und `trust_me()` verknüpft. Die Funktion `clause_output()` übernimmt es, sich um die WAM-Instruktionen einer Klausel zu kümmern. Um die Atome der Klauseln korrekt zu verarbeiten, unterscheidet `clause_output()` Fakten, Regeln und Anfragen und berücksichtigt ausserdem permanente Variablen. Die einzelnen Atome werden von `output_atom()` an `output_v_atom()` oder `output_w_atom()` als 'schreibende' bzw. 'vergleichende' Atome verteilt und dort die entsprechenden WAM-Instruktionen ausgegeben. Für jede WAM-Instruktion existiert im Modul `output.[ch]` eine eigene Funktion (z. B. `OUT_GET_STRUCTURE()` oder `OUT_PUT_STRUCTURE()`). Gesteuert durch die Variablen `direct` und `out` werden die WAM-Instruktionen auf `stdout` ausgegeben und / oder im Code-Segement der VM (siehe Abschnitt 4) abgelegt. Beispiel 16 zeigt, wie die Ausgabe auf `stdout` aussehen könnte.

```
p/3 : try_me_else L1          p/2 :
allocate 2                    get_structure t/0,A1  # t
get_structure f/3,A1  # f     get_variable X3,A2    # U
unify_variable Y2     # X     proceed
unify_variable X4     # a
unify_variable X5     # r
get_structure b/0,A2  # b     r/1 :
get_variable Y1,A3    # Y     get_structure t/0,A1  # t
get_structure a/0,X4  # a     proceed
get_structure r/1,X5  # r
unify_value Y2  # X
put_value Y2,A1       # X     Query Q0 (p/2) :
call r/1                      put_structure t/0,A1  # t
put_value Y1,A1       # Y     put_structure t/0,A2  # t
call r/1
put_value Y2,A1       # X
put_value Y1,A2       # Y
call p/2
deallocate

L1 (p/3) : retry_me_else L2
get_variable X4,A1    # X
get_variable X5,A2    # Y
get_variable X6,A3    # Z
proceed

L2 (p/3) : trust_me
allocate 0
get_variable X4,A1    # U
get_variable X5,A2    # V
get_variable X6,A3    # W
put_value X4,A1       # U
call r/1
deallocate
```

Beispiel 16: WAM-Instruktionen zum Prolog-Programm aus Beispiel 14

4 Virtuelle Maschine

Abschnitt 2 hat die Warren Abstract Machine theoretisch behandelt. Im letzten Abschnitt wurde auf dieser Grundlage ein Übersetzer von Prolog nach WAM-Instruktionen entwickelt.[13] Nun ist es nötig, das theoretische Wissen auch auf die Interpretation der WAM-Instruktionen anzuwenden. Es soll in diesem Abschnitt eine Virtuelle Maschine (VM) konstruiert werden, die durch die WAM-Instruktionen gesteuert wird. Es wird gezeigt, wie die einzelne Instruktionen implementiert werden können und welchen Anforderungen der Speicher genügen muss. Aus der Verbindung von Parser und VM ergibt sich dann ein lauffähiger Pseudo-Interpreter (Abbildung 6). In einigen kleinen Beispielen sollen die Fähigkeiten und Grenzen des Interpreters vorgestellt werden. Ausserdem werden in diesem Abschnitt die Built-In-Prädikate `cut/0`, `fail/0` und `true/0` implementiert und deren Anwendung, sowie der Umgang mit Listen demonstriert.

4.1 Die Verantwortlichen

Jede der von einer VM verabeiteten WAM-Instruktionen repräsentiert eine Reihe von Veränderungen der Register, Variablen und/oder Speicherbereiche. Die Maschine liest eine Instruktion aus dem Code-Segment und startet die mit ihr verknüpfte Funktion (Handler). Je nach Art des Handlers kehrt die Maschine früher oder später zurück und die im Code-Segment folgende Instruktion kann ausgeführt werden. Die durch die Handler ausgelösten Veränderungen lassen sich dabei in zwei Gruppen unterteilen. Die erste Gruppe liest oder schreibt im Heap-Segment und ist damit direkt mit der Datenrepräsentation und Lösungssuche beschäftigt. Die zweite Gruppe hingegen steuert das Verhalten der Maschine und beeinflusst den Ablauf (Abschnitt 2).

Zentrales Element der VM ist der Programmzeiger. Er bestimmt zu jedem Zeitpunkt, welche Adresse im Code-Segment als nächstes gelesen, also welche Instruktion

[13] Der Übersetzer, der Prolog-Programme in WAM-Instruktionen übersetzt, ist als eigenständiges Programm ein Compiler. Innerhalb der Virtuellen Maschine soll er jedoch als Parser bezeichnet werden.

bzw. welcher Handler als nächstes ausgeführt werden soll. Nachdem das Prolog-Programm und die Anfragen vom Parser übersetzt und im Code-Segment fixiert worden sind, wird der Programmzeiger auf die Adresse der ersten Instruktion der ersten Anfrage gesetzt. Im Folgenden wird der Programmzeiger solange incrementiert und verschoben bis alle Anfragen positiv beantwortet bzw. fehlgeschlagen sind.

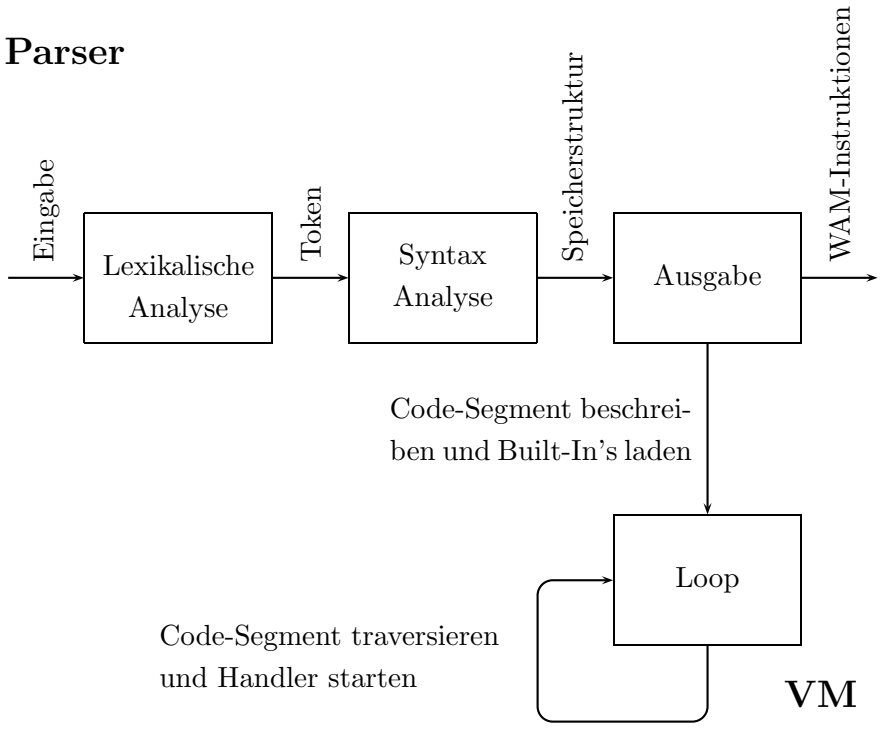

Abbildung 6: der Pseudointerpreter

4.2 Repräsentation im Code-Segment

Der erste Schritt WAM-Instruktionen zu verarbeiten, ist eine geeignete Repräsentation im Code-Segment zu finden. Für jede der Instruktionen ist zu diesem Zweck ein Opcode, eine numerische Repräsentation, im Modul `code.[ch]` definiert. Zusätzlich zur Ausgabe der einzelnen Instruktionen wird ein Makro gestartet und Opcode und Operanden im Code-Segment ablegt. Später, wenn die VM einen Opcode liest, kann sie den verantwortlichen Handler identifizieren und die Kontrolle an ihn übergeben. Der Handler selbst kennt die Anzahl seiner Argumente und liest

diese eins nach dem anderen aus dem Code-Segment. Der Programmzeiger wird dadurch entsprechend erhöht und zum Zeitpunkt, da der Handler zurückkehrt, zeigt der Programmzeiger auf die nächste Instruktion [14].

Die Ausgabe der Klauseln eines Programms erfolgt geordnet. Unabhängig von der Anordnung im Programmcode werden alle Definitionen eines Prädikates hintereinander ausgegeben. Immer wenn mit einem neuen Prädikat begonnen wird, wird der Offset im Codesegment an der durch den Schlüssel des Prädikates definierten Stelle im Array `PREDLABEL` vermerkt. Später können auf diese Weise die Offsets der einzelnen Prädikate schnell ermittelt werden. Der Handler für die WAM-Instruktion `call()` z. B. nutzt dies, um den Programmzeiger ensprechend dem übergebenen Schlüssel zu verschieben.

4.3 Die Speicher der WAM

Die Handler der einzelnen WAM-Instruktionen lesen und schreiben in verschiedenen VM-eigenen Speicherbereichen. Diese Speicher repräsentieren Zustand und Programm der Maschine.

Code Das Code-Segment wird vor dem eigentlichen Start der VM mit den Opcodes und Operanden der einzelnen Instruktionen beschrieben. Der Programmzeiger ist ein Zeiger auf eine Adresse im Code-Segment und bestimmt den Ablauf.

Heap Im Heap wird eine Repräsentation der Daten erstellt und verändert. Existiert eine Lösung zu einer Anfrage, so sind nach der Bearbeitung die notwendigen Unifikationen in Form von Referenzen ablesbar.

Stack Während der Suche nach Lösungen müssen u. U. Probleme in Teilprobleme zerlegt und Alternativen in Betracht gezogen werden. Der Stack ist das Segment, in dem die Verwaltung der Teilaufgaben repräsentiert ist. Die Grösse des Stack-Segmentes ist direkt proportional zur gegenwärtigen Inferenztiefe.

Trail Bei der Verarbeitung von Alternativen ist es eventuell erforderlich, Variablenbindungen rückgängig zu machen. Der Trail ist der Speicherbereich, in dem die bei der Untersuchung der Möglichkeiten vorgenommenen Veränderungen vermerkt werden.

[14] Es sei denn, der Handler hat die Aufgabe, den Ablauf des Programmes in anderer Weise zu beeinflussen.

PDL Die 'Push-Down-List' ist ein vom Unifikationsalgorithmus verwendeter Speicherbereich. Sollen komplexe Terme unifiziert werden, werden diese in Subterme zerlegt, temporär gespeichert und von links nach rechts untersucht.

PREDLABEL Dieses Segment dient dazu, den Schlüsseln der einzelnen Prädikate Offsets im Code-Segment zuzuordnen.

SUBS In den meisten Fällen ist es wünschenswert, bei erfolgreicher Bearbeitung einer Anfrage, neben einem einfachen '`yes`' auch die entsprechende Bindung der Variablen anzugeben. Im Speicherbereich `SUBS` werden dazu Namen und Heapadressen der Variablen der Anfrage gespeichert und durch die zusätzliche WAM-Instruktion `show_substitution()` am Bildschirm ausgegeben.

Die Implementation der Handler für die einzelnen WAM-Instruktionen geht von kontinuierlichen Speicherbereichen aus. Die Segmente werden als Stack oder Array von Speicherzellen organisiert. Dabei ist der Umfang der einzelnen Bereiche erst zur Laufzeit bekannt. Die Speicher-Segmente Stack und Heap müssen je nach Problem enorm gross sein. Um dieser Anforderung gerecht zu werden, wird durch die Module `mem.[ch]` sowie `starray.[ch]` eine Art Speicher-Management zur Verfügung gestellt. Die Idee ist, statt einem grossen zusammenhängenden Speicherbereich, je nach Bedarf, kleinere Speicherseiten zu allozieren. Eine Reihe von Funktionen werden angeboten, die die notwendigen Adressumrechungen vornehmen und einen einheitlichen Zugang zum Speicher des Betriebssystems schaffen.

Das Modul `mem.[ch]` verwaltet eine Liste aller Segmente (Heap, Stack, Trail usw.). Für jedes der Segmente existiert ein Array, in dem die Offsets zu den einzelnen Speicherseiten dieses Segmentes vermerkt sind. Jede dieser Seiten hat einen Umfang von 1024 Zellen zu je 4 Byte. Soll eine bestimmte Zelle bearbeitet werden, so entspricht der ganzzahlige Teil der Divison, von virtueller Adresse und 1024, der Seitennummer. Der Rest der Division ergibt den Offset der Zelle innerhalb einer Seite. Addiert man diesen Offset zu dem im Array gespeicherten Offset der Seite, so erhält man die physische Adresse [15] der Zelle. Neben der Adressberechnung übernimmt es das Modul `mem.[ch]`, neue Seiten zu allozieren oder wieder freizugeben.

Da der Umgang mit dem Speicher auf verschiedenen Plattformen sehr unterschiedlich gehandhabt wird und von der VM die stack- und array-typischen Funktionen

[15] Da das Betriebssystem seinen Speicher in ähnlicher Weise verwaltet, handelt es sich nicht wirklich um eine physische Adresse. Auf Programmebene soll davon jedoch abstrahiert werden.

erwartet werden, wird das Modul `starray.[ch]` als Interface verwendet. Hier kapseln Funktionen wie `starray_push()` und `starray_pop()` bzw. `starray_get()` und `starray_change()` den Zugriff auf die einzelnen Segmente.

Die Grösse einer virtuellen Adresse ist mit der Zellgrösse auf 4Byte begrenzt. Damit ergibt sich ein Adressraum von $2^{32} = 4.294.967.296$ Zellen bzw. 16GByte Speicher pro Segment. Für das Heap-Segment gilt die Einschränkung, dass die niederwertigsten 4 Bits den Typ der Zelle repräsentieren und dadurch die Anzahl der möglichen Addressen auf $2^{28} = 268.435.456$ beschränkt wird. Die maximale Heap-Grösse liegt damit bei 1GB.

4.4 Erste Resultate

Nachdem gezeigt ist, wie sich Parser und VM verbinden lassen, wie dabei der Speicher organisiert werden kann und wie der Programmzeiger den Ablauf bestimmt, bleibt nun die Aufgabe, die einzelnen Handler zu implementieren. Deren Arbeitsweise und Funktionen sowie Zusammenspiel mit Variablen und Registern ist bereits bei der Einführung der Sprachen L0 bis L3 angedeutet worden und in [5] als Pseudocode nachzulesen. Bis auf wenige Einschränkungen ist die dort vorgestellte Umsetzung übernommen worden und soll hier nicht weiter beschrieben werden (siehe Modul `sec:heap.[ch]`, um die Implementation nachzulesen). Einzig der Umgang mit freigegebenen Zellen unterscheidet sich. Während in [5] die jeweils nächste freie Zelle berechnet wird, verwendet die vorliegende Implementation eher eine Stack-ähnliche Struktur. Zellen werden 'echt' freigegeben und die Grösse der Segmente durch die Funktion `discard_mem()` ständig angepasst. Die Funktion `starray_size()` liefert zu jedem Zeitpunkt die Adresse der nächsten freien Zelle.

Sind die Handler einmal implementiert, hat man einen lauffähigen Interpreter, der die in der Sprache L3 beschreibbaren Probleme nach dem Resolutionsverfahren löst.[16]

Das Beispiel 17 demonstriert eine einfache Anwendung des Interpreters, bei dem anhand von Fakten und Regeln der '`moerder`' ausfindig gemacht wird.

Die Funktion `main()`, aus dem Modul `main.[ch]`, lädt via Kommandozeilenschalter den Prolog-Quelltext aus einer Datei und übersetzt diesen in WAM-Instruktionen.

[16] Die Ergebnisse der vorliegenden Arbeit sind in den ausführbaren Dateien `cpl` bzw. `cplparallel` vereint. In allen folgenden Tests wird eines der beiden Programme, in der zum Testzeitpunkt erreichten Ausbaustufe, verwendet.

```
verdaechtig(buttler).          % Der Buttler ist verdächtig.
verdaechtig(gaertner).         % Der Gaertner ist verdächtig.
moerder(X):-                   % Jeder der
        motiv(X),              %  ein Motiv hat,
        tatort(X),             %  am Tatort war und
        luegt(X).              %  luegt ist ein Moerder.
tatort(gaertner).              % Der Gaertner war am Tatort.
tatort(buttler).               % Der Buttler war am Tatort.
motiv(X):-                     % Jeder der Verdächtig ist,
        verdaechtig(X).        %  hat ein Motiv.
motiv(ehemann).                % Der Ehemann hat ein Motiv.
luegt(buttler).                % Der Buttler luegt.
luegt(ehemann).                % Der Ehemann luegt.

loesung():-
        write('der Moerder ist der : '),
        moerder(X),
        write(X).
?-loesung().
```

Beispiel 17: Mörderrätsel

Die WAM-Instruktionen werden ausgegeben und zugleich im Code-Segment etabliert. Der Interpreter untersucht die Anfrage und kommt zu der im Beispiel 18 gezeigten Lösung.

```
[bozo@foo] ./cpl  -f ../example/moerder2.pl
REGISTER_PREDICATE(17,"verdaechtig",1);
TRY_ME_ELSE(1,1);
GET_STRUCTURE_A("buttler",0,1);
...
DEALLOCATE();
REGISTER_QUERY(1,"loesung",0);
CALL(22);
SHOW_SUBSTITUTION();

der Moerder ist der : buttler
Yes

[bozo@foo]
```

Beispiel 18: Ausgabe zu Beispiel 17

4.5 Essentielle Erweiterungen

Beispiel 17 und 18 demonstrieren, mit Ausnahme des Built-In-Prädikates `write()`, das reine Resolutionverfahren. Oft ist es jedoch vorteilhaft oder sogar notwendig, die Lösungssuche, anders als allein über die Reihenfolge der Klauseln, beeinflussen zu können. Drei wichtige Built-In-Prädikate sind `cut/0`, `fail/0` und `true/0`

Der Cut bietet die Möglichkeit, das Backtracking an bestimmten Stellen ausser Kraft zu setzen. Der Suchraum lässt sich damit gezielt reduzieren. Sämtliche Choice-Points der aktuellen Klausel und des Prädikates, dem die Klausel angehört, werden verworfen. Backtracking führt dann entweder zu, den von dem Cut unbeeinflussten Choice-Points oder schlägt fehl. Der Cut ist in dieser Arbeit als Built-In implementiert und wird durch die WAM-Instruktion `call()` wie ein benutzerdefiniertes Prädikat behandelt. Die Umsetzung des Cut-Handlers ist relativ einfach und in [5] beschrieben.

Die Built-In-Prädikate `fail/0` und `true/0` beeinflussen die Lösungssuche, indem

sie die globale Variable `FAIL` direkt setzen und, im Fall von `fail/0`, Backtracking auslösen. Die Implementation von `true/0` ist ein einfaches `NOP` und `fail/0` ruft die Backtracking-Routine `backtrack()` auf.

Trotz der relativ einfachen Implementation wird die Bedeutung der Prädikate, z. B. für die Negation oder beim Umgang mit Listen, deutlich. Im Beispiel 19 bzw. 20 werden alle möglichen Ursprungslisten gefunden, die verbunden, die Liste `l(1,l(2,l(3,l(4,l(5,[])))))` ergeben würden. Die Ausgabe aller Lösungen wird durch `fail/0` erzwungen.[17]

```
listJoin([],L,L).
listJoin(l(X1,Y1),L2,l(X1,Z1)) :-
        listJoin(Y1,L2,Z1).
loesung():-
        listJoin(Y,X,l(1,l(2,l(3,l(4,l(5,[]))))),
        write('Liste 1 = '),write(Y),nl(),
        write('Liste 2 = '),write(X),nl(),nl(),fail().
?-loesung().
```

Beispiel 19: Steuern der Lösungssuche

Ersetzt man im Beispiel 19 den Fakt `listJoin([],L,L).` durch die Kettenregel `listJoin([],L,L):-!.`, so würde die Lösungssuche nach der ersten Lösung terminieren.[18] Dabei würde `fail/0` nach wie vor das Flag `FAIL` setzen und Backtracking auslösen, die alternativen Lösungen wären aber durch den Cut abgeschnitten. Damit würde nur die erste Lösung ausgegeben und danach die Suche mit der Ausgabe `'no'` beendet.

Neben der Möglichkeit, Backtracking zu erzwingen bzw. Choice-Points abzuschneiden, lassen sich durch die Kombination von `cut/0` und `fail/0` andere wichtige Built-In-Prädikate beschreiben. Beispiel 21 zeigt wie eine 'Prädikat-gebundene' Negation beschrieben werden kann.

[17] Die Zeichenkette `'[]'` wird vom Parser als besondere Zeichenkette zur Repräsentation der leeren Liste erkannt und als Konstante interpretiert. Die in [5] vorgeschlagene Optimierung für den Umgang mit Listen ist bislang nicht implementiert. In allen Beispielen dieser Arbeit werden Listen durch Terme wie `l(_,_)` konstruiert. Das Built-In-Prädikat `nl()` wird in Kombination mit `write()` verwendet um einen Zeilenvorschub auf `stdout` auszugeben.

[18] Das Symbol '`!`' ist eine alternative Notation für das Built-In-Prädikat `cut()`.

```
[bozo@foo] ./cpl  -f ../example/opt.pl
...
Liste 1 = []
Liste 2 = l(1, l(2, l(3, l(4, l(5, [])))))
Liste 1 = l(1, [])
Liste 2 = l(2, l(3, l(4, l(5, []))))
Liste 1 = l(1, l(2, []))
Liste 2 = l(3, l(4, l(5, [])))
Liste 1 = l(1, l(2, l(3, [])))
Liste 2 = l(4, l(5, []))
Liste 1 = l(1, l(2, l(3, l(4, []))))
Liste 2 = l(5, [])
Liste 1 = l(1, l(2, l(3, l(4, l(5, [])))))
Liste 2 = []
No
[bozo@foo]
```

Beispiel 20: Ausgabe zu Beispiel 19

```
notluegt(X):-luegt(X),!,fail().
notluegt(X):-true().
```

Beispiel 21: Mögliche Implementation der Negation (\+luegt)

5 Built-In-Prädikate

Im Allgemeinen wird die Qualität von Prolog-Systemen nach der Zahl der Built-In-Prädikate bewertet. Erst durch sie wird Prolog zu einer alltagstauglichen Programmier-Sprache. Die Implementation der VM lässt sich relativ einfach um neue Built-In-Prädikate erweitern. Anhand eines Beispiels soll in diesem Abschnitt das genaue Vorgehen erläutert werden. Nach diesem Muster können dann eine Reihe wichtiger mathematischer Operationen und Built-In-Prädikate hinzugefügt werden.

5.1 Wozu Built-In-Prädikate

Die auf der Basis der WAM enstandene Programmiersprache erlaubt es, Probleme so zu beschreiben, dass, falls es eine Lösung gibt, die VM die entsprechende Unifikation findet und erfolgreich terminiert. Oder, wenn es keine Lösung gibt, nach Untersuchung sämtlicher Alternativen, abbricht bzw. unendlich weiter arbeitet. Die Grundlage der Lösungssuche sind das Resolutionsverfahren und der Unifikationsalgorithmus. Damit lassen sich theoretisch alle Probleme formalisieren. Dennoch wäre die Beschreibung bestimmter Zusammenhänge nur sehr umständlich möglich. Mathematische Beziehungen könnten z. B. als Fakten dargestellt werden. Sehr viel einfacher ist es jedoch, die Programmiersprache durch entsprechende mathematische Built-In-Prädikate zu erweitern.

Neben der Mathematik gibt es eine Reihe weiterer Aufgaben, die durch Built-In-Prädikate eleganter gelöst werden können. Oft ist es z. B. notwendig, die Lösungssuche zu beeinflussen oder das Benutzerinterface zu erweitern.

Seit 1995 gibt es einen ISO-Standard [19], der die Grundbausteine einer vollständigen Prolog-Programmiersprache definiert. Ein Teil der dort beschriebenen Built-In-Prädikate sind auch in dieser Arbeit umgesetzt. Unter anderen gehören dazu die bereits vorgestellten nullstelligen Prädikate `cut/0`, `fail/0` und `true/0`, Prädikate zur Ausgabe von Termen und Speicherbereichen der WAM, mathematische Prädikate,

[19] ISO/IEC 13211-1:1995 Information technology – Programming languages – Prolog – Part 1: General core

Listen-Prädikate und Extralogische Prädikate wie `findall4/4` oder `assert/2` und `retract/2`. Eine Beschreibung aller implementierten Built-In-Prädikate ist in Abschnitt A zu finden. Diese Prädikate reichen für die in dieser Arbeit angestrebten Ziele aus. Dennoch lassen sich weitere Prädikate ohne grossen Aufwand hinzufügen. Die folgenden Teilabschnitte beschreiben an dem Beispiel 'grösser als' (`gt/2`) wie ein solches Built-In-Prädikat hinzugefügt werden kann. Diese Anleitung soll zum einen bei der Erweiterung des Interpreters und der von ihm verstandenen Sprache helfen und zum anderen Einblick in die Funktions- und Arbeitsweise der VM geben [20].

5.2 Zahlen und Mathematik

Noch bevor das Prädikat `gt/2` implementiert werden kann, ist es notwendig, sich über die Repräsentation von Zahlen und Mathematik Gedanken zu machen.

In den meisten Programmiersprachen werden Zahlen in Variablen oder Konstanten gespeichert, deren Typ festgelegt ist. Jeder der zur Verfügung stehenden Typen belegt eine bestimmte, je nach Plattform unterschiedliche, Anzahl von Bytes. D. h. unabhängig vom Wert der Variablen wird genau die vorgesehene Speichermenge verwendet, um die Zahl zu repräsentieren. Dadurch ist die Grösse der Zahlen bzw. deren Genauigkeit begrenzt.

Dem Beispiel von Common Lisp folgend, wird in dieser Implementation ein anderer Ansatz verfolgt. Statt Zahlen zu typisieren, werden sie als normale Konstanten behandelt und nur bei der Verwendung in mathematischen Operationen umgewandelt. Damit existiert, abgesehen von dem vom Betriebssystem zur Verfügung gestellten Speicher, keine Begrenzung in der Grösse und Genauigkeit der Zahlen. Als Konsequenz ergibt sich allerdings auch, dass die durch die WAM-Implementierungssprache C zur Verfügung gestellten mathematischen Funktionen nicht verwendet werden können. Stattdessen müssen die Ziffern der Terme extrahiert und Operationen Ziffer für Ziffer durchgeführt werden. Die Umwandlung und Berechnung ist zwar relativ langsam aber dafür beliebig genau. In dieser Arbeit ist Henrik Johansson's bignum-Paket (Version 1.2) über das Interface `ninterface.[ch]` eingebunden. Alle mathematischen Operationen auf dem Typ `bignum` werden von dort importiert [21].

[20] Das Beispiel `gt/2` ist auch gewählt worden, da dafür, wie auch für eine Reihe anderer Prädikate, ein eigener Handler implementiert werden muss. Viele Built-In-Prädikate kommen jedoch auch ohne zusätzlichen Handler aus. `member/2` z. B. ist zunächst als Prolog-Programm implementiert und die, durch den L3 to WAM-Compiler erzeugten, Instruktionen in das Modul `builtin.[ch]` übernommen worden.

[21] Das Paket `bignum-1.2.tar.gz` ist unter : `http://www.mirrors.wiretapped.net/security/c\`

5.3 Ein neues Built-In-Prädikat

Um ein neues Built-In-Prädikat einzubauen, ist es notwendig, die VM unter vier Aspekten zu erweitern:

- Der Parser muss das Prädikat als Built-In identifizieren.
- Im Code-Segment muss eine entsprechende Repräsentation etabliert und verlinkt werden.
- Der Main-Loop muss auf den neuen Opcode vorbereitet und der entsprechende Handler angebunden werden.
- Der Handler, der Kern des Built-In's, muss implementiert werden.

5.3.1 Der Parser

Den Parser über das neue Prädikat zu informieren, ist relativ einfach, da sich die Syntax von Built-In-Prädikaten nicht von benutzerdefinierten Prädikaten unterscheidet.[22] D. h., dass weder die lexikalische- noch die syntaktische Analyse erweitert werden muss. Die WAM-Instruktion `call()` verwendet allerdings statt Funktor und Arity einen Schlüssel, um die einzelnen Prädikate aufzurufen. Dieser Schlüssel ist für Built-In-Prädikate fest definiert. Bei der Übersetzung von Prolog in WAM-Instruktionen wird der Schlüssel von der, im Modul `builtin.[ch]` definierten, Funktion `is_built_in()` ermittelt. Es ist also notwendig, diese Funktion um die Zuordnung zwischen Built-In-Prädikat und Schlüssel zu erweitern (Beispiel 22).[23]

5.3.2 Das Code-Segment

Wenn die VM gestartet wird, werden als erstes die Built-In-Prädikate im Code-Segment durch die Funktion `built_in()`, abgelegt. In den meisten Fällen ist das der

---

`ryptography/libraries/math/bignum/bignum-1.2.tar.gz` verfügbar.

[22] Verschiedene Built-In-Prädikate verwenden eine besondere Notation. Z. B. werden einige nullstellige Prädikate ohne Klammern zur Verfügung gestellt. Soll also eine besondere Notation oder Prädikatsname, wie z. B. '>', verwendet werden, muss eine entsprechende `TOKEN_BUILTIN_FUNCTOR`-Definition in `lex.l` eingetragen werden.

[23] Es gibt zwei Stellen, an denen der Schlüssel eines Prädikats bestimmt wird. Zum einen beim Parsen der Eingabe und zum anderen wärend der Laufzeit der VM, beim Aufruf eines Prädikats durch das Built-In `call/1` (siehe S. 60). In beiden Fällen werden zunächst die vom Benutzer definierten Prädikate untersucht und danach die Built-In-Prädikate. Auf diese Weise ist es möglich, Built-In-Prädikate durch benutzerdefinerte Prädikate zu überschreiben.

```
int
is_built_in(char *ext_name, unsigned arity)
{
 ...
 if ((strcmp("gt",ext_name)==0) && (arity==2))
    return (9);
 ...
}
```

Beispiel 22: Den Parser auf `gt/2(+Num1,+Num2)` vorbereiten

Opcode des Built-In's, gefolgt von `OC_PROCEED()`. Damit wird erreicht, das, wenn der Programmzeiger `P` während der Laufzeit der VM auf die Adresse des Built-In-Prädikats verschoben wird, ein entsprechender Opcode gelesen, der dazugehörende Handler ausgeführt und danach der Programmzeiger zurückgesetzt wird. Die Funktion `built_in()` muss also so verändert werden, dass sie für das neue Built-In-Prädikat die notwendigen Opcodes, samt Operanden, in das Code-Segment schreibt.[24] Diese Aufgabe kann in vielen Fällen von dem Makro `register_built_in()` erledigt werden (Beispiel 23). Neben der Veränderung des Code-Segmentes muss der Offset des Built-In-Prädikats an der durch den Schlüssel definierten Stelle in das Array `PREDLABEL` gespeichert werden (siehe S. 38).

5.3.3 Der Loop

Das Modul `loop.[ch]` ist das Herz der VM. Die hier definierte Funktion `loop()` ist dafür zuständig, den Programmzeiger durch das Code-Segment zu bewegen, die Opcodes zu lesen und die zuständigen Handler aufzurufen. In einer grossen `switch`-Anweisung sind die bekannten Opcodes von WAM-Instruktionen und Built-In-Prädikaten mit dem jeweiligen Handler verknüpft. `loop()` iteriert so lange, bis alle Anfragen beantwortet oder via `FAIL` beendet worden sind. Für ein neues Built-In muss einfach eine weitere `case`-Anweisung hinzugefügt werden (Beispiel 24).

[24] Wie schon in der Einleitung erwähnt, existieren Built-In-Prädikate, die keinen eigenen Handler benötigen. Sie sind ausschlieslich durch bereits bestehende Prädikate definiert. D. h. ihre Implementation ist eine Art Makro innerhalb des Code-Segmentes. Es ist für diese Prädikate nicht notwendig, den Main Loop zu verändern oder das Modul `heap.[ch]` zu erweitern.

```
#define OC_MATH_GT  54
void
built_in(void)
{
  ...
  register_built_in(9,"gt",2,OC_MATH_GT);
  ...
}
```

Beispiel 23: Modifikation von `built_in()` für `gt/2(+Num1,+Num2)`

```
void
loop (void)
{
 ...
 case OC_MATH_GT: do_math_test(opcode);break;
 ...
}
```

Beispiel 24: Der Main-Loop und `gt/2(+Num1,+Num2)`

5.3.4 Der Handler

Der Handler eines Built-In-Prädikats wird in dem Modul `heap.c` definiert und in `heap.h` deklariert. In den meisten Fällen ist es notwendig, anhand von 'Eingabe-Argumenten' einen oder mehrere 'Ausgabe-Argumente' zu unifizieren. Die Heap-Adressen der Argumente sind in den Registern A1 bis An gespeichert. Diese werden zunächst durch `deref()` dereferenziert und dann in Typ und Wert aufgelöst. Mit den Werten der Eingabe-Argumente können dann Ausgaben berechnet werden. Je nach Mode der Ausgabe-Argumente (gebundene oder ungebundene Variable), müssen die berechneten Werte entweder mit den Ausgabe-Argumenten verglichen oder an die Variablen gebunden werden. Im Beispiel 25 werden die Argumente A1 und A2 zunächst dereferenziert und der Typ ermittelt. Da `gt/2(+Num1,+Num2)` fordert, dass beide Argumente gebunden sind, sollte der ermittelte Typ `STR` sein. Ist dies der Fall, so werden im nächsten Schritt die Funktoren bestimmt.[25] Die Funktoren werden dann an die im Modul `ninterface.[ch]` definierte Funktion `ntest()` weitergereicht. Sollte diese Funktion mit `FALSE` zurückkehren, dann wird, entsprechend der Semantik von `gt/2(+Num1,+Num2)`, Backtracking ausgelöst. Kehrt `ntest()` dagegen mit `TRUE` zurück, geht die Kontrolle wieder an `loop()`.

Nachdem alle notwendigen Veränderungen vorgenommen sind, liefert die Ausführung des Prolog-Programms aus Beispiel 26 die erwartete Ausgabe '`2 > 1`'.[26]

[25] Auf die Überprüfung, ob es sich dabei um Konstanten `(Arity==0)` handelt, wird in diesem Beispiel verzichtet.

[26] Das über das Interface `ninterface.[ch]` angebundende Bignumberpaket liefert falsche Ergebnisse, falls die Argumente unterschiedliche Vorzeichen besitzen oder ein Argument '`0`' ist.

```
void
do_math_test(unsigned opcode)
{
  t_cell cell; t_heap_cell_type t; t_heap_cell_value v;
  t_predicate *predicate; char *a=NULL; char *b=NULL;
  t_cell_name arg1=a_register[1];
  t_cell_name arg2=a_register[2];
  t_cell_name cn1=deref(HEAP,arg1);
  t_cell_name cn2=deref(HEAP,arg2);
  starray_get(HEAP, cn1, &cell);
  eval_cell(cell, &t, &v);
  if (t==STR){
   starray_get(HEAP, v, &cell);
   a=starray_get_string(cell);
  else {
   fprintf (stderr,"WARNING wrong first argument)\n");
   FAIL=TRUE;}
  starray_get(HEAP, cn2, &cell);
  eval_cell(cell, &t, &v);
  if (t==STR){
   starray_get(HEAP, v, &cell);
   b=starray_get_string(cell);
  else {
   fprintf (stderr,"WARNING wrong second argument)\n");
   FAIL=TRUE;}
  if (ntest(a,b,N_GT)!=TRUE) FAIL=TRUE;
  if (FAIL) backtrack();
}
```

Beispiel 25: Der Handler für `gt/2(+Num1,+Num2)`

```
ausgabe():-gt(1,2),write('1 > 2'),nl(),fail().
ausgabe():-gt(2,1),write('2 > 1'),nl(),fail().
ausgabe():-gt(1,1),write('1 > 1'),nl().

?-ausgabe().
```

Beispiel 26: Ein Test für `gt/2(+Num1,+Num2)`

6 4-Gewinnt

In den letzten Abschnitten ist der Interpreter Schritt für Schritt erweitert worden. Es ist jetzt möglich, auch komplexe Programme und Probleme durch den Interpreter lösen zu lassen. Ein klassisches Gebiet der Künstlichen Intelligenz ist die Untersuchung von strategischen Spielen. Je nach Spiel sind die dafür verwendeten Alorithmen sehr rechen- und resourcen-intensiv. Im Folgenden soll das Spiel 4-Gewinnt implementiert und damit die Tauglichkeit des konstruierten Prolog-Systems unter Beweis gestellt werden. Zugleich sollen die gewonnenen Erfahrungen als Ausgangspunkt und Referenz für die Umsetzung in die Multirechnerumgebung dienen.

6.1 Spielbeschreibung

4-Gewinnt ist ein Strategiespiel für zwei Spieler. Das Spiel beginnt mit einem leeren Spielfeld von 7 Feldern Breite und 6 Feldern Höhe. Die beiden Spieler setzen abwechselnd einen Stein ihrer Farbe. Dabei legt man sich auf eine Spalte fest und der Stein kommt in dem untersten noch nicht belegten Feld dieser Spalte zum Liegen. Derjenige, der als erster eine vertikale, horizontale oder diagonale Reihe, bestehend aus 4 Steinen der eigenen Farbe, aufgebaut hat, hat gewonnen.

6.2 MiniMax-Algorithmus

Der MiniMax Algorithmus ist die Standard Methode, den Wert einer Spiel-Stellung und den zum Erreichen dieses Wertes notwendigen nächsten Zug zu bestimmen. Die Entscheidung basiert auf der Untersuchung des Spielbaums. Der Algorithmus verwendet eine beschränkte Tiefensuche und geht von jeweils optimalem Spiel der Kontrahenten aus. Die Endknoten des Spielbaumes werden von einer Bewertungsfunktion eingeschätzt. Auf jeder Ebene wird, abhängig vom 'Spieler am Zug', der jeweils beste (Max) bzw. schlechteste (Min) Knotenwert eines Kinderknoten an seinen Elternknoten zurückgereicht. Nach Untersuchung aller, in der Ausgangsstellung möglichen, Züge, steht ein maximaler Wert und der dafür notwendige Zug fest.

6.3 4-Gewinnt in Prolog

Für die Implementation des Spiels 4-Gewinnt ist es zunächst notwendig, eine Repräsentation für das Spielfeld, Spielsteine und Stellungen zu finden sowie eine Reihe weiterer Vereinbahrungen zu treffen. In Tabelle 10 sind die entsprechenden Variablen und deren Bildung bzw. Beispielterme aufgelistet.

Variable		**Bildung**
`Feld`	ϵ	{11-16,21-26,...,71-76}
`Spalte`	ϵ	{1-7}
`Spaltenhoehe`	ϵ	{0-6}
`Farbe`	ϵ	{o,x}
`Wert`	ϵ	{-3,-2,-1,0,1,2,3}
`Level`	ϵ	{0-6}
`Stein`	:	`s(Farbe,Feld)`
`Steinliste`	:	`l(Stein,l(Stein,l(...,[])))`
`Zug`	:	`z(Spalte,Stein)`
`Zuege`	:	`l(Zug,l(Zug,l(...,[])))`
`Record`	:	`record(Zug,Wert)`

`Stellung`	:	`stellung(Spaltenhoehe1,Spaltenhoehe2,`
		`Spaltenhoehe3,Spaltenhoehe4,`
		`Spaltenhoehe5,Spaltenhoehe6,`
		`Spaltenhoehe7, Steinliste)`

Tabelle 10: Vereinbahrungen für 4-Gewinnt

Unter Verwendung dieser Vereinbarungen sind alle Prädikate und Hilfsfunktionen des Spiels 4-Gewinnt implementiert worden. Die Funktionen von einigen wesentlichen Prädikaten sind in der anschliessenden Liste aufgeführt. Zentrale Bedeutung haben dabei `evaluateAndChoose/6`, `minimax/4` und `update/4`. Sie repräsentieren den MiniMax-Algorithmus und sind zum Teil [12] entnommen. Mit Hilfe des Prädikates `minimax/4` kann der günstigste Zug für eine Stellung ermittelt werden.

`aufBrett/3(+Stein1,+Stein2,+Stein3,+Stellung)`
: Wahr, falls die Steine `<Stein1>`, `<Stein2>` und `<Stein3>` bereits Teil von `<Stellung>` sind.

`gewinnt/2(+Stein,+Stellung)`
: Wahr, falls das Hinzufügen von `<Stein>` zu `<Stellung>` ein Gewinn für die Farbe von `<Stein>` bedeutet.

`gegnerFarbe/2(?Farbe1,?Farbe2)`
: Wahr, falls `<Farbe2>` die entgegengesetzte Farbe von `<Farbe1>` ist.

`figurZug/3(+Farbe,+Stellung,?Zug)`
: Erfüllt, falls es in der aktuellen `<Stellung>` für `<Farbe>` möglich ist, den `<Zug>` zu setzen.

`ziehe/2(+StellungAlt,+Zug,?StellungNeu)`
: Erfüllt, falls `<StellungNeu>` die um `<Zug>` erweiterte `<StellungAlt>` ist. D. h. die Steinliste um Stein erweitert und der Wert der entsprechenden Spalte um eins erhöht wurde.

`evaluateAndChoose/6(+Zuege,+Stellung,+Level,+Farbe,+Rec1,-Rec2)`
: Wahr, falls `<Rec2>` das beste Zug-Wert-Paar aus der Liste der `<Zuege>` in der gegebenen `<Stellung>` für `<Farbe>` ist.

`checkDirekt/4(+Zuege,+Stellung,-Zug,-Wert)`
: Wahr, falls `<Zug>` in der Liste der `<Zuege>` ist und in `<Stellung>` einen Gewinn für einen der Spieler wäre. Ein direkter Gewinn bedeutet einen `<Wert>` von 2.

`minimax/4(+Level,+Stellung,+Farbe,-RecordNeu)`
: Wahr, falls `<RecordNeu>` das beste Zug-Wert-Paar bei der `<Level>`-tiefen Untersuchung des Spielbaumes aus `<Stellung>` für `<Farbe>` ist.

`update/4(+ZugNeu,+WertNeu,+RecordAlt,-RecordNeu)`
: Wahr, falls das Zug-Wert-Paar `<RecordNeu>`, dem Wert nach, besser oder gleich dem Zug-Wert-Paar `<RecordAlt>` ist.

6.4 Ein Zug wird ermittelt

Um die Umsetzung des Spiels zu testen, wird die MiniMax-Analyse von dem Prädikat `lsg/0` und einer entsprechenden Anfrage gestartet (Beispiel 27).

```
teststellung(stellung(3,0,0,0,0,0,0,
                      l(s(x,13),l(s(x,12),l(s(x,11),[])))))).
lsg():-    teststellung3(X),display(X),
           minimax(4,X,x,record(Zug,Wert)),
           write('Der beste Zug : '),write(Zug),
           write(' liefert den Wert : '),write(Wert),nl().
?-lsg().
```

Beispiel 27: Das 4-Gewinnt-Testprädikat `lsg/0`

Je nach übergebener Teststellung blockt der Algorithmus einen Gewinn des Gegners, sieht den eigenen Gewinn oder ist bestrebt, das Zentrum zu beherrschen. Das Ergebnis ist ein entsprechender Zugvorschlag und Wert (Beispiel 28).

```
[bozo@foo] ./cpl -o -f ../example/4gewinnt.pl
...

|#X#|   |   |   |   |   |   |
-----------------------------
| X |   |   |   |   |   |   |
-----------------------------
| X |   |   |   |   |   |   |
-----------------------------

Der beste Zug : z(1, s(x, 14)) liefert den Wert : 2

Yes
[bozo@foo]
```

Beispiel 28: Testlauf für Beispiel 27

Die für die Lösung benötigte Zeit und Speicherbedarf hängt von der Stellung und gewünschter Halbzugtiefe ab. Die Zahl der möglichen Züge ist fast konstant 7. Damit wächst, abgesehen von Gewinn- oder Verluststellungen, die Zahl der zu untersuchenden Stellungen mit jedem weiteren Halbzug um den Faktor 7. Bei einer Halbzugtiefe von 4 sind das etwa 2.400 Endstellungen. Bei 6 Halbzügen sind es

bereits rund 118.000 Stellungen, die entwickelt werden. Neben der Berechnungszeit steigt auch der Speicherbedarf drastisch an. Besonders die Speichersegmente Heap und Stack sind davon betroffen. Ausgehend von der Grundstellung, dem leeren Spielfeld, und einer Berechnungstiefe von 6 Halbzügen, sind die in Tabelle 11 aufgelisteten Einstellungen [27] ausreichend. Eine Stack-Grösse von 10.000 Speicherseiten bzw. 40MByte war in diesem Test nicht genügend.

Segment	**Seitenindex**	**Speicher**
`SUBS,PREDLABEL`	10	40 KByte
`CODE,PDL`	100	400 KByte
`HEAP`	10.000	$\approx$ 39 MByte
`STACK`	20.000	$\approx$ 78 MByte

Tabelle 11: Speichereinstellungen für 6 Halbzüge

Sind die einzelnen Speichergrössen entsprechend angepasst, ergibt sich eine Berechnugszeit von ca. 12 Minuten und 15 Sekunden (Beispiel 29).

```
[bozo@foo] time ./cpl -o -f ../example/4gewinnt.pl
...
Der beste Zug : z(4, s(x, 41)) liefert den Wert : 0
...
real    12m14.631s
user    12m14.620s
sys     0m0.000s
[bozo@foo]
```

Beispiel 29: Aufwand für 6 Halbzüge aus der Grundstellung

Der erhebliche Speicher- und Zeitbedarf wäre durch Optimierungen wie z. B. Last-Call-Optimisation, schnellere mathematische Operationen oder verfeinerte Heap-Repräsentationen deutlich zu reduzieren. Für das angestrebte Ziel, Prolog-Programme partiell parallel auszuführen, ist die bestehende Umsetzung ausreichend und soll in dieser Richtung nicht weiter verbessert werden.

[27] Die Speichergrössen sind in der Datei `common.h` definiert.

7 Parallelität

Der Stand, der mit dem letzten Abschnitt erreicht wurde, ist ein lauffähiger Prolog-Interpreter, eine Reihe von Built-In-Prädikaten und eine Beispiel-Implementation des Strategiespiels 4-Gewinnt. Im Folgenden soll der Interpreter so erweitert werden, dass ein effizienter Einsatz auf Cluster-Computern möglich wird. Dazu sollen zuerst bestehende Lösungen und die darin verwendeten Ansätze untersucht werden. Auf dieser Grundlage werden dann die Idee der vorliegenden Arbeit, resultierende Probleme und Lösungen vorgestellt. Unter Verwendung eines MPI-Paketes, soll das Built-In-Prädikat `findallparallel4/4` erarbeitet werden.

7.1 Ansätze in anderen Arbeiten

Die Idee, die Lösungssuche in Prolog-Programmen zu parallelisieren, ist nicht neu und es existieren zahlreiche Arbeiten zu diesem Thema. Eine Reihe verschiedener Ansätze werden dabei verfolgt, die sich nach den Kriterien Art der Parallelität, Art der Nutzung, unterstützte Hardware sowie Art der Kommunikation klassifizieren lassen.

Art der Parallelität

Es besteht die Möglichkeit, die einzelnen Teilziele einer Klausel parallel zu untersuchen. Herrschen Abhängigkeiten zwischen den Teilzielen, so wird diese Art der Parallelität als UND-Parallelität (AND-Parallelism) und anderenfalls als Unabhängige UND-Paralellelität (Independent AND-Parallelism) bezeichnet. Eine andere Art ist die ODER-Parallelität (OR-Parallelism), bei der die unterschiedlichen Klauseln eines Prädikats parallel untersucht werden. In einer dritten Art werden bei der Wahl und Bearbeitung einer Klausel besondere Randbedingungen berücksichtigt oder Alternativen unter bestimmten Voraussetzungen gänzlich vernachlässigt (Guards bzw. Committed Choice).

Art der Nutzung

Es gibt zwei Arten wie der Benutzer in den Genuss der Parallelität kommen kann. Entweder kann der Programmierer die parallele Ausführung durch einen erweiterten Sprachumfang explizit steuern oder die Möglichkeiten werden vom

Compiler bzw. Interpreter implizit ausgenutzt. Es existieren Systeme die Parallelität sowohl implizit nutzen als auch explizite Konstrukte anbieten.

Unterstützte Hardware

Bei der parallelen Lösungssuche arbeiten mehrere Prozesse an unterschiedlichen Teilproblemen. Es ist möglich, alle Prozesse von einem einzigen Prozessor berechnen zu lassen. Sinnvoller ist es jedoch, Systeme mit mehreren Prozessoren zu verwenden. Systeme, die über einen gemeinsamen Speicher verfügen, werden als Multiprozessorsystem bezeichnet. Systeme, bei denen jeder Prozessor über einen eigenen Speicher verfügt, auf den nur er Zugriff hat, werden als Multirechnersysteme bezeichnet. In der späteren Variante, die in einer eng vernetzten Version als Cluster-Computer (Cluster) bezeichnet wird, erfolgt der komplette Datenaustausch durch Versenden von Nachrichten über ein Verbindungsnetzwerk.

Art der Kommunikation

Unumgänglich ist die Kommunikation und Synchronisation der einzelnen Prozesse. Es existiert die Möglichkeit, Daten und Informationen als Nachrichten über ein gemeinsames Netzwerk und ein sogenanntes Message Passing Interface (MPI) auszutauschen. Eine zweite Variante verwendet gemeinsame Speicherbereiche (Blackboards). Ein Blackboard kann z. B. eine Datenbank sein.

In Tabelle 12 wird versucht, bestehende Systeme, soweit eine eindeutige Entscheidung möglich ist, zu klassifizieren [7, 13, 14, 8, 4, 9, 10, 16, 3].

Die meisten Umsetzungen arbeiten in Multiprozessorumgebungen und nutzen eine der Arten von Parallelität implizit. Die verwendeten Algorithmen bauen auf den gemeinsamen Speicher und lassen sich nur schwer in Multirechnerumgebungen integrieren.

7.2 Ansatz dieser Arbeit

In dieser Arbeit soll ein Cluster als Hardware-Grundlage dienen. Dieser Typ von Parallelrechner gewinnt in jüngster Zeit, dank seiner einfachen und vergleichsweise billigen Skalierbarkeit, immer mehr an Bedeutung. Da in einem solchen Multirechnersystem die Kommunikation zeitaufwendig ist, soll die Verantwortung an den Programmierer übertragen, also ein expliziter Ansatz gewählt werden. Diese Art der Nutzung erschwert zwar die Programmierung, kann aber zu höchsten Performancegewinnen führen. Es soll ODER-Parallelität angeboten und ein Message Passing

System	Parallel	Nutzung	Hardware	Kommunikation
YAP	ODER	implizit	Multiprozessorsysteme	gem. Speicher
Ciao	UND	implizit, explizit	Multiprozessorsysteme	gem. Speicher
Muse	ODER	implizit	Multiprozessorsysteme	gem. Speicher
Aurora	ODER, UND	implizit	Multiprozessorsysteme	gem. Speicher
Parlog	UND ODER	explizit	Multiprozessorsysteme	gem. Speicher Guards
&-Prolog	UND	implizit, explizit	Multiprozessorsysteme	gem. Speicher
Delta Prolog	UND	explizit	Multirechnersysteme	BSD-Sockets
PLoSys	ODER	implizit	Multirechnersysteme	MPI
Pals	ODER	implizit	Multirechnersysteme	MPI

Tabelle 12: Klassifikation verschiedener bestehender Systeme

Interface für die Kommunikation zwischen den einzelnen Prozessen verwendet werden.

Um ODER-Parallelität (OP) zur Verfügung zu stellen, bietet es sich an, das Prädikat `findall/3(+Var,+Pred,-List)` zu verwenden. Alle alternativen Klauseln des Prädikats `+Pred` können auf die Knoten des Clusters verteilt und zugleich untersucht werden. `List` wird dann mit der Liste sämtlicher Lösungen für `Var` unifiziert und `findall/3` kehrt zurück. Die Verteilung der Choice-Points, das Kopieren der Speicherbereiche, der indirekte Aufruf von Prädikaten sowie das Sammeln von Ergebnissen erfordert jedoch noch einige Modifikationen und Erweiterungen des Prolog-Interpreters.

7.2.1 Konstanten

Folgt man den Ausführungen in [5], so arbeitet die WAM mit den Speicherbereichen `HEAP`, `CODE`, `PDL`, `STACK` und `TRAIL`. Für die VM sind die Segmente `SUBS` und `PREDLABEL` hinzugefügt worden (Abschnitt 4.3). Mit dem Ziel, die Ausführung der WAM-Instruktionen zu parallelisieren, muss die Behandlung von Konstanten verfei-

nert werden.

Bisher war es möglich, Zeichenketten wie Funktoren oder Zahlen einfach durch eine Referenz auf den Betriebssystem-Speicher zu behandeln. WAM-Instruktionen wie `GET_STRUCTURE()` oder `PUT_STRUCTURE()` haben den notwendigen Speicher via `allocate()` angefordert, die Zeichenketten kopiert und die Zeiger als Referenz im jeweiligen WAM-eigenen Speicherbereich, z. B. Heap-Segment, in einer einzigen Zelle abgelegt. Die einzelnen Knoten eines Clusters verfügen über keinen gemeinsamen Speicher. Das bedeutet, dass die Kopien von Stack und Heap ungültige Zeiger enthalten könnten und der Unifikationsalgorithmus schon im Ansatz scheitern würde. Auf Rechnern, die keinen gemeinsamen Speicher besitzen, ist es also notwendig, Zeichenketten in einem VM-eigenen, kopierbaren, Speicherbereich zu sichern. Ein Problem, welches sich dabei ergibt, ist das Alignment der Zeichenketten. Während das Modul `mem.[ch]` darauf ausgelegt ist, mit definierten Zellen vom Typ `unsigned` zu arbeiten und jede Speicherseite eine feste Anzahl dieser Zellen besitzt, müssen Zeichenketten auf `Byte`-Basis gespeichert werden. Ausserdem ist die Länge der einzelnen Zeichenketten variabel und das Ende der einzelnen Speicherseiten darf nicht überschrieben werden.

Der neu eingeführte Speicherbereich `STRINGS` wird durch die beiden Funktionen `starray_push_string()` und `starray_get_string()` beschrieben bzw. abgefragt. Die Funktionen stammen aus dem Modul `starray.[ch]` und dienen dazu, die Konsistenz von Heap und Stack-Segmenten, auch in den Kopien auf den anderen Knoten des Clusters, zu erhalten.[28]

7.2.2 Das Built-In-Prädikat `call/1`

Die Implementation von `findall/3` ist deutlich komplizierter als die normaler Built-In-Prädikate. Zum einen muss aus einem Argument ein Aufruf eines Prädikats extrahiert werden und zum anderen müssen via Backtracking alle Lösungen identifiziert und gesichert werden.

Es bietet sich an, zunächst das Prädikat `call/1` zu implementieren. `call/1` besitzt als einzigstes Argument das eigentliche Teilziel samt seinen Argumenten. Eine Rei-

[28] Ein hier unbehandeltes Problem ist die Garbage-collection über diesen Zeichenketten. Eine Lösung für dieses Problem ist jedoch relativ leicht zu implementieren. Z. B. könnte man die WAM-Speicherbereiche nach Referenzen untersuchen, Waisen enfernen und am Anfang frei gewordene Speicherbereiche durch Zeichenketten in hinteren Bereichen auffüllen. Ausserdem könnten die einzelnen Zeichenketten ähnlich wie der Heap durch den Trail überwacht werden.

he von Schritten ist bei der Umsetzung des Built-In's notwendig. Zunächst muss ein neuer Environment-Frame erzeugt werden. Als nächstes wird das eigentlich aufzurufende Prädikat identifiziert und der entsprechende Offset im Code-Segment ermittelt.[29] Anders als bei normalen Prädikaten müssen dazu sämtliche Prädikate via `PREDLABEL` untersucht werden. Die Verwendung von Schlüsseln ist in diesem Fall nicht möglich. Die A- und X-Register, die normalerweise durch `PUT`-Instruktionen gesetzt werden, müssen vom `call`-Handler manuell bestimmt sowie Programmzeiger und Continuation-Pointer angepasst werden. Nach Rückkehr des aufgerufenen Prädikats ist es notwendig, die alte Umgebung wieder herzustellen bevor mit der normalen Programmausführung fortgefahren werden kann.[30]

Die Implementation `call/1` ist neben der Verwendung in `findall/3` sehr nützlich. Verschiedene Operatoren sind darauf angewiesen, als Argumente Prädikate zu akzeptieren und in Abhängigkeit von deren Ausführung ein Ergebnis zu bestimmen. So lässt sich jetzt z.B. das Prädikat `not/1` bzw. `\+/1` implementieren.

Anders als die im Beispiel 21 vorgestellt Version, kann `call/1` verwendet werden, das entsprechende Prädikat 'durchzureichen'.

Auf diese Weise wird die Implementation unabhängig vom zu negierenden Prädikat (Beispiel 30). Die aus dieser Version erzeugten WAM-Instruktionen lassen sich nun ohne Aufwand als Built-In-Prädikat sichern.

```
not(X):-call(X),
        !,
        fail().
not(X):-true().
```

Beispiel 30: Prädikat-unabhängige Negation

7.2.3 `assert/2` und `retract/2`

Ein weiterer notwendiger Teilschritt, `findall/3` zu implementieren, ist, einen Weg zu finden, Zwischenergebnisse zu sichern. Dies ist notwendig, da Backtracking die

[29] Bislang gibt der Parser einen Fehler aus, falls das aufgerufene Prädikat nullstellig ist. Da ein Subterm mit 'leeren' Klammern bisher nicht verarbeitet wird.

[30] Die genaue Umsetzung des Handlers und Built-In-Definition ist in den Modulen `heap.[ch]` bzw. `builtin.[ch]` nachzulesen.

Segmente Trail und Heap 'putzt' und somit alte Lösungen verloren gehen würden. Der übliche Weg ist, die einzelnen Lösungen via `assert/2` zu speichern und in einer 'späteren' Klausel mit `retract/2` wiederherzustellen.

Für die Implementation dieser Prädikate werden zwei neue Speicherbereiche eingeführt. Im Segment `ASSERT` (Assert) werden Terme gespeichert und durch Einträge im Segment `ASSERTLABEL` (Assertlabel) referenziert.

`assert/2(+Identifier,+Term)` kopiert den `Term` vom Heap in das Segment Assert. Ausserdem speichert `assert/2` den Offset samt `Identifier` sowie die Informationen über die untere und obere Grenze des belegten Speichers in Assertlabel. `retract/2(+Identifier,-Term)` verhält sich genau umgekehrt. Zunächst werden alle Einträge in Assertlabel nach dem `Identifier` durchsucht und, sofern ein solcher existiert, der entsprechende `Term` von Assert in den Heap kopiert. Nachdem der `Term` kopiert ist, werden die Informationen aus Assertlabel und Assert entfernt. Kann der `Identifier` nicht gefunden werden, scheitert `retract/2`. Auf der Basis von `retract/2` ist zusätzlich das Prädikat `retractOrNIL/2` implementiert, welches im Fehlerfall die Konstante `[]` zurückliefert.

Für die Prädikate `assert/2` und `retract/2` existiert jeweils ein eigener Händler. Der Kopiervorgang wird in beiden Fällen durch `copy_term()` erledigt. Die Schwierigkeit beim Kopieren besteht darin, den Term nachzubilden statt einen String zu kopieren. Die einzelnen Referenzen müssen aufgelöst und die Struktur neu aufgebaut werden. Würde man die Struktur nicht nachbilden, wäre es später unmöglich, die Zwischenergebnisse in ihre Bestandteile zu zerlegen.

7.2.4 `findall/3` und `findall4/4`

Nachdem es möglich ist, ein Prädikat indirekt aufzurufen und Zwischenergebnisse zu speichern, lässt sich nun das Prädikat `findall/3` definieren (Beispiel 31). Anders als sonst üblich, wird das Prädikat als vierstelliges Prädikat entwickelt. Das zusätzliche Argument `Id` ist der von `assert/2` benutzte Identifier.[31]

Beispiel 32 erweitert Beispiel 31 zu einem vollständigen Programm, dessen Ausführung in Beispiel 33 gezeigt ist.

[31] Es bestünde die Möglichkeit, in einer Built-In-Version von `findall`, auf das zusätzliche Argument zu verzichten und stattdessen z. B. den Offset im Code-Segment zu verwenden. In dieser Arbeit soll jedoch darauf verzichtet und die Besonderheit durch den Namen `findall4/4` gekennzeichnet werden.

```
findall4(Var,Goal,Id,List) :- call(Goal),
                      retractOrNIL(Id,X),
                      assert(Id,l(Var,X)),
                      fail().
findall4(Var,Goal,Id,List) :- retractOrNIL(Id,List).
```

Beispiel 31: Benutzerdefinierte Implementation von `findall4/4`

```
doit(1,1).
doit(1,2).
doit(1,3).
doit(2,4).
doit(2,5).
doit(1,6).

?-findall4(V,doit(1,V),tmp,L).
```

Beispiel 32: Verwendung von `findall4/4`

```
[bozo@foo] ./cpl -o -f ../example/findall.pl

V = _
L = l(6, l(3, l(2, l(1, []))))

Yes

[bozo@foo]
```

Beispiel 33: Test zu 31 und 32

Interessant ist, dass die Beziehungen der Variablen `V` innerhalb des Aufrufs `findall4(V,doit(1,V),tmp,L)` auch in die Klauseln von `findall4/4` übernommen werden. Obwohl in `call(Goal)`, dem ersten Teilziel der ersten Klausel von `findall`, die Variable `Var` nicht vorkommt, bleibt die Referenz erhalten und `Var` wird korrekt gebunden.

7.3 MPI

MPI ist eine Spezifikation und es exitieren eine Reihe von Implementationen. Um den Pseudo-Interpreter auf einem Cluster parallel laufen zu lassen, wird die frei verfügbare MPICH-Implementation der University of Chicago und der Mississippi State University verwendet.[32] Das Paket umfasst verschiedene Bibliotheken, Compiler und Helfer, die das Erstellen von parallelen Programmen vereinfachen. Wesentlicher Teil der Biblotheken sind Routinen zum Senden und Empfangen von Nachrichten. Dabei werden Funktionen zum Senden von blockierenden und nichtblockierenden Nachrichten zwischen einzelnen Prozessen und Gruppen von Prozessen angeboten.

Die Idee von MPI ist, durch geeignete Helfer mehrere Instanzen eines Programms zu starten. Ähnlich wie beim 'forken' von Prozessen, besitzt jeder der parallelen Prozesse eine besondere Identifikationsnummer / Rang (ID), über die der Ablauf gesteuert werden kann. In der Regel bearbeitet einer der Prozesse, der Server, den sequentiellen Teil eines Programms. Die anderen Knoten, die Clients, sind in dieser Zeit 'arbeitslos' und warten auf ein Signal vom Server. Durch Nachrichten werden die Prozesse synchronisiert und Daten ausgetauscht. Alle Knoten berechnen dann unterschiedliche Teilaufgaben eines Problems und senden ihre Ergebnisse an den Server. Oft terminieren die Clients, während der Server noch mit abschliessenden, sequenziellen Aufgaben, beschäftigt ist.

7.4 `findallparallel4/4`

Der erste Schritt um verschiedene Klauseln eines Prädikates parallel auszuführen, ist, aus der benutzerdefinierten Lösung von `findall4/4`, ein Built-In-Prädikat zu erzeugen. Diese Aufgabe ist relativ einfach, da die initial erzeugten WAM-Instruktionen direkt als Built-In-Prädikat übernommen werden können. Ein eigener

[32] Die Quellen sind anonym unter `ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz` verfügbar.

Handler oder Opcode ist zunächst nicht nötig. Die Modifikationen beschränken sich auf die Funktionen `is_built_in()` und `built_in()` aus dem Modul `builtin.[ch]` (Beispiel 34).

Nachdem das Built-In-Prädikat `findall4/4` die gewünschten Eigenschaften hat, wird eine Kopie dies Prädikates als `findallparallel4/4` registriert. Zu diesem Zeitpunkt verhalten sich beide Prädikate gleich. Die Aufgaben von `findallparallel4/4` werden jedoch, durch die im Folgenden aufgelisteten, erweitert.

- den Beginn der parallelen Arbeit signalisieren und den Zustand der WAM kopieren
- den Choice-Point des parallel zu bearbeitenden Prädikates markieren
- Client-abhängige Bearbeitung ausgewählter Klauseln des markierten Choice-Points und Speichern der Zwischenergebnisse
- nach Aufforderung die Ergebnisse in den Heap des Servers kopieren
- die Clients beenden die Arbeit ihrer VM und warten auf das Programmende oder auf neue Aufgaben

7.4.1 `OC_SPLIT`

Bis zum Erreichen von `findallparallel4/4`, soll die Lösungssuche nur auf dem Server erfolgen. Am Anfang des Built-In-Prädikates wird daher der Opcode `OC_SPLIT` eingefügt, der die 'Schlafenden' weckt. Der Handler von `OC_SPLIT` hat die Aufgabe, die von den Clients erwartete Nachricht `MYMPI_MESSAGE_SPLIT` zu senden. Damit werden die Clients darauf vorbereitet, eine Menge von Daten entgegen zu nehmen. Die Funktion `bcast_vmState()` aus dem Modul `par.[ch]` sendet in einer bestimmten Reihenfolge Broadcast-Nachrichten. Diese Nachrichten enthalten alle Informationen, um auf den Clients ein Abbild der VM des Servers zu erstellen. Neben den Speicherbereichen werden auch die Register und globalen Variablen kopiert. Nachdem `OC_SPLIT` abgearbeitet ist, befinden sich die VM's aller Clients im gleichen Zustand und jede der Maschinen setzt ihre Arbeit mit `TRY_ME_ELSE(4)` fort (vgl. Beispiel 34).

```
...
REGISTER_PREDICATE(22,"findall4",4);
                      /* << starray_push(CODE,OC_SPLIT); */
TRY_ME_ELSE(4);
ALLOCATE(3);
GET_VARIABLE_YA(1,1);
GET_VARIABLE_XA(5,2);
GET_VARIABLE_YA(2,3);
GET_VARIABLE_XA(6,4);
PUT_VALUE_XA(5,1);
CALL(17);             /* call/1 bzw.
                      << CALL(28) callparallel/1 */
PUT_VALUE_YA(2,1);
PUT_VARIABLE_YA(3,2);
CALL(21);             /* retractOrNIL/2 */
PUT_VALUE_YA(2,1);
PUT_STRUCTURE_A("l",2,2);
SET_VALUE_Y(1);
SET_VALUE_Y(3);
CALL(18);             /* assert/2 */
CALL(1);              /* fail/0 */
DEALLOCATE();
REGISTER_LABEL();
TRUST_ME();
ALLOCATE(0);
GET_VARIABLE_XA(5,1);
GET_VARIABLE_XA(6,2);
GET_VARIABLE_XA(7,3);
GET_VARIABLE_XA(8,4);
PUT_VALUE_XA(7,1);
PUT_VALUE_XA(8,2);
CALL(21);             /* retractOrNIL/2 bzw.
                      << starray_push(CODE,OC_JOIN); */
DEALLOCATE();
number_of_built_ins++;
...
```

Beispiel 34: `findall4/4` bzw. `findallparallel4/4` in `built_in()`

7.4.2 OC_CALL_PARALLEL

Nachdem die VM auf allen Knoten gestartet wurde, werden jetzt die gleichen Aufgaben berechnet. Es ist daher nötig, das durch das Built-In-Prädikat `call/1` aufgerufene Prädikat und den damit erzeugten Choice-Point in besonderer Weise zu markieren. Das neue Prädikat `callparallel/1`, mit dem Opcode `OC_CALL_PARALLEL`, wird verwendet, um die globale Variable `PADDRESS` auf die Stack-Adresse des folgenden Choicepoints zu setzen. Jedesmal, wenn `TRY_ME_ELSE`, `RETRY_ME_ELSE` oder `TRUST_ME` den Choice-Point, an der in `PADDRESS` gespeicherten Adresse, erreichen, wird `P` und `BP` auch in Abhängigkeit von der ID des Prozesses, bestimmt. D. h., nur ausgewählte Klauseln des aufgerufenen Prädikates werden untersucht. Die Aufteilung der Klauseln ist abhängig von der Zahl der Klauseln und der Zahl der zur Verfügung stehenden Prozesse. Gibt es z. B. 10 Klauseln und 4 Prozesse, so werden die Klauseln 0, 4 und 8 vom Prozess mit der ID 0 bearbeitet. Prozess 1 untersucht 1, 5 und 9 und die Prozesse 2 und 3 berechnen die Klauseln 2 und 6 bzw. 3 und 7 (Abbildung 7).[33]

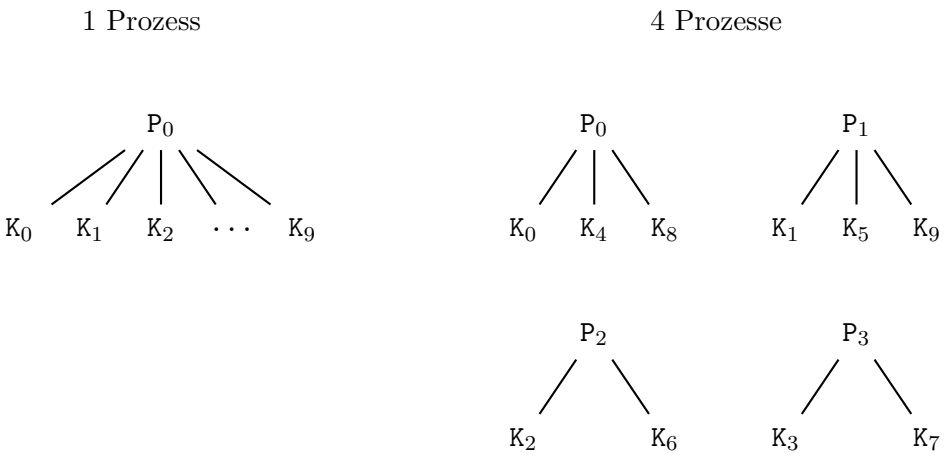

Abbildung 7: Aufteilung der Klauseln bei 1 vs. 4 Prozessen

Durch die Bearbeitung nur ausgewählter Klauseln wird es erforderlich, die Handler

[33]Es wäre besser, statt der statischen Aufteilung der Klauseln eine andere Art der Lastverteilung zu finden. Häufig werden dafür z. B. eigenständige Prozesse verwendet, die die Arbeit dynamisch verteilen. In dieser Arbeit wurde jedoch eine einfache Variante gewählt.

von TRY_ME_ELSE, RETRY_ME_ELSE und TRUST_ME zu erweitern. Z. B. könnte es passieren, dass die VM eines Knotens den Handler von TRUST_ME aufruft, der Knoten aber gar nicht für die Berechnung dieser letzten Klausel zuständig ist. Statt die Klausel zu berechnen, muss die VM des Knotens Backtracking auslösen. Ein ähnliches Problem ergibt sich, wenn die, einem Knoten zugewiesene, Klausel die letzte Klausel ist und statt von RETRY_ME_ELSE von TRUST_ME behandelt werden sollte. Um diese Fälle zu berücksichtigen und den einzelnen Knoten die korrekten Klauseln zuzuordnen, rufen die Handler von TRY_ME_ELSE, RETRY_ME_ELSE und TRUST_ME die Funktion try_retry_trust_me() auf. Diese Funktion prüft zunächst, ob es sich um den in PADDRESS vermerkten Choice-Point handelt. Nur wenn dies der Fall ist, werden die zu bearbeitenden Klauseln neu ermittelt und Grenzfälle behandelt. Die Variable L wird dabei in allen Knoten gleich gesetzt. In einer Situation wie in Abbildung 7, würde L in allen Knoten, während der Ausführung der jeweils ersten Klausel, die Adresse der vierten enthalten. In der nächsten Runde ermitteln alle Knoten, ausgehend von dieser Klausel, die nächste Klausel, für die sie zuständig sind. Dieser Vorgang wird solange wiederholt, bis alle Klauseln des Prädikats untersucht sind.

7.4.3 OC_JOIN

Jeder der Knoten untersucht zwar nur einen Teil der Klauseln, findallparallel4/4 verhält sich aber ansonsten wie findall4/4. D. h. die Zwischenergebnisse werden einzeln berechnet und durch die Built-In-Prädikate assert/2 und retract/2 zu einer Liste zusammengesetzt. Bevor das Prädikat findallparallel4/4 beendet werden kann, müssen die Ergebnisse der Clients an den Server zurückgereicht werden. Der Opcode OC_JOIN wird dazu in das Prädikat eingebaut (vgl. Beispiel 34). Statt retractOrNIL/2 übernimmt es der Handler von OC_JOIN, sich mit dem Server zu synchronisieren und die Ergebnisliste zu senden. Der Server erfragt nacheinander die Ergebnisse von jedem Client. Die Nachrichten und Daten werden dabei von den, im Modul par.[ch] definierten, Funktionen par_starray_push_string(), par_starray_push(), par_starray_size() und receive_results() von den Clients gesendet bzw. vom Server empfangen. Der Prozess des Sendens wird auf Client-Seite durch die Funktion copy_term_net() gesteuert. Diese Funktion kopiert, ähnlich wie copy_term(), einen Term zwischen zwei Speicherbereichen der WAM. Anders als bei copy_term(), können sich bei copy_term_net() die Speicher auf unterschiedlichen Knoten des Clusters befinden.

Für die Clients bleibt nach dem Senden der Ergebnisse nur die Aufgabe, die Variable end_of_parallel zu setzen. Unabhängig vom nächsten Opcode, wird dadurch die

Funktion `loop()` verlassen und die Speicher der VM gelöscht. Die Clients werden bis zu einem erneuten `MYMPI_MESSAGE_SPLIT` 'schlafen' geschickt oder durch die Nachricht `MYMPI_MESSAGE_END_OF_PROGRAM` zum Terminieren aufgefordert.

Auf Server-Seite wird mit dem normalen Ablauf der VM fortgefahren.

7.4.4 Ein Ablaufbeispiel

Abbildung 8 stellt den zeitlichen Ablauf eines parallel interpretierten Prolog-Programms dar. Nach der Initialisierung aller Prozesse übernimmt es der Server, die Kommandozeile auszuwerten, das Prolog-Programm in WAM-Instruktionen zu übersetzen und das Code-Segement zu beschreiben. Der Einsprungpunkt der ersten Anfrage wird ermittelt und die VM beginnt damit, die einzelnen Opcodes des Code-Segmentes zu interpretieren. Trifft die VM auf den Opcode `OC_SPLIT`, werden die Clients aufgeweckt und der gegenwärtige Zustand der VM übertragen. Jeder der Prozesse ist jetzt im gleichen Zustand und die Maschinen führen alle die gleichen Arbeiten aus. Erst der Opcode `OC_CALL_PARALLEL` veranlasst sie, nur ausgewählte Klauseln des folgenden Prädikats zu untersuchen. Jede der Maschinen berechnet ihre Ergebnisse. Der Opcode `OC_JOIN` synchronisiert die Prozesse und der Server nimmt die Ergebnisse der Clients entgegen. Diese löschen daraufhin ihre Speicherbereiche und kehren in den Schlafmodus zurück. Der Vorgang wiederholt sich solange, bis der Server alle Anfragen abgearbeitet hat. Daraufhin werden die anderen Prozesse mit einer Nachricht über das Ende informiert und das Programm mit `MPI_Finalize` beendet.

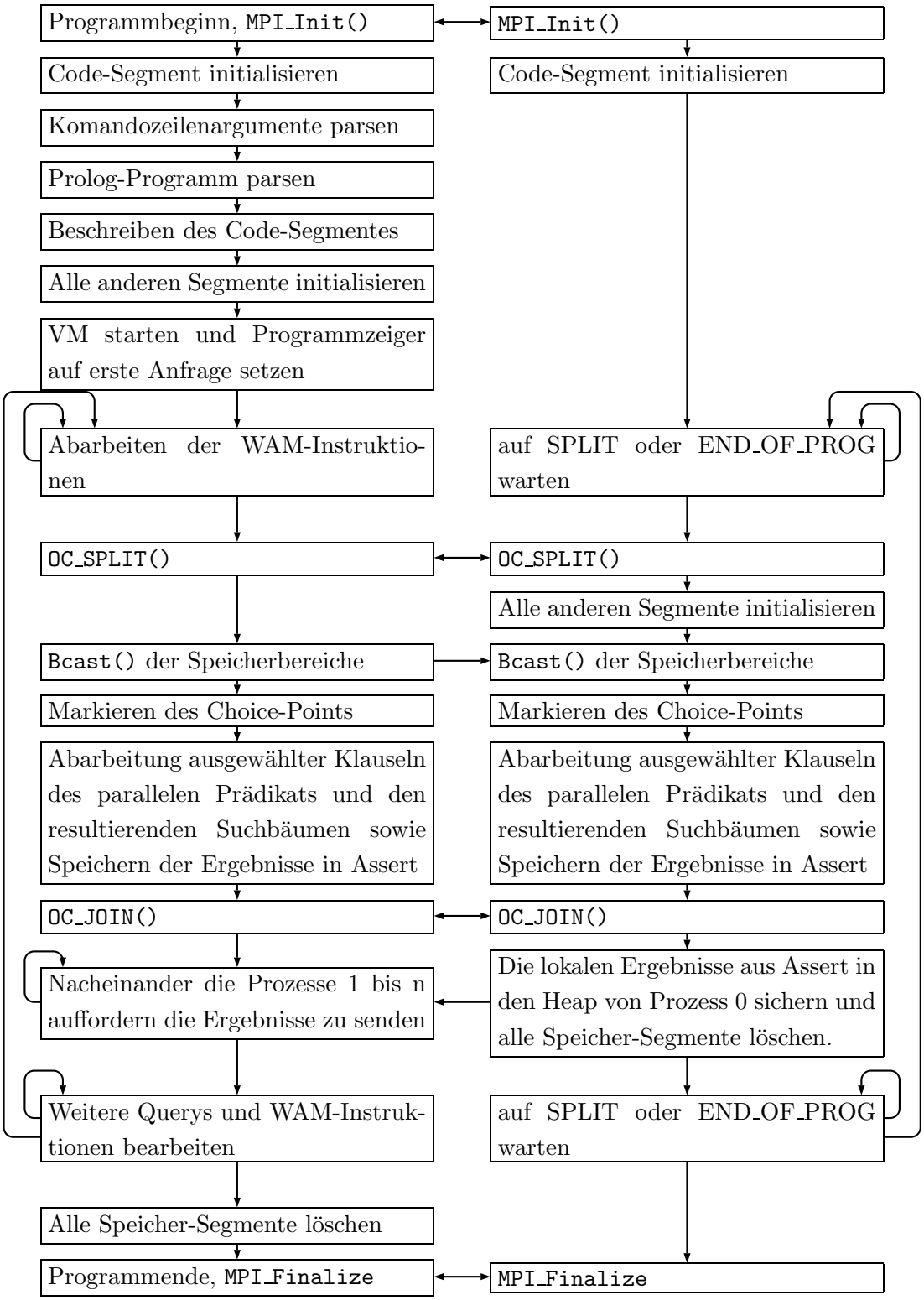

Abbildung 8: Ablauf eines mit `cplparallel` interpretierten Programmes

8 Tests und Ergebnisse

Im Folgenden sollen einige Performancetests durchgeführt und deren Ergebnisse diskutiert werden.

8.1 Testaufbau

Grundlage für die Tests sind die Programme `cpl` und `cplparallel`. `cpl` ist eine Version des Pseudo-Interpreters, die mit Hilfe von `gcc` compiliert wurde. Diese Version ist gänzlich unabhängig von einer MPI-Installation und unter Linux und Solaris getestet. `cplparallel` wurde von `mpicc` unter Verwendung der Option `-D_MPI` erzeugt. Mit der zusätzlichen Option werden die für die parallele Verarbeitung relevanten Programmteile und MPI-Bibliotheken vom Präprozessor eingebunden. Für die Zeitmessung wurde das Prolog-Programm `4gewinnt.pl` mit unterschiedlichen Einstellungen interpretiert. Dabei sind die Halbzugtiefe, die Ausgangsstellung sowie das verwendete Built-In-Prädikat variiert worden. Das Shell-Programm `time` hat die Programmlaufzeiten ermittelt. Der erste Test (Abbildung 9) wurde auf einer Einprozessormaschine Pentium IV, 1.7GHz, 512MByte RAM mit MPI-Installation durchgeführt. Alle anderen Tests liefen auf dem Cluster-Computer der TU-Berlin, Fachbereich Informatik, Fachgebiet Kommunikations- und Betriebssysteme. Dieser Cluster verfügt über 16 Doppelprozessor-Knoten mit je 1 GB Hauptspeicher, die über SCI zu einem 2D-Torus vernetzt sind.

Die in den Zeit-Diagrammen verwendeten Beschriftungen sind in Tabelle 13 ausgeführt. Die Balkenlängen repräsentieren die benötigte Berechnungszeit. Kurze Balken bedeuten bessere Ergebnisse. Die Balkenlängen sind in den einzelnen Tests unterschiedlich skaliert.

8.2 Tests

Der erste Test (Abbildung 9) verwendet das Built-In-Prädikat `findall4/4`. Dieses Prädikat profitiert nicht von einer parallelen Verarbeitung. In diesem Test sollen lediglich die beiden Implementationen verglichen werden. Der Test zeigt, das die

Abk.	Programmaufruf
r0	`time ./cpl -o -f ../example/4gewinnt.pl`
r1	`time mpirun -np 1 -machinefile ~/machinefile ./cpl\ parallel -o -f ../example/4gewinnt.pl`
r4	`time mpirun -np 4 -machinefile ~/machinefile ./cpl\ parallel -o -f ../example/4gewinnt.pl`
r8	`time mpirun -np 8 -smp -machinefile ~/machinefile \ ./cplparallel -o -f ../example/4gewinnt.pl`

Tabelle 13: Programmaufrufe

Verwendung der MPI-Version den Zeitaufwand nur unwesentlich vergrössert. Auch wenn die Implementation selbst sehr langsam ist, wird der Aufwand durch das Hinzufügen von MPI wenig beeinflusst.

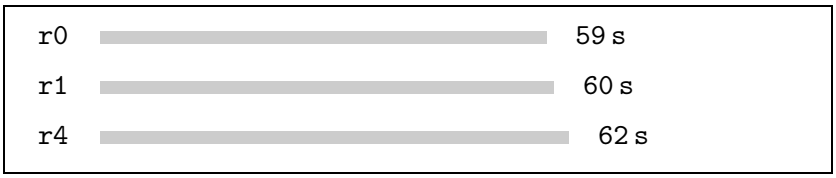

Abbildung 9: 5 Halbzüge, `findall4/4`

Die Tests in den Abbildungen 10, 11 und 12 verwenden das Built-In-Prädikat `findallparallel4/4` und zeigen den Performancegewinn, der durch die parallele Verarbeitung erzielt werden kann. In den Tests werden unterschiedliche Halbzugtiefen berechnet. Die verwendete Spielstellung ist dabei unausgewogen, d. h. die von den einzelnen Prozessen zu berechnenden Teilbäume sind unterschiedlich mächtig. Da die Klauseln statisch auf die Prozessoren verteilt werden, ist der Performancegewinn vom Verhältnis der einzelnen Teilbäume zum Gesamtbaum sowie dem Verhältnis der Klauselanzahl zur Anzahl der Prozessoren abhängig. Dennoch ist auch unter diesen ungünstigen Voraussetzungen der Gewinn beeindruckend.

Der letzte Test (Abbildung 13) verwendet als Spielstellung den Spielbeginn. Das bedeutet, das der Berechnungsaufwand weitgehend ausgeglichen ist und damit günstige Voraussetzungen für die parallele Verarbeitung gegeben sind. Der Test berechnet 6 Halbzüge und verkürzt bei 4 verwendeten Prozessoren die Berechnungszeit etwa um den Faktor 3.5. Bei 8 Prozessoren bleibt einer der Prozessoren unbeschäftigt und

die restlichen 7 erzielen einen Faktor von etwa 6.3.

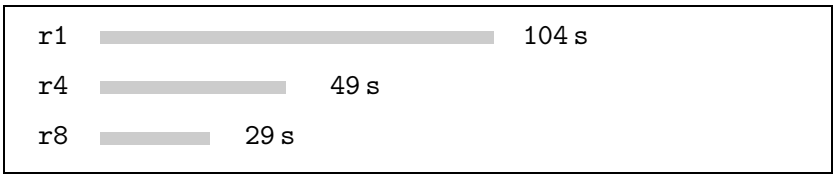

Abbildung 10: 5 Halbzüge, `findallparallel4/4`

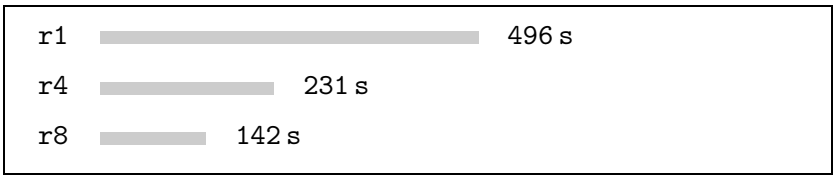

Abbildung 11: 6 Halbzüge, `findallparallel4/4`

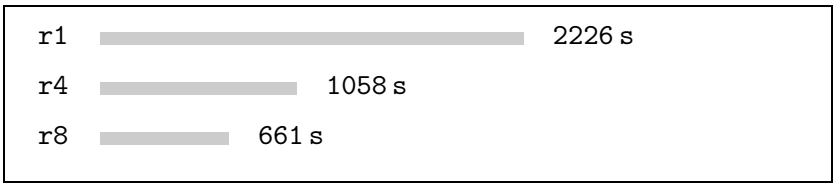

Abbildung 12: 7 Halbzüge, `findallparallel4/4`

8.3 Faktoren und Seiteneffekte

Neben der bereits erwähnten Relevanz der Klauselanzahl und der Ausgewogenheit des Berechnungsaufwandes in den einzelnen Klauseln, spielt der Zeitpunkt des Aufrufs von `findallparallel4/4` eine wesentliche Rolle. Je weiter oben im Suchbaum das Prädikat aufgerufen wird, desto weniger Speicher-Volumen muss über das Netzwerk ausgetauscht werden, da zu diesem Zeitpunkt die 'grossen' Speicher wie Heap, Stack und Strings noch einen überschaubaren Umfang besitzen. Wärend SMP-Maschinen über einen gemeinsamen Speicher verfügen, ist in Cluster-Systemen der Austausch von Daten relativ zeitaufwendig und ein wesentlicher Faktor für die Effizienz. Die Zahl der zur Verfügung stehenden Prozessoren beeinflusst die Geschwindigkeit nur solange, soweit genügend Klauseln vorhanden sind alle Prozessoren auszulasten. Es ist günstiger, viele weniger rechenintensive als wenige sehr aufwendige

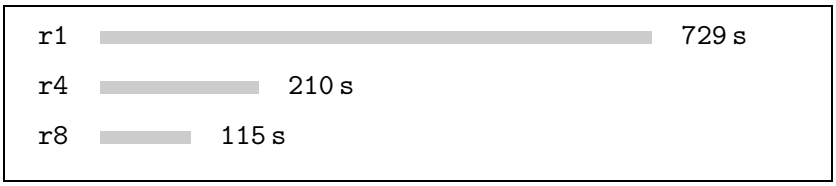

Abbildung 13: 6 Halbzüge, `findallparallel4/4` Spielbeginn

Klauseln zu parallelisieren, da die Wahrscheinlichkeit einer gleichmässigen Auslastung mit der Klauselzahl steigt.

Ein Problem, welches sich durch das Parallelisieren ergibt, ist, dass die Reihenfolge der Ergebnisse und insbesondere deren Berechnung, nicht erhalten bleibt. D. h., das Seiteneffekte falsch oder nur zum Teil berücksichtigt werden. Wird der Cut z. B. in einer der Klauseln, des von `findallparallel4/4` aufgerufenen Prädikats, verwendet, werden nur diejenigen Klauseln abgeschnitten, die im gleichen Prozess berechnet werden. Alle extralogischen Prädikate müssen mit Vorsicht verwendet werden und verschiedene Algorithmen lassen sich erst nach der Teilung der Klauseln effizient einsetzen. Der Alpha-Beta-Algorithmus z. B. propagiert seine Werte nur innerhalb eines Cluster-Knotens.

8.4 Schlussfolgerung und Ausblick

Der Pseudo-Interpreter `cpl` kann sich nicht mit bestehenden, kommerziell oder frei verfügbaren, Prolog-Systemen messen. Die Ausführungs-Geschwindigkeit sequenzieller Prolog-Programme und die Zahl der Built-In-Prädikate liegt weit hinter anderen Implementationen. Dennoch lassen sich damit auch komplexe Probleme in angemessener Zeit lösen. Alle bekannten Fehler der Implementation wurden behoben und in den aktuellen Versionen von `cpl` und `cplparallel` gab es bislang, auch bei rechenintensiven Prolog-Programmen, keine negativen Überraschungen.

Obwohl `cplparallel` in den untersuchten Testszenarios eine hohe Effizienz erzielt hat, gibt es eine Reihe von Erweiterungen, die in eine Folgeversion einfliessen sollten. Wie in den meisten anderen Systemen der Fall, sollte der Prolog-nach-WAM-Compiler in Prolog implementiert werden. Die in [5] vorgeschlagenen Optimierungen sollten vollständig umgesetzt und die Garbage-Collection verbessert werden. Um Rechenzeit einzusparen, wäre es notwendig, auch typisierte Zahlen anzubieten und den Umgang mit Operatoren zu standardisieren. Für den parallelen Teil wären Gewinne durch eine verbesserte Lastenverteilung zu erzielen.

Vor Erstellung dieser Arbeit wäre es dem Autor kaum möglich gewesen, ein bestehendes Prolog-System zu modifizieren. Nunmehr können die gewonnenen Erfahrungen dazu verwendet werden, ein freies System durch explizite ODER-Parallelität zu ergänzen und dadurch dessen Performance nochmals zu verbessern.

A Liste der Built-In-Prädikate

`!/0` oder `cut/0()`

Ist immer wahr und schneidet Choice-Points der vorhergehenden Teilziele der aktuellen Klausel sowie alle alternativen Klauseln des aktuellen Prädikates ab.

`true/0()`

Ist immer wahr.

`fail/0()`

Schlägt immer fehl und löst damit Backtracking aus.

`showheap/0()`

Zeigt den Heap an, wobei die einzelnen Zellen in Typ und Wert aufgelöst werden.

`showmem/1(+Segment)`

Zeigt den Inhalt des angegebenenen Segmentes an. Segment muss eine der folgenden Kontstanten sein: `code`, `heap`, `pdl` oder `stack`.

`gt/2(+Num1,+Num2)` oder `>/2(+Num1,+Num2)`

Wahr, falls `Num1` und `Num2` Zahlen und `Num1` die grössere der beiden.

`ge/2(+Num1,+Num2)` oder `>=/2(+Num1,+Num2)`

Wahr, falls `Num1` und `Num2` Zahlen und `Num1` grössere oder gleich `Num2`.

`eq/2(+Num1,+Num2)` oder `=:=/2(+Num1,+Num2)`

Wahr, falls `Num1` und `Num2` Zahlen und `Num1` gleich `Num2`.

`le/2(+Num1,+Num2)` oder `=</2(+Num1,+Num2)`

Wahr, falls `Num1` und `Num2` Zahlen und `Num1` kleiner oder gleich `Num2`.

`lt/2(+Num1,+Num2)` oder `>/2(+Num1,+Num2)`

Wahr, falls `Num1` und `Num2` Zahlen und `Num1` die kleinere der beiden.

`add/3(+Num1,+Num2,?Res)`

`Res` ist die Summe von `Num1` und `Num2`.

`sub/3(+Num1,+Num2,?Res)`

`Res` ist die Differenz zwischen `Num1` und `Num2` ist.

`mul/3(+Num1,+Num2,?Res)`

Wahr, falls `Res` das Produkt aus `Num1` und `Num2` ist.

`div/3(+Num1,+Num2,?Res)`

Wahr, falls `Res` das Ergebnis der gerundeten Division von `Num1` durch `Num2` ist.

`write/1(+Term)` oder `writeterm/1(+Term)`

Gibt einen Term nach `stdout` aus. Der Term kann eine Variable, eine Konstante oder ein komplexer Term sein. `write(hallo(X)).` wäre eine gültige Verwendung bei der `hallo` mit der Bindung von `X` ausgegeben wird.

`not/1(+Term)`

Wahr, falls `Term` ein Built-in- oder benutzerdefiertes Prädikat ist und der Aufruf dieses Prädikates scheitert.

`call/1(+Term)`

Wahr, falls `Term` ein Built-in- oder benutzerdefiniertes Prädikat ist und der Aufruf dieses Prädikates erfolgreich ist[34].

`assert/2(+Ident,+Term)`

Wahr, falls sich `Term` vom Heap nach Assert sichern und referenzieren lässt.

[34]Neben `call/1` existiert ein weiteres Built-In-Prädikat `callparallel/1`, welches das aufgerufene Prädikat für die parallele Bearbeitung markiert und von `findallparallel4/4` verwendet wird. Das Verhalten ausserhalb von `findallparallel4/4` ist unbestimmt.

`retract/2(+Ident,-Term)`

Wahr, falls in Assert ein Term mit dem Label `Ident` existiert und sich dieser in den Heap kopieren und an `Term` binden lässt.

`retractOrNIL/2(+Ident,-Term)`

Immer wahr. Wie `retract/2` oder `Term` wird an `[]` gebunden.

`findall4/4(+Var,+Goal,+Ident,-List)`

Erzeugt die Liste (`List`) aller Bindungen für `Var`, bei denen der Aufruf von `Goal` erfolgreich terminiert. Für `Goal` muss ein entsprechendes Prädikat existieren und die Variable `Var` in `Goal` enthalten sein. Die einzelnen Bindungen werden durch Backtracking erzeugt und temporär in Assert gespeichert. Die Prädikate `assert/2` und `retractOrNIL/2` verwenden die Konstante `Ident` als Identifier. Existiert keine Lösung wird `List` an `[]` gebunden.

`member/2(?Term,+List)`

Wahr, falls `Term` in `List` enthalten ist.

`memberchk/2(?Term,+List)`

Wie `member/2` lässt jedoch keine Choice-Points.

`writeID/0`

Immer wahr. Gibt auf `stdout` den Rang der Maschine aus, auf dem der Prozess läuft. Hilfreich, um anhand von Ausgaben zu debuggen.

`findallparallel4/4(+Var,+Goal,+Ident,-List)`

Ähnlich wie `findall4/4`, allerdings werden die einzelnen Klauseln von Goal gleichmässig auf die zur Verfügung stehenden Prozessoren verteilt und dort bearbeitet. `List` ist dann mit einer Liste von Listen instantiiert, die die jeweiligen Ergebnisse der einzelnen Knoten enthalten. Die Reihenfolge der Lösungen für `Var` ist dabei von der Zahl der verwendeten Knoten abhängig. Seiteneffekte in `Goal` bleiben auf den Knoten des Auftretens beschränkt.

`atom/1(+Term)`

Wahr, falls Term eine Konstante wie z. B. `x` ist.

`complex/1(+Term)`

Wahr, falls Term ein komplexer Term wie z. B. `x(1)` ist.

`unbound/1(+Term)`

Wahr, falls Term eine ungebundene Variable ist.

`isList/1(+Term)`

Wahr, falls Term eine korrekt formatierte Liste ist. Z. B. `l(1,l(2,[]))`

`isList/1(?List1,?List2,?List12)`

Wahr falls `List12` die Liste ist, welche entsteht, wenn man `List2` an `List1` anhängt.

B Quelltext von 4-Gewinnt

```
%******* Helfer ************************


neg(3,-3).neg(-3,3).
neg(2,-2).neg(-2,2).
neg(1,-1).neg(-1,1).
neg(0,0).

gt(3,2).gt(3,1).gt(3,0).gt(3,-1).gt(3,-2).gt(3,-3).
gt(2,1).gt(2,0).gt(2,-1).gt(2,-2).gt(2,-3).
gt(1,0).gt(1,-1).gt(1,-2).gt(1,-3).
gt(0,-1).gt(0,-2).gt(0,-3).
gt(-1,-2).gt(-1,-3).
gt(-2,-3).

sub(7,1,6).sub(6,1,5).sub(5,1,4).sub(4,1,3).
sub(3,1,2).sub(2,1,1).sub(1,1,0).

legal(0).legal(1).legal(2).legal(3).
legal(4).legal(5).



% findallparallel4 liefert eine liste von listen zurück, die
% wir als einfache Liste haben wollen
% flatten/2
% flatten(+List1,-List2)
flatten(l(X,[]),X):-!.
flatten(l(X,Y),Z):-
        flatten(Y,U),
        joinLists(X,U,Z).
```

```
besterZug([],R,R):-!.
besterZug(l(record(Zug, Wert),Y), record(ZugAlt,WertAlt), record(  ↩
Zug1, Wert1)):-
        besterZug(Y, record(ZugAlt,WertAlt), record(Zug1,Wert1)),
        gt(Wert1,Wert),!.
besterZug(l(record(Zug, Wert),Y), record(ZugAlt,WertAlt), record(  ↩
Zug,Wert)).




%******* Das Brett darstellen *********************

displayChip(s(o,Chip),Steine,Chip):-
        write('|#O#'),!.
displayChip(Special,Steine,Chip):-
        memberchk(s(o,Chip),Steine),
        write('| O '),!.
displayChip(s(x,Chip),Steine,Chip):-
        write('|#X#'),!.
displayChip(Special,Steine,Chip):-
        memberchk(s(x,Chip),Steine),
        write('| X '),!.
displayChip(Special,X,Chip):-
        write('|   ').


displayRow(Special,Steine,First):-
        add(First,10,Chip2),
        add(First,20,Chip3),
        add(First,30,Chip4),
        add(First,40,Chip5),
        add(First,50,Chip6),
        add(First,60,Chip7),
```

```
        displayChip(Special,Steine,First),
        displayChip(Special,Steine,Chip2),
        displayChip(Special,Steine,Chip3),
        displayChip(Special,Steine,Chip4),
        displayChip(Special,Steine,Chip5),
        displayChip(Special,Steine,Chip6),
        displayChip(Special,Steine,Chip7),
        write('|'),nl(),
        write('---------------------------'),
        nl().

display(stellung(S1,S2,S3,S4,S5,S6,S7,l(Special,Steine))):-
        displayRow(Special,Steine,16),
        displayRow(Special,Steine,15),
        displayRow(Special,Steine,14),
        displayRow(Special,Steine,13),
        displayRow(Special,Steine,12),
        displayRow(Special,Steine,11),nl().

display(stellung(S1,S2,S3,S4,S5,S6,S7,l(Steine))):-
        displayRow(0,Steine,16),
        displayRow(0,Steine,15),
        displayRow(0,Steine,14),
        displayRow(0,Steine,13),
        displayRow(0,Steine,12),
        displayRow(0,Steine,11),nl().


%***** Helfer für 4Gewinnt **************************


% aufBrett/3(+Stein1,+Stein2,+Stein3,+Stellung)
% wahr falls die Steine <Stein1>, <Stein2> und Stein3
% bereits Teil der Stellung <Stellung> sind.
aufBrett(Stein1,Stein2,Stein3,stellung(S1,S2,S3,S4,S5,S6,S7,Stell  ↩
ung)) :-
    memberchk(Stein1,Stellung),
```

```
        memberchk(Stein2,Stellung),
        memberchk(Stein3,Stellung).


% gewinnt/2(+Stein0,+Stellung)
% wahr falls durch Hinzufügen von <Stein0> die <Stellung> ein Gew ↩
inn
% für die Farbe von <Stein0> bedeutet
%vertikal
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,1,Feld1), add(Feld,2,Feld2), add(Feld,3,Feld3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-1,Feld1), add(Feld,1,Feld2), add(Feld,2,Feld3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-2,Feld1), add(Feld,-1,Feld2), add(Feld,1,Feld3) ↩
,
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-3,Feld1), add(Feld,-2,Feld2), add(Feld,-1,Feld3 ↩
),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
%horizontal
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,10,Feld1), add(Feld,20,Feld2), add(Feld,30,Feld3 ↩
),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
```

```
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-10,Feld1), add(Feld,10,Feld2), add(Feld,20,Feld ↩
3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-20,Feld1), add(Feld,-10,Feld2), add(Feld,10,Fel ↩
d3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-30,Feld1), add(Feld,-20,Feld2), add(Feld,-10,Fe ↩
ld3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
%diagonal
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,11,Feld1), add(Feld,22,Feld2), add(Feld,33,Feld3 ↩
),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-11,Feld1), add(Feld,11,Feld2), add(Feld,22,Feld ↩
3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-22,Feld1), add(Feld,-11,Feld2), add(Feld,11,Fel ↩
d3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
```

```
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-33,Feld1), add(Feld,-22,Feld2), add(Feld,-11,Fe ↩
ld3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-27,Feld1), add(Feld,-18,Feld2), add(Feld,-9,Fel ↩
d3),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-18,Feld1), add(Feld,-9,Feld2), add(Feld,9,Feld3 ↩
),
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,-9,Feld1), add(Feld,9,Feld2), add(Feld,18,Feld3) ↩
,
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.
gewinnt(s(Farbe,Feld),Stellung) :-
        add(Feld,9,Feld1), add(Feld,18,Feld2), add(Feld,27,Feld3) ↩
,
        aufBrett(s(Farbe,Feld1),s(Farbe,Feld2),s(Farbe,Feld3),Ste ↩
llung),
        !.


% gegnerFarbe/2(?Farbe1,?Farbe2).
% wahr falls Farbe2 die entgegengesetzte Farbe von Farbe1 ist
gegnerFarbe(x,o).
gegnerFarbe(o,x).
```

```
% figurZug/3(+Farbe,+Stellung,Zug)
% erfüllt falls es in der aktuellen <Stellung> für <Farbe> möglic  ↩
h ist
% <Zug> zu machen.
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(1,s(Farbe,Z))):  ↩
-
        legal(S1),
        add(S1,11,Z).
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(7,s(Farbe,Z))):  ↩
-
        legal(S7),
        add(S7,71,Z).
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(2,s(Farbe,Z))):  ↩
-
        legal(S2),
        add(S2,21,Z).
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(6,s(Farbe,Z))):  ↩
-
        legal(S6),
        add(S6,61,Z).
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(3,s(Farbe,Z))):  ↩
-
        legal(S3),
        add(S3,31,Z).
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(5,s(Farbe,Z))):  ↩
-
        legal(S5),
        add(S5,51,Z).
figurZug(Farbe,stellung(S1,S2,S3,S4,S5,S6,S7,X),z(4,s(Farbe,Z))):  ↩
-
        legal(S4),
        add(S4,41,Z).
```

```
% ziehe/2(StellungAlt,Zug,StellungNeu)
% erfüllt falls <StellungNeu> die um <Zug> erweiterte <StellungAl ↩
t> ist.
% D.h. die Steinliste um Stein erweitert wurde und der Wert der e ↩
ntsprechenden
% Spalte um 1 incrementiert wurde
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(1,Stein),stellung(Z,S2,S ↩
3,S4,S5,S6,S7,l(Stein,X))):-
        add(S1,1,Z).
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(2,Stein),stellung(S1,Z,S ↩
3,S4,S5,S6,S7,l(Stein,X))):-
        add(S2,1,Z).
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(3,Stein),stellung(S1,S2, ↩
Z,S4,S5,S6,S7,l(Stein,X))):-
        add(S3,1,Z).
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(4,Stein),stellung(S1,S2, ↩
S3,Z,S5,S6,S7,l(Stein,X))):-
        add(S4,1,Z).
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(5,Stein),stellung(S1,S2, ↩
S3,S4,Z,S6,S7,l(Stein,X))):-
        add(S5,1,Z).
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(6,Stein),stellung(S1,S2, ↩
S3,S4,S5,Z,S7,l(Stein,X))):-
        add(S6,1,Z).
ziehe(stellung(S1,S2,S3,S4,S5,S6,S7,X),z(7,Stein),stellung(S1,S2, ↩
S3,S4,S5,S6,Z,l(Stein,X))):-
        add(S7,1,Z).




%********** MiniMax **********************


% evaluateAndChoose/6(Zuege,Stellung,Level,Farbe,RecordAlt,Record ↩
```

```
Best)
% wahr falls RecordBest das beste Zug-Wert-Paar aus der Liste der ↩
 Zuege
% in der gegebenen Stellung für Farbe ist
evaluateAndChoose(l(Zug,Zuege),Stellung,Level,Farbe,RecordAlt,Rec ↩
ordBest) :-
   ziehe(Stellung,Zug,Stellung1),
   minimax(Level, Stellung1, Farbe, record(Zug1,Wert1)),          ↩

   neg(Wert1,Wert),
   update(Zug,Wert,RecordAlt,RecordNeu),
   evaluateAndChoose(Zuege,Stellung,Level,Farbe,RecordNeu,RecordB ↩
est).

evaluateAndChoose([],Stellung,Level,Farbe,Record,Record).


% checkDirekt/4(+Zuege,+Stellung,-Zug,-Wert)
% wahr falls Zug in der Liste der Zuege ist und in Stellung einen ↩
 Gewinn
% für einen der Spieler wäre.
% Ein direkter Gewinn bedeutet einen Wert von 2 die Verhinderung  ↩
des
checkDirekt([],Stellung,z(Spalte,Stein),0):-
    fail(),!.
checkDirekt(l(z(Spalte,Stein),Zuege),Stellung,z(Spalte,Stein),2): ↩
-
    gewinnt(Stein,Stellung),!.
checkDirekt(l(z(Spalte,Stein),Zuege),Stellung,z(Spalte2,Stein2),W ↩
ert):-
    checkDirekt(Zuege,Stellung,z(Spalte2,Stein2),Wert).



% doit/6(+Zuege,+Stellung,+Level,+Farbe,+RecordAlt,-record(Zug,We ↩
rt))
% wie evaluateAndChoose/6. Statt direkt zu evaluateAndChoose zu g ↩
```

```
ehen
% wird zunächst auf diekten Gewinn oder Verhinderung von Verlust
% geprüft. Die Zweite Klausel reicht die Anfrage evaluateAndChoos  ↩
e/6 weiter.
% doit existiert weil wir kein oder haben. siehe minimax.
doit(Zuege,Stellung,Level,Farbe,RecordAlt,record(Zug,Wert)) :-
   checkDirekt(Zuege,Stellung,Zug,Wert),
    !.
doit(Zuege,Stellung,Level,Farbe,RecordAlt,RecordBest) :-
    evaluateAndChoose(Zuege,Stellung,Level,Farbe,RecordAlt,Record  ↩
Best).




% minimax/4(Level,Stellung,Farbe,RecordNeu)
% wahr falls RecordNeu das beste Zug-Wert-Paar bei der Level-tief  ↩
en
% Untersuchung des Spielbaumes aus Stellung für Farbe ist.
minimax(0,Stellung,Farbe,record(Zug,0)):-
        !.
minimax(Level,Stellung,Farbe,RecordNeu) :-
        findall4(Zug,figurZug(Farbe,Stellung,Zug),tmp,Zuege),
        sub(Level,1,LevelNeu),
        gegnerFarbe(Farbe,FarbeNeu),
        doit(Zuege,Stellung,LevelNeu,FarbeNeu,record(z(0,0),-3),R  ↩
ecordNeu).




% update/4(ZugNeu,WertNeu,RecordAlt,RecordNeu)
% wahr falls das Zug-Wert-Paar RecordNeu besser oder gleich dem
% Zug-Wert-Paar RecordAlt ist.
update(ZugNeu,WertNeu,record(ZugAlt,WertAlt),record(ZugNeu,WertNe  ↩
```

```
u)) :-
        gt(WertNeu,WertAlt),
        !.
update(ZugNeu,WertNeu,RecordAlt,RecordAlt).




%***** Prädikat für MiniMax der 1. Ebene für findall(parallel)4 *  ↩
*****


toplevel(HT,Stellung,Amzug,record(z(1,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(1,Stein)),
        ziehe(Stellung,z(1,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
iteID(),nl().

toplevel(HT,Stellung,Amzug,record(z(2,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(2,Stein)),
        ziehe(Stellung,z(2,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
```

```
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
iteID(),nl().

toplevel(HT,Stellung,Amzug,record(z(3,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(3,Stein)),
        ziehe(Stellung,z(3,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
iteID(),nl().

toplevel(HT,Stellung,Amzug,record(z(4,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(4,Stein)),
        ziehe(Stellung,z(4,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
iteID(),nl().

toplevel(HT,Stellung,Amzug,record(z(5,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(5,Stein)),
        ziehe(Stellung,z(5,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
```

```
iteID(),nl().

toplevel(HT,Stellung,Amzug,record(z(6,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(6,Stein)),
        ziehe(Stellung,z(6,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
iteID(),nl().

toplevel(HT,Stellung,Amzug,record(z(7,Stein),Wert2)):-
        figurZug(Amzug,Stellung,z(7,Stein)),
        ziehe(Stellung,z(7,Stein),Stellung1),
        sub(HT,1,HTD),
        gegnerFarbe(Amzug,Gegner),
        minimax(HTD,Stellung1,Gegner,record(Zug1,Wert1)),
        neg(Wert1,Wert2),
        write('für Zug '),write(Stein),write(' in Stellung, liefe  ↩
rt minimax '),
        write(record(Zug1,Wert1)),write(' --- sagt prozess '), wr  ↩
iteID(),nl().


%**** Teststellungen  **********
teststellung1(stellung(1, 0, 4, 4, 0, 3, 4,l(s(o,11),l(s(x, 31),  ↩
l(s(o, 32), l(s(o, 33), l(s(x, 34), l(s(x, 41), l(s(o, 42), l(s(o  ↩
, 61), l(s(o, 62), l(s(x, 63), l(s(o,71), l(s(x,72), l(s(x, 73),  ↩
l(s(x,43), l(s(x, 75), l(s(o, 76), l(s(x, 44), l(s(o, 35), l(s(o,  ↩
 74), [])))))))))))))))))))).

teststellung2(stellung(0,0,0,0,0,0,0,l([]))).

teststellung3(stellung(3,0,0,0,0,0,0,
```

```
                    l(s(x,13),
                    l(s(x,12),
                    l(s(x,11),[])))))).


%**** Tests  **********
%findall
test1():-
        teststellung1(X),
        display(X),
        findall4(R,toplevel(4,X,x,R),grr,L),
        write(L),nl(),
        besterZug(L, record(11,-3), record(Zug,Wert)),
        write('besterZug :'),write(record(Zug,Wert)).

%findallparallel
test2():-
        teststellung2(X),
        display(X),
        findallparallel4(R,toplevel(6,X,x,R),grr,L),
        flatten(L,L2),
        write(L2),nl(),
        besterZug(L2, record(11,-3), record(Zug,Wert)),
        write('besterZug :'),write(record(Zug,Wert)).



test3():-
        teststellung2(X),
        display(X),
        minimax(3,X,x,record(Zug,Wert)),
        write('Der beste Zug : '),write(Zug),
        write(' liefert den Wert : '),write(Wert),nl().


%****** Querys ***********************************
```

```
?-test3().
```

Literatur

[1] D.M.Russinoff. A verified Prolog compiler for the Warren abstract machine. *The Journal of Logic Programming*, 13(1, 2, 3 & 4):367–412, 1992.

[2] E.Lusk, R.Butler, T.Disz, R.Olson, R.Overbeek, R.Stevens, D.H.D.Warren, A.Calderwood, P. Szeredi, S.Haridi, P.Brand, M.Carlsson, A.Ciepielewski, and B.Hausman. The aurora or-parallel prolog system. *New Generation Computing*, 1989.

[3] E.Morel, J.Briat, J.Kergommeaux, and C.Geyer. Side-effects in plosys or-parallel prolog on distributed memory machines, 1996.

[4] G.Gupta, E.Pontelli, K.A.M.Ali, M.Carlsson, and M.V.Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[5] H.Aït-Kaci. *Warren's Abstract Machine - A Tutorial Reconstruction*. Massachusetts Institute of Technology, 1991.

[6] H.K.Büning and S.Schmitgen. *Prolog*. Teubner, 1988.

[7] J.C.Cunha, P.D.Medeiros, M.B.Carvalhosa, and L.M.Pereira. Delta prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In P.Kacsuk and M.Wise, editors, *Implementations of Distributed Prolog*, pages 333–356. Wiley & Sons, 1992.

[8] K.A.M.Ali and R.Karlsson. The Muse approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[9] K.L.Clark. Parlog and its applications. *IEEE Transactions on Software Engineering*, 14(12):1792–1804, 1988.

[10] K.Villaverde, E.Pontelli, H.F.Guo, and G.Gupta. Pals: An or-parallel implementation of prolog on beowulf architectures. In *Logic Programming, 17th International Conference, ICLP 2001*, volume 2237, pages 27–42. Springer, 2001.

[11] K.Villaverde, H.Guo, E.Pontelli, and G.Gupta. Incremental stack-splitting mechanisms for efficient parallel implementation of search-based AI systems. In *Proceedings of 2001 International Conference on Parallel Processing (30th ICPP'01)*, Valencia, Spain, 2001. Universidad Politecnia de Valencia.

[12] L.Sterling and E.Shapiro. *The Art of Prolog*. Massachusetts Institute of Technology, 1986.

[13] M.Hermenegildo. Some methodological issues in the design of ciao - a generic, parallel, concurrent constraint system, 1994.

[14] M.Hermenegildo, F.Bueno, D.Cabeza, M.Carro, M.G.d.l.Banda, P.López, and G.Puebla. The ciao parallel execution environment for (c)lp languages: A progress report (extended abstract).

[15] P.Deransart, L.Cervoni, and A.Ed-Dbali. *Prolog: The Standard*. Springer Verlag, 1996.

[16] R.Rocha, F.M.A.Silva, and V.S.Costa. Yapor: an or-parallel prolog system based on environment copying. In *Portuguese Conference on Artificial Intelligence*, pages 178–192, 1999.