Entwurf und Implementierung eines übersetzenden Systems für das intuitionistische logische Programmieren basierend auf der Warren Abstract Machine

Christian Schulte

Diplomarbeit

Betreuer:
Prof. Dr. P. Deussen
Dipl.–Inform. C. Brzoska

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe (TH)

27. Juli 1992

Zusammenfassung

Das Hornklauselfragment der Prädikatenlogik, das bisher hauptsächlich als Grundlage für das logische Programmieren dient, insbesondere für die Programmiersprache PROLOG, ist in seiner Ausdrucksmächtigkeit sehr eingeschränkt. In den letzten Jahren sind viele Erweiterungen der Hornklausellogik untersucht worden. Eine Erweiterung, die durch zusätzliche Typen von Formeln und entsprechende Ableitungsregeln diese Einschränkungen teilweise aufhebt, ist das *intuitionistische logische Programmieren.*

Die theoretischen Grundlagen dieser Erweiterung sind in der Literatur schon umfangreich untersucht worden. Im Gegensatz dazu ist der praktische Aspekt dieser Erweiterung noch weitgehend unberücksichtigt geblieben. Diese Arbeit beschäftigt sich mit der Entwicklung, Implementierung und Untersuchung eines kompletten und praktikablen Systems für das intuitionistische logische Programmieren.

Einleitend werden die Konzepte, die diese Erweiterung ausmachen, vorgestellt, und die Sprache, ein erweiterter Unifikationsalgorithmus und die zugehörige prozedurale Semantik definiert.

Dieses liefert den Ausgangspunkt für die praktische Umsetzung. Es wird eine Erweiterung der „Warren Abstract Machine“ (WAM) entworfen, deren Instruktionssatz als Zielsprache für die Übersetzung intuitionistischer logischer Programme dient. Bei dem Entwurf wird insbesondere auf die Möglichkeit zur Integration anderer Erweiterungen des logischen Programmierens, die in ihrer Realisierung auch auf die WAM aufsetzen, geachtet.

Basierend auf dem KA–PROLOG System, wird ein komplettes System für das intuitionistische logische Programmieren realisiert. Dieses System umfaßt die vorher entworfene Maschine, den zugehörigen Übersetzer und weitere Komponenten, wie zum Beispiel einen Debugger und ein Modulkonzept.

Die Untersuchung der Praktikabilität hinsichtlich der effizienten Unterstützung der Paradigmen des intuitionistischen logischen Programmierens durch Laufzeitmessungen und Untersuchung der gemessenen Werte bildet die Grundlage für die Beurteilung und Diskussion des realisierten Systems. Die Diskussion berücksichtigt neben dem Vergleich des erhaltenen mit einem bereits existierenden System auch Schwächen und Erweiterungsmöglichkeiten der entworfenen und realisierten Systemkomponenten. Ein Ausblick auf weitere Möglichkeiten schließt die Arbeit ab.

Erklärung

Ich habe die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt.

7500 Karlsruhe, den 27. Juli 1992

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Intuitionistisches logisches Programmieren . . . 1

1.1.1 Hypothetische Implikationen . . . 1

1.1.2 Explizite Quantifizierung . . . 2

1.1.3 Gemeinsamkeiten . . . 4

1.1.4 Wieso keine klassische Logik? . . . 5

1.2 Ausgangspunkt und Zielsetzung . . . 5

1.3 Überblick . . . 6

2 Sprache und Beweisverfahren **9**

2.1 Zugrundeliegende Syntax . . . 9

2.2 Erweitert bereichskorrekte Unifikatoren . . . 10

2.3 Beweisverfahren . . . 16

2.4 Korrektheit und Vollständigkeit . . . 19

2.5 Vorüberlegungen zur Implementierung . . . 19

3 Entwurf der I–WAM **21**

3.1 Vorüberlegungen . . . 21

3.2 Entwurfskriterien . . . 22

3.3 Unifikation . . . 22

3.3.1 Repräsentation von Termen . . . 23

3.3.2 Unifikationsalgorithmus . . . 24

3.4 Explizite Quantifizierung . . . 26

3.4.1 Bereitstellung von Argumenten . . . 27

3.4.2 Unifikation von Köpfen . . . 31

3.5 Hypothetische Implikationen . . . 34

3.5.1 Implikationspunkte . . . 35

3.5.2 Geteilte Variablen . . . 37

3.5.3 Bereitstellung dynamischer Klauseln . . . 40

3.5.4 Benutzung dynamischer Klauseln . . . 41

3.6 Sonstiges . . . 43

3.7 Zwei abschließende Beispiele . . . 44

4 Das KAIN–PROLOG System **47**
4.1 Schichtenmodell . . . 47
4.2 Die Programmiersprache . . . 47
4.3 Der Übersetzer . . . 48
4.3.1 Bezeichneranalyse . . . 49
4.3.2 Variablenklassifikation . . . 50
4.3.3 Erzeugung von Klauselcode . . . 51
4.3.3.1 Übersetzungsschemata . . . 51
4.3.3.2 Optimierungen . . . 56
4.3.4 Erzeugung von Indexcode . . . 58
4.4 Modulverwaltung . . . 58
4.5 Der Debugger . . . 59
4.5.1 Anforderungen . . . 59
4.5.2 Explizite Quantifizierung . . . 59
4.5.3 Hypothetische Implikationen . . . 60

5 Praktikabilität des KAIN–PROLOG Systems **63**
5.1 Speicherbedarf . . . 63
5.2 Laufzeitverhalten . . . 64
5.2.1 Prozeduraufrufe . . . 65
5.2.2 Verwaltung der Wahlpunkte . . . 66
5.2.3 Unifikation . . . 68
5.2.4 Programmix . . . 70
5.3 Praktikabilität . . . 70
5.3.1 Sortieren . . . 70
5.3.2 Das n–Damen Problem . . . 73
5.3.3 Der Cocke–Younger–Kasami–Algorithmus . . . 74
5.3.4 Weitere Beispiele . . . 75
5.3.4.1 Deduktive Datenbanken . . . 75
5.3.4.2 Generierung von Lemmata . . . 76

6 Diskussion und Ausblick **77**
6.1 Vergleich mit einem anderen Ansatz . . . 77
6.1.1 Datenrepräsentation . . . 77
6.1.2 Explizite Quantifizierung . . . 77
6.1.3 Hypothetische Implikationen . . . 78
6.1.4 Fazit . . . 79
6.2 Bewertung . . . 79
6.2.1 I–WAM . . . 79
6.2.1.1 Unifikation und explizite Quantifizierung . . . 79
6.2.1.2 Hypothetische Implikationen . . . 80
6.2.2 Übersetzung . . . 80
6.2.3 Standardprädikate . . . 81
6.2.4 Modulkonzept . . . 81
6.3 Ausblick . . . 82

6.3.1 Integration anderer Erweiterungen 82
6.3.1.1 Ordnungssortiertes logisches Programmieren 82
6.3.1.2 Logisches Programmieren mit Randbedingungen 83
6.3.2 Komplexität der WAM 83

A Instruktionen und Operationen der I–WAM 85
A.1 Datenrepräsentation 85
A.2 Register 87
A.3 Arbeitsbereiche 88
A.3.1 Die Halde 89
A.3.2 Der Kontrollstapel 89
A.3.2.1 Umgebungen 89
A.3.2.2 Wahlpunkte 90
A.3.2.3 Implikationspunkte 92
A.3.3 Der Spurstapel 93
A.4 Maschinenoperationen 94
A.5 Operationen zur Unifikation 95
A.6 `get`–Instruktionen 99
A.7 `unify`–Instruktionen 100
A.8 `put`–Instruktionen 102
A.9 `set`–Instruktionen 106
A.10 Kontrollinstruktionen 108
A.11 Indexinstruktionen 109

B Sprachdefinition und Bedienungsanleitung 113
B.1 Modulstruktur 113
B.1.1 Deklaration 113
B.1.1.1 Operatordeklaration 114
B.1.1.2 Sichtbarkeitsdeklaration 114
B.1.2 Implementierung 115
B.1.2.1 Klauseln 115
B.1.2.2 Abarbeitung 116
B.2 Standardmodule 117
B.2.1 Das Modul `public` 117
B.2.2 Weitere Module 118
B.3 Bedienung 119
B.3.1 Toplevel 119
B.3.1.1 Setzen des aktuellen Moduls 119
B.3.1.2 Sichtbare Prädikate 119
B.3.1.3 Anfragen an das aktuelle Modul 119
B.3.1.4 Toplevel–Prädikate 120
B.3.2 Debugger 121
B.3.2.1 Toplevel–Prädikate für den Debugger 121
B.3.2.2 Debugger–Kommandos 122
B.3.3 Beispielsitzung 122

B.4 Installation 124

C Beispielprogramme **127**

C.1 Sortieren 127

C.2 Das n–Damen Problem 129

C.3 Der Cocke–Younger–Kasami–Algorithmus 131

Literaturverzeichnis **135**

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit dem *intuitionistischen logischen Programmieren*, einer Erweiterung des logischen Programmierens mit Hornklauseln. Auf Zielsetzung und Inhalte der Arbeit wird nach Vorstellung des intuitionistischen logischen Programmierens eingegangen.

1.1 Intuitionistisches logisches Programmieren

In den hier betrachteten Typen von logischen Formeln können in den Rümpfen von Klauseln wieder Implikationen auftreten. Diese werden dann *hypothetische* oder *eingebettete* Implikationen genannt. Zusätzlich ist es erlaubt, daß in den Rümpfen sowohl Existenz- als auch Allquantoren vorkommen. Wie die operationale Semantik dieser beiden Erweiterungen aussieht, und wie man diese benutzen kann, wird als nächstes kurz dargestellt. Eine detailiertere Einführung findet man in [GR84, Sch91b].

1.1.1 Hypothetische Implikationen

Eine hypothetische Implikation ist eine Formel der Bauart $C \rightarrow G$, wobei C eine Klausel und G eine Anfrage ist. Hat man ein Programm $\mathbf{P}$, aus dem diese Anfrage abgeleitet werden soll, so geht man wie folgt vor:

- Die Klausel C, also die *hypothetische* Information, wird dem Programm $\mathbf{P}$ hinzugefügt; man erhält das um die hypothetische Information erweiterte Programm $\mathbf{P} \cup \{C\}$.
- Aus diesem Programm $\mathbf{P} \cup \{C\}$ wird die neue Anfrage G abgeleitet.
- Nach dem Ableiten von G wird die hypothetische Information entfernt; man erhält wieder das Ausgangsprogramm $\mathbf{P}$.

Dieses Vorgehen entspricht gerade dem Deduktionstheorem. Läßt man als Prämisse einer hypothetischen Implikation eine Konjunktion von Klauseln $C_1 \wedge \cdots \wedge C_k$ zu, so kann die Ableitbarkeit — bezeichnet durch $\vdash$ — folgendermaßen zusammengefaßt werden:

$$\mathbf{P} \vdash C_1 \wedge \cdots \wedge C_k \rightarrow G \iff \mathbf{P} \cup \{C_1, \ldots, C_k\} \vdash G.$$

An dem folgenden Beispiel ist zu erkennen, wie sich hypothetische Implikationen benutzen lassen.

Beispiel 1.1 (Studienberatung) Man betrachte das logische Programm $\mathbf{P}$, das zur Beratung von Studenten — an einer utopischen Universität — vor dem Vordiplom dient:

$$\begin{aligned} \mathit{vordiplom} &\leftarrow \mathit{vorlesung}(\mathit{analysis}) \wedge \mathit{vorlesung}(\mathit{informatik}) \wedge \mathit{vorlesung}(\mathit{algebra}), \\ \mathit{auszeichnung} &\leftarrow \mathit{vordiplom} \wedge \mathit{vorlesung}(\mathit{logik}). \end{aligned}$$

Möchte jetzt zum Beispiel ein Student wissen, der sein Vordiplom hat, ob es ausreicht, die Logik–Vorlesung zu besuchen, um eine Auszeichnung zu bekommen, so kann er die Anfrage

$$\mathit{vordiplom} \wedge \mathit{vorlesung}(\mathit{logik}) \rightarrow \mathit{auszeichnung}$$

an das Programm stellen. Die Anfrage wird erfolgreich sein: Nachdem dem Programm die beiden Fakten *vordiplom* und $\mathit{vorlesung}(\mathit{logik})$ der Konjunktion hinzugefügt worden sind, ist die neue Anfrage *auszeichnung* aus dem erweiterten Programm $\mathbf{P} \cup \{\mathit{vordiplom}, \mathit{vorlesung}(\mathit{logik})\}$ ableitbar.

Ein weiterer Student, der schon die Vorlesungen Analysis und Algebra besucht hat, möchte wissen, welche Vorlesung er noch besuchen muß, um sein Vordiplom zu bekommen: Stellt er die Anfrage

$$\mathit{vorlesung}(\mathit{analysis}) \wedge \mathit{vorlesung}(\mathit{algebra}) \wedge \mathit{vorlesung}(x) \rightarrow \mathit{vordiplom}$$

an das Programm, so werden die drei Konjunkte $\mathit{vorlesung}(\mathit{analysis})$, $\mathit{vorlesung}(\mathit{algebra})$ und $\mathit{vorlesung}(x)$ der Anfrage dem Programm hinzugefügt, und es wird an das erweiterte Programm $\mathbf{P} \cup \{\mathit{vorlesung}(\mathit{analysis}), \mathit{vorlesung}(\mathit{algebra}), \mathit{vorlesung}(x)\}$ die Anfrage *vordiplom* gestellt: Diese ist erfolgreich und liefert als Antwort für x die Vorlesung *informatik*. □

Bei der Ableitung der zweiten Anfrage in dem obigen Beispiel wurde dem Programm eine Klausel hinzugefügt, die eine freie Variable enthält. Bei Benutzung dieser Klausel lieferte uns dies eine Antwortsubstitution für die darin auftretende Variable. Insbesondere wurde diese Variable bei Benutzung der Klausel *nicht umbenannt*.

Das Stellen der Anfrage $C(x) \rightarrow G(x)$ an ein Programm $\mathbf{P}$ bedeutet also: Gesucht ist ein x, so daß $\mathbf{P} \cup \{C(x)\} \vdash G(x)$ gilt. Wir wollen hier die sich durch dieses Auftreten von Variablen ergebenden Möglichkeiten nicht weiter diskutieren. In dem weiteren Verlauf der Arbeit wird klar werden, daß diese Eigenschaft außerordentlich viele interessante Einsatzmöglichkeiten bietet.

1.1.2 Explizite Quantifizierung

Um eine Anfrage $\forall x\, G(x)$ aus einem Programm $\mathbf{P}$ abzuleiten, werden wir wie folgt vorgehen: Wir erzeugen eine neue Konstante c, die weder in $G(x)$ noch in den Klauseln von $\mathbf{P}$ vorkommt. Dann leiten wir die Anfrage $G(c)$, in der wir die Vorkommen von x durch c ersetzt haben, aus dem Programm ab. Dieses „übliche“ Vorgehen können wir kurz so umreißen:

$$\mathbf{P} \vdash \forall x\, G(x) \iff \mathbf{P} \vdash G(c), \quad c \text{ neue Konstante.}$$

Bevor wir uns jetzt auch den Existenzquantoren zuwenden, wollen wir ein Beispiel betrachten, in dem die Einsatzmöglichkeiten von Allquantoren sehr schön zu Tage treten.

Beispiel 1.2 (Sterilisation) Wir haben folgende Situation in einem biologischen Labor:

1. Ein Behälter ist steril, wenn jeder Keim in diesem Behälter tot ist.
2. Ein Keim ist tot, wenn er sich in einem Behälter befindet, der erhitzt wird.
3. Wir wollen nun wissen, ob ein erhitzter Behälter steril ist.

Nun übersetzen wir diese Situation in logische Formeln:

1. $\mathit{steril}(x) \leftarrow \mathit{behälter}(x) \wedge \forall y\,(\mathit{keim}(y) \wedge \mathit{in}(y,x) \rightarrow \mathit{tot}(y))$
2. $\mathit{tot}(x) \leftarrow \mathit{keim}(x) \wedge \mathit{behälter}(x) \wedge \mathit{in}(x,y) \wedge \mathit{erhitzt}(y)$
3. $\forall x\,(\mathit{behälter}(x) \wedge \mathit{erhitzt}(x) \rightarrow \mathit{steril}(x))$

Die erste und zweite Formel bilden jetzt unser logisches Programm **P**, das die Situation in dem Labor beschreibt, die dritte Formel ist die Anfrage an das Programm **P**.

Jetzt verfolgen wir die Ableitung dieser Anfrage aus dem Programm **P**. Der erste Schritt besteht darin, eine neue Konstante c für die allquantifizierte Variable x zu erzeugen. Wir erhalten also die neue Anfrage $\mathit{behälter}(c) \wedge \mathit{erhitzt}(c) \rightarrow \mathit{steril}(c)$. Die beiden Konjunkte der Prämisse der hypothetischen Implikation fügen wir dem Programm hinzu. Damit haben wir das neue Programm $\mathbf{P} \cup \{\mathit{behälter}(c), \mathit{erhitzt}(c)\}$ und die neue Anfrage $\mathit{steril}(c)$.

Im nächsten Schritt unifizieren wir wie gewöhnlich die Anfrage mit der ersten Klausel des Programms; damit haben wir als neue Anfrage $\mathit{behälter}(c) \wedge \forall y\,(\mathit{keim}(y) \wedge \mathit{in}(y,c) \rightarrow \mathit{tot}(c))$. Das erste Atom der Anfrage unifiziert jetzt mit dem ersten Faktum des erweiterten Programms, als neue Anfrage resultiert dann also $\forall y\,(\mathit{keim}(y) \wedge \mathit{in}(y,c) \rightarrow \mathit{tot}(c))$. Die nächsten beiden Schritte erzeugen wiederum eine neue Konstante d für die allquantifizierte Variable y, und die beiden Konjunkte der Prämisse der hypothetischen Implikation erweitern nochmals das Programm. Damit erhalten wir das neue Programm $\mathbf{P} \cup \{\mathit{behälter}(c), \mathit{erhitzt}(c), \mathit{keim}(d), \mathit{in}(d,c)\}$ und die neue Anfrage $\mathit{tot}(c)$.

Durch Unifizieren dieser Anfrage mit der ersten Klausel aus **P** erhalten wir die neue Anfrage $\mathit{keim}(c) \wedge \mathit{behälter}(c) \wedge \mathit{in}(c,z) \wedge \mathit{erhitzt}(z)$. Diese Atome lassen sich alle wie gewohnt aus den hinzugefügten atomaren Klauseln des Programms ableiten.

Damit haben wir die Antwort „ja“ auf unsere Frage, ob erhitzte Behälter steril sind. □

Die in dem obigen Beispiel diskutierte Situation ist einem Aufsatz von John McCarthy [McC87] entnommen. In diesem wird insbesondere kritisiert, daß es in dem Hornklauselfragment der Prädikatenlogik nicht möglich ist, dieses Problem zu formalisieren. Im Hornklauselfragment, also auch in der Programmiersprache PROLOG, kann jede Bakterie nur einzeln „getötet“ werden. Es gibt keine Möglichkeit eine Aussage zu formalisieren oder abzuleiten, die das eigentliche Konzept der Sterilisation beschreibt: Alle Bakterien auf einmal durch Erhitzen abzutöten. Als Essenz gibt McCarthy an:

> *“Reasoning and problem–solving programs must eventually allow the full use of quantifiers and sets, and have strong enough control methods to use them without combinatorial explosion.”*

Durch die Verbindung von Allquantoren und hypothetischen Implikationen hat man einen mächtigen Formalismus: Man kann hypothetische Information spezifizieren, die nicht nur auf

ein bestimmtes Individuum zutrifft, sondern universell für alle Individuen. Zusätzlich hat man ein zielgerichtetes Vorgehen bei der Ableitung solcher universeller Aussagen.

Welche Semantik Existenzquantoren besitzen, werden wir jetzt klären. Bedeutung ist ihnen erst dann beizumessen, wenn sie zusammen mit Allquantoren auftreten. Dazu betrachten wir zuerst ein Beispiel.

Beispiel 1.3 (Existenzquantoren) Gegeben sei das Programm

$$\mathbf{P} = \{p(z, z)\}.$$

Stellen wir an das Programm **P** die Anfrage

$$\exists x \forall y \, p(x, y),$$

so erwarten wir, daß diese Anfrage fehlschlägt: Wir können x nicht so wählen, daß für alle y gilt: $x = y$.

Stellen wir jedoch die Anfrage

$$\forall y \exists x \, p(x, y)$$

an das Programm **P**, so soll diese Anfrage ableitbar sein: Wir können x gerade als die Konstante c wählen, die für den Beweis von $\forall y$ erzeugt wurde. □

Die operationale Semantik eines Existenzquantors kann also jetzt so wiedergegeben werden:

$$\mathbf{P} \vdash \exists x \, G(x) \iff \mathbf{P} \vdash G(x), \quad \text{x darf nur an einen Term gebunden werden, der nur bisher schon erzeugte Konstanten für allquantifizierte Variablen enthält.} \tag{1.1}$$

1.1.3 Gemeinsamkeiten

Sowohl die hypothetischen Implikationen als auch die explizite Quantifizierung lassen sich beide als verschiedene Ausprägungen desselben Konzeptes sehen: Einführung von Gültigkeitsbereichen.

Bei der Abarbeitung einer hypothetischen Implikation $C \to G$ wird für die Abarbeitung von G ein Gültigkeitsbereich definiert: Nur in diesem Bereich steht die Klausel C bei der Abarbeitung von G zur Verfügung. Hier führen wir also einen Bereich ein, der die Gültigkeit von Klauseln für Prädikate betrifft.

Bei der Abarbeitung eines All- oder Existenzquantors erzeugen wir dagegen Gültigkeitsbereiche für Individuen. Wird die Konstante c bei der Abarbeitung von $\forall x \, G(x)$ erzeugt, so ist diese nur bei der Abarbeitung von $G(c)$ gültig.

Diese Gültigkeitsbereiche bereichern das logische Programmieren erstmals um ein Konzept, das in anderen Programmiersprachen selbstverständlich ist. Man denke dabei nur an das Blockkonzept in imperativen Programmiersprachen wie ALGOL 68 oder Pascal.

1.1.4 Wieso keine klassische Logik?

Nachdem nun die Grundzüge der operationalen Semantik für diese Erweiterung bekannt sind, drängt sich die Frage auf, wieso diese unter dem Begriff *intuitionistisches* logisches Programmieren zusammengefaßt werden. Erklärt man die deklarative Semantik intuitionistischer logischer Programme bezüglich der *intuitionistischen Logik*, so ist die prozedurale Semantik diesbezüglich vollständig und korrekt. Dies wird erst nach einem formalen Ausbau des Beweisverfahrens in Kapitel 2 zur Sprache kommen. Hier wollen wir zumindest klarmachen, daß wir bezüglich der klassischen Logik keine Vollständigkeit erreichen können.

Beispiel 1.4 Die Formel $((p \rightarrow q) \rightarrow p) \rightarrow p$, in anderer Schreibweise $(p \leftarrow (p \rightarrow q)) \rightarrow p$, ist eine klassische Tautologie. Denn belegt man p mit „wahr", so ist die äußerste Implikation auf Grund der Wahrheit der Konklusion wahr. Belegt man p mit „falsch", so ist $p \rightarrow q$ wahr, also $((p \rightarrow q) \rightarrow p$ falsch, und die ganze Formel somit wahr.

Wäre das bisher skizzierte Beweisverfahren bezüglich der klassischen Logik vollständig, so müßte die obige Formel aus dem leeren Programm ableitbar sein. Das gilt nicht, wie die folgende Ableitung zeigt.

$$\begin{aligned} \{\} \vdash (p \leftarrow (p \rightarrow q)) \rightarrow p \quad &\Longleftrightarrow \quad \{p \leftarrow (p \rightarrow q)\} \vdash p \\ &\Longleftrightarrow \quad \{p \leftarrow (p \rightarrow q)\} \vdash p \rightarrow q \\ &\Longleftrightarrow \quad \{p \leftarrow (p \rightarrow q), p\} \vdash q \end{aligned}$$

□

Nun ist es aber auch nicht so, daß der Ableitbarkeitsbegriff ungeschickt gewählt ist. Vielmehr will man keine klassische Logik als deklarative Semantik. Die hypothetische Implikation $p \rightarrow q$ soll bedeuten, daß unter der Annahme von p die Aussage q ableitbar ist. Durch die klassische logische Äquivalenz von $p \rightarrow q$ zu $\neg p \vee q$ hätte man jedoch folgende Semantik: $p \rightarrow q$ ist ableitbar, wenn q ableitbar oder p nicht ableitbar ist. Diese Semantik ist jedoch im Kontext des logischen *Programmierens* nicht wünschenswert, ja sogar kontraintuitiv.

Besonders kraß tritt die Kontraintuitivität zu Tage, wenn wir auch Disjunktionen betrachten. Denn die Anfrage $(p \rightarrow q) \vee r$ ist klassisch logisch äquivalent zu $(p \rightarrow q) \vee (p \rightarrow r)$: Das unterläuft die Idee, der Ableitung von q die hypothetische Information p zur Verfügung zu stellen, der Ableitung von r jedoch nicht.

1.2 Ausgangspunkt und Zielsetzung

Es hat sich in meiner Studienarbeit [Sch91b] gezeigt, daß das intuitionistische logische Programmieren viele interessante Aspekte bietet, die nahelegen, sich intensiv mit dieser Erweiterung des logischen Programmierens für Hornklauseln zu beschäftigen. Es gibt mehrere Ansätze in der Literatur, die sich nur mit hypothetischen Implikationen [GR84, Gab85, Mil89a] beschäftigen, und andere, die auch den Aspekt der expliziten Quantifizierung berücksichtigen [McC88a, McC88b, Mil89b, Mil90]. Einen Überblick und Vergleich dieser Ansätze ist in [Sch91b] zu finden.

Die deklarative Semantik des intuitionistischen logischen Programmierens ist auch schon sehr gut erforscht, hier gibt es viele interessante Ergebnisse, die diese Erweiterung als eine natürliche des logischen Programmierens mit Hornklauseln darstellt.

Der entscheidene Punkt jedoch, ob sich ein logisches Konzept als eine Programmiersprache durchsetzt, ist die Möglichkeit einer effizienten Ausführung dieser Programme. Von besonderer Wichtigkeit ist hierbei die Übersetzung logischer Programme in eine Folge von Operationen, die sich durch jeden Rechner ausführen lassen.

Bei der Sprache PROLOG ist hier als bahnbrechende Errungenschaft die abstrakte Maschine von Warren [War83], im weiteren kurz als WAM bezeichnet, zu nennen: Sie erlaubt es, durch einen einfachen Prozeß PROLOG–Programme in eine Folge von Instruktionen — WAM–Instruktionen — für diese Maschine zu übersetzen und effizient auszuführen. Die Maschine ist dabei in der Regel abstrakt als Softwaresimulation auf irgendeinem handelsüblichen Rechner implementiert.

Das angestrebte Ziel dieser Arbeit ist es, ausgehend von der WAM, eine Maschine zu entwerfen, deren Funktionalität so erweitert ist, daß ihre Instruktionen als Zielsprache für die Übersetzung intuitionistischer logischer Programme dienen kann. Es soll so behutsam erweitert werden, daß andere Erweiterungen des logischen Programmierens, die in ihrer Implementierung auf die WAM aufsetzen, weiterhin möglich bleiben. Daneben gilt es natürlich auch, den zugehörigen Übersetzer zu entwerfen, und beides zu implementieren. Das sich ergebende System heißt KAIN–PROLOG[1]. Zusätzlich soll gezeigt werden, daß dieses System *praktikabel* ist: Daß es mehr an Nutzen einbringt, als es Kosten mit sich bringt.

1.3 Überblick

In diesem Abschnitt werden kurz die einzelnen Kapitel der Arbeit und deren Inhalte vorgestellt.

Kapitel 2 Neben der Definition der Formeln, die in dieser Arbeit die Programmiersprache bilden, enthält dieses Kapitel einen erweiterten Unifikationsalgorithmus für Terme, die bei der Ableitung von expliziten Quantoren entstehen. Darauf folgt die Vorstellung des Beweisverfahrens, das Umreißen der deklarativen Semantik und Vorüberlegungen zur Umsetzung des Beweisverfahrens in eine konkrete Implementierung.

Kapitel 3 Hier wird die abstrakte Maschine für das intuitionistische logische Programmieren — I–WAM genannt — entworfen.

Kapitel 4 Dieses Kapitel behandelt die Realisierung des kompletten KAIN–PROLOG Systems, insbesondere wird der Übersetzungsvorgang und der Debugger beschrieben.

Kapitel 5 In diesem Kapitel wird das Laufzeitverhalten und der Speicherbedarf des KAIN–PROLOG Systems untersucht. Anhand praktischer Beispiele werden Paradigmen des intuitionistischen logischen Programmierens aufgezeigt, und die Praktikabilität des Systems wird nachgewiesen.

Kapitel 6 Das letzte Kapitel enthält neben einem Vergleich mit einer anderen Maschine für das intuitionistische logische Programmieren eine ausführliche Diskussion des erreichten Zieles der Arbeit und einen Ausblick auf nötige und wünschenswerte weitere Arbeiten.

[1]KAIN–PROLOG für Karlsruher intuitionistisches PROLOG

Anhang A Dieser Anhang enthält eine genaue und umfangreiche Definition der Instruktionen und Operationen der in Kapitel 3 entworfenen Maschine.

Anhang B Eine komplette Beschreibung der Implementierung des KAIN–PROLOG Systems aus Benutzersicht ist in diesem Anhang enthalten.

Anhang C Die Beispiele aus Kapitel 5 sind hier als Module des KAIN–PROLOG Systems aufgeführt.

Kapitel 2

Sprache und Beweisverfahren

In diesem Kapitel geht es darum, die Syntax der zu implementierenden Sprache festzulegen, das Beweisverfahren zu formulieren und auch anzugeben, wie die deklarative Semantik dieser Sprache ist.

2.1 Zugrundeliegende Syntax

Grundlage für die Definition der Sprache ist, wie sonst auch, der Begriff der Signatur für Terme und Formeln.

Definition 2.1 (Signatur) *Eine Signatur Σ ist ein Quadrupel $\langle C, F, P, \alpha \rangle$, wobei C die Menge der* Konstantensymbole*, F die abzählbare Menge der* Funktionssymbole *und P die abzählbare Menge der* Prädikatensymbole *ist. Die drei Mengen sind paarweise disjunkt. Die Abbildung $\alpha : F \cup P \to \mathbb{N}$ mit $\alpha(f) > 0$ für alle $f \in F$ ordnet jedem Symbol aus F und P seine* Stelligkeit *zu. Jedes Konstantensymbol $c^n \in C$ besitzt einen* Schachtelungsindex *$n \in \mathbb{N}$, und für jedes $n \in \mathbb{N}$ ist $\{c^n \mid c^n \in C\}$ abzählbar unendlich.*

Ist bei einer Konstanten kein Schachtelungsindex angegeben, so ist er gleich null. Die Trennung von Funktionssymbolen und Konstantensymbolen und die Einführung von Schachtelungsindices für Konstantensymbole hat den Zweck, die in Formel 1.1 auf Seite 4 angegebene Bedingung bei der Unifikation einzuhalten.

Auch die Variablen besitzen aus den gleichen Gründen wie die Konstantensymbole Schachtelungsindices.

Definition 2.2 (Variablen) *V ist die abzählbar unendliche Menge der* Variablen*. Jede Variable $v^n \in V$ besitzt einen* Schachtelungsindex *$n \in \mathbb{N}$, und für jedes $n \in \mathbb{N}$ ist die Menge $\{x^n \mid x^n \in V\}$ abzählbar unendlich.*

Ist der Schachtelungsindex einer Variablen nicht angegeben, so ist er gleich null. Die Menge der Terme ist jetzt bezüglich einer Signatur Σ und der Menge der Variablen V wie üblich definiert.

Definition 2.3 (Terme) *Die Menge der* Terme *$\mathcal{T}_\Sigma$ zur Signatur $\Sigma = \langle C, F, P, \alpha \rangle$ und zur Menge der Variablen V ist die kleinste Menge mit folgenden Eigenschaften:*

1. *$C \cup V \subseteq \mathcal{T}_\Sigma$,*

2. *Ist $f \in F$ mit $k = \alpha(f)$, und sind $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$, dann ist $f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$.*

Zu einem Term $t \in \mathcal{T}_\Sigma$ gibt $\mathrm{var}(t)$ die Menge der Variablen an, die in t vorkommen, und $\mathrm{con}(t)$ die Menge der Konstanten, die in t vorkommen. Jetzt sind wir in der Lage, die Menge der Klauseln und Anfragen zu definieren.

Definition 2.4 (Formeln) *Die Mengen der* atomaren Formeln $\mathcal{A}_\Sigma$, *der* Programmformeln *oder* Klauseln $\mathcal{C}_\Sigma$ *und der* Anfrageformeln *oder* Anfragen $\mathcal{G}_\Sigma$ *zu der Signatur* $\Sigma = \langle C, F, P, \alpha \rangle$ *und der Menge der Variablen V sind wie folgt definiert:*

- $\mathcal{A}_\Sigma = \{p(t_1, \ldots, t_k) \mid p \in P, \alpha(p) = k \text{ und } t_1, \ldots, t_k \in \mathcal{T}_\Sigma\}$.
- *$\mathcal{C}_\Sigma$ ist die kleinste Menge mit folgenden Eigenschaften:*
  1. *$\mathcal{A}_\Sigma \subseteq \mathcal{C}_\Sigma$.*
  2. *Falls $G \in \mathcal{G}_\Sigma$ und $A \in \mathcal{A}_\Sigma$, dann gilt: $A \leftarrow G \in \mathcal{C}_\Sigma$.*
  3. *Falls $C \in \mathcal{C}_\Sigma$ und $x \in V$, dann gilt: $\forall x\, C \in \mathcal{C}_\Sigma$.*
- *$\mathcal{G}_\Sigma$ ist die kleinste Menge mit folgenden Eigenschaften:*
  1. *$\mathcal{A}_\Sigma \subseteq \mathcal{G}_\Sigma$.*
  2. *Sind $G_1, G_2 \in \mathcal{G}_\Sigma$, so sind $G_1 \wedge G_2, G_1 \vee G_2 \in \mathcal{G}_\Sigma$.*
  3. *Sind $C_1, \ldots, C_k \in \mathcal{C}_\Sigma$ und $G \in \mathcal{G}_\Sigma$, so ist $C_1 \wedge \cdots \wedge C_k \rightarrow G \in \mathcal{G}_\Sigma$.*
  4. *Ist $x \in V$ und $G \in \mathcal{G}_\Sigma$, so sind $\exists x\, G, \forall x\, G \in \mathcal{G}_\Sigma$.*

Zu einer Formel $F \in \mathcal{A}_\Sigma \cup \mathcal{C}_\Sigma \cup \mathcal{G}_\Sigma$ gibt $\mathrm{free}(F)$ die wie üblich definierte Menge der freien Variablen in F an.

Im weiteren sei die Menge V der Variablen und eine Signatur $\Sigma = \langle C, F, P, \alpha \rangle$ fest vorgegeben. Soweit nicht anders vermerkt, bezeichnen c^{n_c}, d^{n_d} Konstanten, x^{n_x}, y^{n_y}, z^{n_z}, u^{n_u}, v^{n_v}, w^{n_w} Variablen, f, g, h Funktionssymbole und s, t, s_i, t_i Terme.

2.2 Erweitert bereichskorrekte Unifikatoren

In diesem Abschnitt wird die Unifikation von Termen dahingehend erweitert, daß die „Neuheit“ von Konstanten, die bei dem Beweis eines Allquantors entstehen, korrekt behandelt wird. Den Einstieg bildet die grundlegende Definition der Substitution.

Definition 2.5 (Substitution) *Eine* Substitution *ist eine Abbildung $\sigma : V \rightarrow \mathcal{T}_\Sigma$, die für fast alle Variablen die Identität ist. Für eine gegebene Substitution σ bezeichnet* $\mathrm{dom}(\sigma) = \{x^n \mid \sigma(x^n) \neq x^n\}$ *den* Definitionsbereich von σ, *und* $\mathrm{cod}(\sigma) = \{t \mid t = \sigma(x^n), x^n \in \mathrm{dom}(\sigma)\}$ *den* Wertebereich von σ.

Ist σ eine Substitution, so schreiben wir σ auch als $\{x_1^{n_1} \mapsto t_1, \ldots, x_k^{n_k} \mapsto t_k\}$, falls $\mathrm{dom}(\sigma) = \{x_1^{n_1}, \ldots, x_k^{n_k}\}$ und $\sigma(x_i^{n_i}) = t_i$ für $i = 1, \ldots, k$. Insbesondere wird die identische Substitution durch $\{\}$ bezeichnet. In der üblichen Art und Weise wird eine Substitution σ auf Terme und Formeln zu einer Abbildung σ^* fortgesetzt, und diese Abbildung σ^* wieder als σ bezeichnet. Auch die Definition des Unifikators ist wie üblich.

Definition 2.6 (Unifikator) *Ein* Unifikator *zweier Terme t_1 und t_2 ist eine Substitution σ mit $\sigma(t_1) = \sigma(t_2)$. Ein Unifikator θ ist ein* allgemeinster Unifikator *der Terme t_1 und t_2, wenn für jeden Unifikator σ von t_1 und t_2 eine Substitution λ existiert mit $\sigma = \lambda \circ \theta$.*

Das eigentliche Problem, das es zu lösen gilt, ist, wie oben schon erwähnt, die „Neuheit“ von Konstanten, die bei dem Beweis einer allquantifizierten Anfrage erzeugt werden, zuzusichern. Betrachten wir dazu schematisch die folgende Situation:

$$\exists x \, \overbrace{\forall y \, \exists z \, \underbrace{\forall u \, G(x, y, z, u)}_{\text{Bereich für } u}}^{\text{Bereich für } y}.$$

Bei dem Beweis dieser Anfrage werden wir für die existentiell quantifizierten Variablen x und z durch Unifikation Substitutionen berechnen, für die allquantifizierten Variablen y und u neue Konstanten erzeugen.

Betrachtet man noch einmal die Relation (1.1) auf Seite 4, so heißt die „Neuheit“ der Konstanten, die für y und u erzeugt werden, nichts anderes, als daß diese nur in den oben skizzierten Bereichen für y respektive u auftreten dürfen. In diesem Beispiel heißt das: Die für y erzeugte Konstante darf in keinem Term auftreten, an den x gebunden wird, jedoch in einem Term, an den z gebunden wird, und die für u erzeugte Konstante darf weder in einem Term vorkommen, an den x gebunden wird, noch in einem Term, an den z gebunden wird. Intuitiv bedeutet das, daß wenn x an einen Term gebunden würde, der die für y erzeugte Konstante enthält, so verließe diese Konstante den oben skizzierten Bindungsbereich des entsprechenden Quantors. Wir suchen also nach Substitutionen für existentiell quantifizierte Variablen, die die Bindungsbereiche nicht verletzen.

Um diese Einschränkungen bei der Bindung von existentiell quantifizierten Variablen einzuhalten, gehen wir wie folgt vor: Wir geben den existentiell quantifizierten Variablen und den Konstanten, die für die allquantifizierten Variablen erzeugt werden, Schachtelungsindices, die die Bindungsbereiche der Allquantoren repräsentieren. In unserer Situation also:

$$G(x^0, c^1, z^1, d^2),$$

wobei c^1 und d^2 die Konstanten sind, die für y, beziehungsweise für u, erzeugt wurden.

Jetzt reduziert sich das Problem, die korrekten Bindungsbereiche einzuhalten, darauf, sicherzustellen, daß x^0 nicht an einen Term gebunden wird, der eine Konstante mit einem Schachtelungsindex größer als 0 enthält, und z^1 nicht an einen Term, der eine Konstante mit einem Schachtelungsindex größer als 1 enthält.

Die oben geschilderte Idee fassen wir nun in der nächsten Definition zusammen.

Definition 2.7 (Bereichskorrektheit) *Eine Substitution* $\sigma = \{x_1^{n_1} \mapsto t_1, \ldots, x_k^{n_k} \mapsto t_k\}$ *ist* bereichskorrekt *genau dann, wenn* $n_i \geq \max\{n \mid c^n \in \mathrm{con}(t_i)\}$ *für* $i = 1, \ldots, k$[1].

Diese Einschränkung von Substitutionen ist jedoch noch zu schwach, wie das nächste Beispiel zeigt.

Beispiel 2.1 Sei $t_1 = f(x^1, h(y^2))$ und $t_2 = f(g(c^1, y^2), h(g(z^2, d^2)))$. Bei der Berechnung eines Unifikators erhalten wir zuerst die Substitution $\sigma_1 = \{x^1 \mapsto g(c^1, y^2)\}$, die wir erfolgreich auf Bereichskorrektheit geprüft haben. Dann wird die Substitution $\sigma_2 = \{y^2 \mapsto g(z^2, d^2)\}$ berechnet, die auch bereichskorrekt ist. Bildet man jedoch die Komposition $\sigma_2 \circ \sigma_1 = \{x^1 \mapsto g(c^1, g(z^2, d^2)), y^2 \mapsto g(z^2, d^2)\}$, so entsteht eine Substitution, die nicht mehr bereichskorrekt ist, da d^2 in $(\sigma_2 \circ \sigma_1)(x^1)$ vorkommt. □

Betrachtet man das obige Beispiel, so wird das Problem klar: Beim Binden von x^1 an $g(c^1, y^2)$ muß zusätzlich vermerkt werden, daß y^2 in einem Term vorkommt, der an eine Variable gebunden ist, die den Schachtelungsindex 1 hat. Die Lösung liegt nun auf der Hand: Alle Variablen in einem Term, die einen Schachtelungsindex besitzen, der größer ist als der Schachtelungsindex der Variable, an die dieser Term gebunden wird, müssen durch Variablen ersetzt werden, die den Schachtelungsindex der Variable besitzen, an die der Term gebunden wird. Diese Maßnahme wird zu Substitutionen führen, die auch unter Komposition ihre Bereichskorrektheit behalten. Dazu definieren wir:

Definition 2.8 (Erweiterte Bereichskorrektheit) *Eine bereichskorrekte Substitution* $\sigma = \{x_1^{n_1} \mapsto t_1, \ldots, x_k^{n_k} \mapsto t_k\}$ *ist genau dann* erweitert bereichskorrekt, *wenn für alle* i $(1 \leq i \leq k)$ *gilt:* $n_i \geq \max(\{n \mid x^n \in \mathrm{var}(t_i)\})$.

Nun überzeugen wir uns davon, daß erweitert bereichskorrekte Substitutionen auch die Eigenschaft haben, die wir verlangen.

Lemma 2.1 *Seien* σ *und* τ *erweitert bereichskorrekte Substitutionen. Dann ist auch* $\tau \circ \sigma$ *erweitert bereichskorrekt.*

Beweis Sei $x^n \in \mathrm{dom}(\tau \circ \sigma)$. Wir müssen zeigen, daß $n \geq p$ für $c^p \in \mathrm{con}((\tau \circ \sigma)(x^n))$ und $n \geq q$ für $y^q \in \mathrm{var}((\tau \circ \sigma)(x^n))$. Dazu unterscheiden wir zwei Fälle.

1. $x^n \in \mathrm{dom}(\sigma)$. Sei nun $y^m \in \mathrm{var}(\sigma(x^n))$. Damit gilt nun $n \geq m$. Ist $y^m \notin \mathrm{dom}(\tau)$, so sind wir fertig. Andernfalls sei also $\tau(y^m) = t \neq y^m$. Sei $c^p \in \mathrm{con}(t)$: Dann gilt $n \geq m \geq p$, also auch $n \geq p$. Dasselbe gilt auch für $u^q \in \mathrm{var}(t)$: Also $n \geq m \geq q$, damit $n \geq q$.
2. $x^n \notin \mathrm{dom}(\sigma)$. Also ist $(\tau \circ \sigma)(x^n) = \tau(x^n)$. Damit folgen die zu beweisenden Eigenschaften direkt aus der erweiterten Bereichskorrektheit von τ.

□

[1] Hier und auch im weiteren soll $\max(\emptyset) = 0$ sein.

Nachdem wir die Ideen für die Berechnung eines erweitert bereichskorrekten allgemeinsten Unifikators entwickelt haben, brauchen wir noch eine Definition zur Formulierung des Unifikationsalgorithmus:

Definition 2.9 (Differenzmenge zweier Terme) *Die* Differenzmenge $D(\{s,t\})$ *der Terme* s *und* t *ist wie folgt definiert:*

$$D(\{s,t\}) = \begin{cases} \{s\} & : \quad s = t \\ D(\{s_l,t_l\}) & : \quad \begin{array}{l} s \neq t, s = f(s_1,\ldots,s_k), t = f(t_1,\ldots,t_k) \textit{ wobei} \\ s_i = t_i \textit{ für } i = 1,\ldots,l-1 < k \textit{ und } s_l \neq t_l \end{array} \\ \{s,t\} & : \quad \textit{sonst} \end{cases}$$

Nun kann ein Algorithmus zur Berechnung erweitert bereichskorrekter Unifikatoren präsentiert werden. Der vorgestellte Algorithmus geht von dem Robinson–Algorithmus [Rob65] aus, erweitert um die oben präsentierten Ideen.

Algorithmus 2.1 (Berechnung erweitert bereichskorrekter Unifikatoren)

Eingabe Zwei Terme t_1 und t_2.

Ausgabe Abbruch oder $\langle \text{Erfolg}, \sigma \rangle$, wobei σ ein erweitert bereichskorrekter Unifikator für t_1 und t_2 ist.

Verfahren[2]

```
U ← {t₁, t₂};
σ ← {};
while #U ≠ 1 do
  if D(U) = {uⁿ, t} ∧ uⁿ ∈ V then
    if uⁿ ∈ var(t) ∨ n < max{m | cᵐ ∈ con(t)} then
      return Abbruch;
    else
      ρ ← {vᵐ ↦ wⁿ | vᵐ ∈ var(t), m > n, wⁿ neu};
      comment Anpassung der Schachtelungsindices der vᵐ
    fi;
    σ ← ρ ∘ {uⁿ ↦ t} ∘ σ;
    U ← (ρ ∘ {uⁿ ↦ t})(U);
  else
    return Abbruch;
  fi;
od;
return ⟨Erfolg, σ⟩;
```

□

[2] $\#M$ bezeichnet die Kardinalität der Menge M.

Wir wollen die einzelnen Schritte bei der Berechnung eines erweitert bereichskorrekten Unifikators an einem Beispiel verfolgen.

Beispiel 2.2 Die zu unifizierenden Terme sind $t_1 = f(g(x^2), x^2)$ und $t_2 = f(y^1, c^0)$. Bei dem Start des Algorithmus wird U mit $\{f(g(x^2), x^2), f(y^1, c^0)\}$ initialisiert, und die Substitution σ auf die Identität gesetzt. Jetzt betrachten wir die Durchläufe der **while**–Schleife:

1. Hier berechnet sich $D(U)$ zu $\{y^1, g(x^2)\}$. Das heißt, u^n ist gleich y^1 und t gleich $g(x^2)$, und der Schleifenrumpf wird betreten. Weder y^1 noch eine Konstante mit einem Schachtelungsindex größer als 1 kommt in $g(x^2)$ vor, also wird als nächstes die Substitution $\rho = \{x^2 \mapsto z^1\}$ berechnet, wobei z^1 eine neue Variable ist. Leicht zu sehen ist, daß nach Durchlauf der Schleife die Werte von σ und U wie folgt sind: $\sigma = \{x^2 \mapsto z^1, y^1 \mapsto g(z^1)\}$ und $U = \{f(g(z^1), z^1), f(g(z^1), c^0)\}$.

2. $D(U)$ ist jetzt $\{z^1, c^0\}$. Also wird auch jetzt der Schleifenrumpf betreten, und nach der Überprüfung von $t = c^0$ und Berechnung von $\rho = \{\}$ ergeben sich die folgenden Werte: $\sigma = \{x^2 \mapsto c^0, y^1 \mapsto g(c^0), z^1 \mapsto c^0\}$ und $U = \{f(g(c^0), c^0), f(g(c^0), c^0)\}$.

3. Dann berechnet sich $D(U)$ zu $\{f(g(c^0), c^0)\}$, das heißt die Kardinalität der Differenzmenge ist eins, und die Schleife terminiert.

Zurückgeliefert wird $\langle \text{Erfolg}, \{x^2 \mapsto c^0, y^1 \mapsto g(c^0), z^1 \mapsto c^0\}\rangle$. □

An dem obigen Beispiel wird noch ein Problem deutlich: Der berechnete Unifikator σ ist nicht allgemeinst. Ein allgemeinster Unifikator ist aber aus σ zu erhalten, wenn er auf die Variablen, die ursprünglich in den Eingabetermen vorkamen, eingeschränkt wird. Also muß in diesem Fall die Bindung $z^1 \mapsto c^0$ entfernt werden.

Nach dieser Vorüberlegung kann nun die Korrektheit des Algorithmus bewiesen werden, der Beweis ist eine Erweiterung des Beweises in [Rob65].

Satz 2.2 (Korrektheit des Algorithmus 2.1) *Seien $t_1, t_2 \in \mathcal{T}_\Sigma$ und $V_O = \text{var}(t_1) \cup \text{var}(t_2)$. Folgende Aussagen gelten für den Algorithmus 2.1:*

1. *Der Algorithmus terminiert für jede Eingabe t_1, t_2.*

2. *Terminiert der Algorithmus mit Ausgabe $\langle \text{Erfolg}, \sigma\rangle$, so ist σ ein erweitert bereichskorrekter Unifikator für t_1 und t_2.*

3. *Sei θ ein erweitert bereichskorrekter Unifikator für t_1 und t_2. Dann terminiert der Algorithmus mit der Ausgabe $\langle \text{Erfolg}, \sigma\rangle$, und es gibt eine Substitution λ mit $\theta|_{V_O} = (\lambda \circ \sigma)|_{V_O}$. Insbesondere ist $\sigma|_{V_O}$ ein erweitert bereichskorrekter allgemeinster Unifikator.*

4. *Der Algorithmus liefert genau dann* Abbruch *zurück, wenn t_1 und t_2 nicht erweitert bereichskorrekt unifizierbar sind.*

Beweis Bezeichnet werden mit U_k und σ_k die Werte der Programmvariablen U und σ beim k–ten Eintritt in die **while**–Schleife, also zum Beispiel ist U_1 gleich $\{t_1, t_2\}$. Analog sind die Bezeichnungen $u_k^{n_k}$, t_k und ρ_k zu verstehen, vorausgesetzt es werden die berechnenden Anweisungen hierfür erreicht.

1. U_{k+1} enthält echt weniger Variablen als U_k, da die Auftreten von $u_k^{n_k}$ in U_k durch t_k, wobei $u_k^{n_k}$ ja nicht in t_k vorkommen kann, ersetzt werden, und ρ_k die Anzahl der Variablen nicht verändert. Also muß der Algorithmus irgendwann mit $\langle \text{Erfolg}, \sigma \rangle$ terminieren, wenn nicht vorher schon Abbruch geliefert wurde.
2. Leicht ist durch Induktion zu zeigen, daß $U_k = (\sigma_k)(\{t_1, t_2\})$ ist. Wird irgendwann $\langle \text{Erfolg}, \sigma \rangle$ zurückgeliefert, so gilt $\#U_k = 1$, das heißt σ_k ist ein Unifikator für $\{t_1, t_2\}$. Die erweiterte Bereichskorrektheit folgt direkt aus dem Lemma 2.1, da $\rho_k \circ \{u_k^{n_k} \mapsto t_k\}$ nach der Definition von ρ_k selber erweitert bereichskorrekt ist.
3. Wir zeigen durch Induktion über k: Es wird nicht Abbruch ausgegeben, und es existiert ein λ_k mit $\theta|_{V_O} = (\lambda_k \circ \sigma_k)|_{V_O}$.
   Ist $k = 0$, so wählen wir λ_0 gerade als θ.
   Gelte nun $\theta|_{V_O} = (\lambda_k \circ \sigma_k)|_{V_O}$. Nach 2. gilt $(\sigma_k)(U_1) = U_k$. Da θ ein erweitert bereichskorrekter Unifikator für U_1 ist, ist λ_k ein bereichskorrekter Unifikator für $(\sigma_k|_{V_O})(U_1)$. Also ist $D(U_k) = \{u_{k+1}^{n_{k+1}}, t_{k+1}\}$ mit $u_{k+1}^{n_{k+1}} \in V$, denn sonst wäre U_k nicht unifizierbar. Da $\lambda_k(u_{k+1}^{n_{k+1}}) = \lambda_k(t_{k+1})$ ist, kommt $u_{k+1}^{n_{k+1}}$ nicht in t_{k+1} vor. Zusätzlich ist λ_k bereichskorrekt, also kann in t_{k+1} keine Konstante c^m mit $m > n_{k+1}$ vorkommen. Also wird im $(k+1)$–ten Schleifendurchlauf nicht Abbruch ausgegeben.
   Nun bleibt noch zu zeigen, daß es ein λ_{k+1} mit $\theta|_{V_O} = (\lambda_{k+1} \circ \sigma_{k+1})|_{V_O}$ gibt. Dazu definieren wir diese Substitution λ_{k+1} wie folgt:

$$\lambda_{k+1}(x^n) = \begin{cases} x^n & : \quad x^n = u_{k+1}^{n_{k+1}} \\ \lambda_k(y^m) & : \quad (y^m \mapsto x^n) \in \rho_{k+1} \\ \lambda_k(x^n) & : \quad \text{sonst} \end{cases} \tag{2.1}$$

   Sei jetzt $x^n \in V_O$. Zu zeigen ist $\theta(x^n) = (\lambda_{k+1} \circ \sigma_{k+1})(x^n)$. Sei dazu zuerst $x^n = u_{k+1}^{n_{k+1}}$. Damit ergibt sich:

$$\begin{array}{rcll} \theta(x^n) & = & (\lambda_k \circ \sigma_k)(x^n) & \text{Induktionsvorraussetzung} \\ & = & \lambda_k(x^n) & x^n \notin \text{dom}(\sigma_k) \\ & = & \lambda_k(t_{k+1}) & \text{Siehe weiter oben} \\ & = & \lambda_{k+1}(\rho_{k+1}(t_{k+1})) & \text{Siehe (2.2)} \\ & = & (\lambda_{k+1} \circ \sigma_{k+1})(x^n) & \text{Definition von } \sigma_{k+1} \end{array}$$

   Ist jetzt $x^n \neq u_{k+1}^{n_{k+1}}$, so gilt $\theta(x^n) = (\lambda_{k+1} \circ \sigma_{k+1})(x^n)$ trivialerweise. Denn da $x^n \in V_O$ ist, kann nicht $x^n \in \text{cod}(\rho_{k+1})$ gelten. Somit bleibt für dieses x^n alles beim Alten.
   Nachzutragen ist noch der Beweis der folgenden Gleichung:

$$\lambda_k(t_{k+1}) = \lambda_{k+1}(\rho_{k+1}(t_{k+1})) \tag{2.2}$$

   Wir zeigen diese Gleichheit, in dem wir dies für eine beliebige Variable $x^n \in \text{var}(t_{k+1})$ tun. Dazu betrachten wir zwei Fälle. Sei zuerst $x^n \notin \text{dom}(\rho_{k+1})$. Da x^n natürlich auch nicht gleich $u_{k+1}^{n_{k+1}}$ sein kann, gilt die Behauptung hier nach der dritten Alternative in der Definition von λ_{k+1}. Gilt $x^n \in \text{dom}(\rho_{k+1})$ mit $\rho_{k+1}(x^n) = y^m$, so schlägt gerade die zweite Alternative der Definition von λ_{k+1} zu: Es gilt dann $\lambda_{k+1}(\rho_{k+1}(x^n)) = \lambda_{k+1}(y^m) = \lambda_k(x^n)$, was zu zeigen war.

4. Folgt unmittelbar aus den drei obigen Aussagen.

□

Jetzt haben wir also einen Algorithmus zur Berechnung erweitert bereichskorrekter Unifikatoren, der sofort, wenn eine Bindung die Bereichskorrektheit verletzt, abbricht. Die Komplexität des Algorithmus ist genauso wie die des „normalen“ Unifikationsalgorithmus von Robinson exponentiell in der Komplexität der zu unifizierenden Terme. Es ist jedoch zu berücksichtigen, daß kein zusätzlicher Aufwand anfällt: Bei der beim „Occur–Check“ nötigen kompletten Durchmusterung des Terms, kann auch die Untersuchung auf die Bereichskorrektheit und die Anpassung der Schachtelungsindices vorgenommen werden.

Dadurch besteht die Möglichkeit, die Verbesserungen aus [CB83, RP89] durch den Übergang in der Repräsentation von Termbäumen zu Termgraphen zu integrieren. Somit kann einfach ein Algorithmus zur Berechnung erweitert bereichskorrekter Unifikatoren mit quadratischer Komplexität [CB83], beziehungsweise mit semilinearer Komplexität [RP89] erhalten werden.

2.3 Beweisverfahren

Jetzt geht es darum, Regeln für die Ableitung einer Anfrageformel aus einer Menge von Programmformeln festzulegen. Bei dem logischen Programmieren mit Hornklauseln verändert sich im Laufe der Ableitung nur die Anfrageformel. Bei dem intuitionistischen logischen Programmieren kann sich auch das Programm durch die Abarbeitung hypothetischer Implikationen ändern. Zusätzlich muß auch bei der Ableitung noch der aktuelle Schachtelungsindex berücksichtigt werden, der die Anzahl der schon betrachteten Allquantoren angibt. Also müssen bei den Ableitungsregeln diese drei veränderlichen Größen berücksichtigt werden.

Wird bei der Ableitung eine atomare Formel mit einer Klausel aus dem Programm unifiziert, so müssen vorher die Variablen dieser Klausel umbenannt und an den aktuellen Schachtelungsindex angepaßt werden. Um dieses korrekt erklären zu können, brauchen wir zwei Einschränkungen für Programme.

Definition 2.10 (Programm) *Ein* Programm *ist eine Menge* $\mathbf{P} \subseteq \mathcal{C}_\Sigma$*, so daß für alle* $C \in \mathbf{P}$ *gilt:* $\mathrm{free}(C) = \emptyset$ *und* $\max\{n \mid c^n \in \mathrm{con}(C)\} = 0$*.*

Die obige Definition verlangt, daß alle Variablen in der Klausel eines Programms gebunden auftreten, und daß alle Konstanten darin den Schachtelungsindex null haben. Die erste Forderung dient zur Anpassung der Schachtelungsindices von Variablen, die zweite verhindert, daß im Ausgangsprogramm Konstanten vorkommen, die eigentlich nur durch die Ableitung von Allquantoren erzeugt werden dürfen.

Die nächste Definition gibt nun sehr einfach die Umbenennung von Variablen und die Anpassung von Schachtelungsindices wieder.

Definition 2.11 *Die Abbildung* $new : \mathcal{C}_\Sigma \times \mathbb{N} \to \mathcal{C}_\Sigma$ *ist wie folgt definiert:*

$$new(C,n) = \begin{cases} \{x \mapsto y^n\}(new(C',n)) & : \quad C = \forall x\, C', y^n \textit{ neue Variable} \\ C & : \quad \textit{sonst} \end{cases}$$

$$\frac{\langle\{\langle G_1 \wedge G_2, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \theta\rangle}{\langle\{\langle G_1, \mathbf{P}, n\rangle, \langle G_2, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \emptyset\rangle} \wedge$$

$$\frac{\langle\{\langle G_1 \vee G_2, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \theta\rangle}{\langle\{\langle G_i, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \emptyset\rangle} \vee$$

$i = 1$ oder $i = 2$

$$\frac{\langle\{\langle C_1 \wedge \cdots \wedge C_k \to G, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \theta\rangle}{\langle\{\langle G, \mathbf{P} \cup \{C_1, \ldots, C_k\}, n\rangle\} \uplus \mathcal{S}, \emptyset\rangle} \to$$

$$\frac{\langle\{\langle \exists x\, G, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \theta\rangle}{\langle\{\langle \{x \mapsto y^n\}(G), \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \emptyset\rangle} \exists$$

y^n neue Variable

$$\frac{\langle\{\langle \forall x\, G, \mathbf{P}, n\rangle\} \uplus \mathcal{S}, \theta\rangle}{\langle\{\langle \{x \mapsto c^{n+1}\}(G), \mathbf{P}, n+1\rangle\} \uplus \mathcal{S}, \emptyset\rangle} \forall$$

c^{n+1} neue Konstante

$$\frac{\langle\{\langle A, \mathbf{P}, n\rangle\} \uplus \{\langle G_i, \mathbf{P}_i, n_i\rangle \mid 1 \leq i \leq k\}, \theta\rangle}{\langle\{\langle \sigma(G), \sigma(\mathbf{P}), n\rangle\} \uplus \{\langle \sigma(G_i), \sigma(\mathbf{P}_i), n_i\rangle \mid 1 \leq i \leq k\}, \sigma\rangle} \mathcal{C}$$

$A \in \mathcal{A}_\Sigma$, $A' \leftarrow G = \mathit{new}(C, n)$, $C \in \mathbf{P}$ und σ ist ein erweitert bereichskorrekter allgemeinster Unifikator von A und A'

$$\frac{\langle\{\langle A, \mathbf{P}, n\rangle\} \uplus \{\langle G_i, \mathbf{P}_i, n_i\rangle \mid 1 \leq i \leq k\}, \theta\rangle}{\langle\{\langle \sigma(G_i), \sigma(\mathbf{P}_i), n_i\rangle \mid 1 \leq i \leq k\}, \sigma\rangle} \mathcal{A}$$

$A \in \mathcal{A}_\Sigma$, $A' = \mathit{new}(C, n)$, $C \in \mathbf{P}$ und σ ist ein erweitert bereichskorrekter allgemeinster Unifikator von A und A'

Abbildung 2.1: Ableitungsregeln, $\uplus$ bezeichnet die disjunkte Vereinigung

Nach diesen Vorbereitungen kann nun der Ableitbarkeitsbegriff definiert werden:

Definition 2.12 (Ableitbarkeit) *Die Anfrage $G \in \mathcal{G}_\Sigma$ mit $\max(\{n \mid x^n \in \mathrm{free}(G)\} \cup \{n \mid c^n \in \mathrm{con}(G)\}) = 0$ ist aus dem Programm $\mathbf{P}$ mit Antwortsubstitution θ genau dann ableitbar, wenn:*

$$\langle\{\langle G, \mathbf{P}, 0\rangle\}, \{\}\rangle = \langle \mathcal{S}_0, \theta_0\rangle \vdash_1 \langle \mathcal{S}_1, \theta_1\rangle \vdash_2 \cdots \vdash_k \langle \mathcal{S}_k, \theta_k\rangle = \langle\{\}, \theta_k\rangle,$$

wobei $\vdash_i$ für $1 \leq i \leq k$ eine der Regeln aus Abbildung 2.1 ist, und:

$$\theta = \theta_k \circ \theta_{k-1} \circ \cdots \circ \theta_1.$$

Im nächsten Beispiel wird die Situation aus Beispiel 1.1 auf Seite 2 zur Verdeutlichung der Ableitbarkeit herangezogen:

Beispiel 2.3 (Studienberatung) Um die Übersicht bei der Ableitung nicht zu verlieren, ist hier das logische Programm **P** mit leicht verkürzten Bezeichnern wiedergegeben:

$$\begin{array}{rcl} vd & \leftarrow & vl(an) \land vl(in) \land vl(al), \\ az & \leftarrow & vd \land vl(lo). \end{array}$$

Die Abkürzungen sind: *vd* für *vordiplom*, *vl* für *vorlesung*, *an* für *analysis*, *in* für *informatik*, *al* für *algebra*, *lo* für *logik* und *az* für *auszeichnung*.

Die Ableitung der Anfrage $vl(an) \land vl(al) \land vl(x) \rightarrow vd$ aus dem obigen Programm **P** sieht dann wie folgt aus:

$$\begin{array}{c}
\langle\{\langle vl(an) \land vl(al) \land vl(x) \rightarrow vd, \mathbf{P}, 0\rangle\}, \{\}\rangle \quad \rightarrow \\ \hline
\langle\{\langle vd, \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle\}, \{\}\rangle \quad \mathcal{C} \\ \hline
\langle\{\langle vl(an) \land vl(in) \land vl(al), \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle\}, \{\}\rangle \quad \land \\ \hline
\langle\{\langle vl(an), \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle, \langle vl(in) \land vl(al), \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle\}, \{\}\rangle \quad \mathcal{A} \\ \hline
\langle\{\langle vl(in) \land vl(al), \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle\}, \{\}\rangle \quad \land \\ \hline
\langle\{\langle vl(in), \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle, \langle vl(al), \mathbf{P} \cup \{vl(an), vl(al), vl(x)\}, 0\rangle\}, \{\}\rangle \quad \mathcal{A} \\ \hline
\langle\{\langle vl(al), \mathbf{P} \cup \{vl(an), vl(al), vl(in)\}, 0\rangle\}, \{x \mapsto in\}\rangle \quad \mathcal{A} \\ \hline
\langle\{\}, \{\}\rangle
\end{array}$$

□

Das nächste Beispiel greift noch einmal das Beispiel 1.2 auf Seite 2 auf, wobei auch hier aus Gründen der Übersichtlichkeit eine etwas knappere Darstellung gewählt wurde:

Beispiel 2.4 (Sterilisierung) Das Programm **P** besteht aus den beiden folgenden Klauseln:

$$\begin{array}{rcl} \forall x\,(steril(x) & \leftarrow & \forall y\,(in(y,x) \rightarrow tot(y))) \\ \forall x\,(tot(x) & \leftarrow & \exists y\,(in(x,y) \land heiß(y))) \end{array}$$

Die Ableitung der Anfrage $\forall x\,(heiß(x) \rightarrow steril(x))$ aus dem Programm **P** sieht folgendermaßen aus:

$$\begin{array}{c}
\langle\{\langle \forall x\,(heiß(x) \rightarrow steril(x)), \mathbf{P}, 0\rangle\}, \{\}\rangle \quad \forall \\ \hline
\langle\{\langle heiß(c^1) \rightarrow steril(c^1), \mathbf{P}, 1\rangle\}, \{\}\rangle \quad \rightarrow \\ \hline
\langle\{\langle steril(c^1), \mathbf{P} \cup \{heiß(c^1)\}, 1\rangle\}, \{\}\rangle \quad \mathcal{C} \\ \hline
\langle\{\langle \forall y\,(in(y,c^1) \rightarrow tot(y)), \mathbf{P} \cup \{heiß(c^1)\}, 1\rangle\}, \{x_1^1 \mapsto c^1\}\rangle \quad \forall \\ \hline
\langle\{\langle in(d^2,c^1) \rightarrow tot(d^2), \mathbf{P} \cup \{heiß(c^1)\}, 2\rangle\}, \{\}\rangle \quad \rightarrow \\ \hline
\langle\{\langle tot(d^2), \mathbf{P} \cup \{heiß(c^1), in(d^2,c^1)\}, 2\rangle\}, \{\}\rangle \quad \mathcal{C} \\ \hline
\langle\{\langle \exists y\,(in(d^2,y) \land heiß(y)), \mathbf{P} \cup \{heiß(c^1), in(d^2,c^1)\}, 2\rangle\}, \{x_2^2 \mapsto d^2\}\rangle \quad \exists \\ \hline
\langle\{\langle in(d^2,z^2) \land heiß(z^2), \mathbf{P} \cup \{heiß(c^1), in(d^2,c^1)\}, 2\rangle\}, \{\}\rangle \quad \land \\ \hline
\langle\{\langle in(d^2,z^2), \mathbf{P} \cup \{heiß(c^1), in(d^2,c^1)\}, 2\rangle, \langle heiß(z^2), \mathbf{P} \cup \{heiß(c^1), in(d^2,c^1)\}, 2\rangle\}, \{\}\rangle \quad \mathcal{A} \\ \hline
\langle\{\langle heiß(c^1), \mathbf{P} \cup \{heiß(c^1), in(d^2,c^1)\}, 2\rangle\}, \{z^2 \mapsto d^2\}\rangle \quad \mathcal{A} \\ \hline
\langle\{\}, \{\}\rangle
\end{array}$$

□

2.4 Korrektheit und Vollständigkeit

In diesem Abschnitt bleibt noch zu klären, welche deklarative Semantik das oben formulierte Beweisverfahren besitzt. Es werden hier keine Beweise angegeben, die vorgestellten Ergebnisse sind der jeweils erwähnten Literatur zu entnehmen.

Der nächste Satz liefert den Zusammenhang zwischen dem Ableitbarkeitsbegriff aus Definition 2.12 und der Ableitbarkeit im intuitionistischen Prädikatenkalkül. Die Ableitbarkeit im intuitionistischen Prädikatenkalkül [Fit69, vD86] wird hier mit $\vdash_{\mathrm{I}}$ bezeichnen.

Satz 2.3 ([Mil89a, Mil90]) *Sei* $\mathbf{P}$ *ein Programm und* G *eine Anfrage.* G *ist aus* $\mathbf{P}$ *mit Antwortsubstitution* θ *genau dann ableitbar, wenn* $\mathbf{P} \vdash_{\mathrm{I}} \tilde{\exists} G$*, wobei* $\tilde{\exists} G$ *den existentiellen Abschluß von* G *bezeichnet. Weiterhin gilt auch* $\mathbf{P} \vdash_{\mathrm{I}} \theta(G)$*.*

In den oben referierten Literaturstellen wird bei dem Beweis die Unifikation außer Acht gelassen, die Aussagen werden nur für variablenfreie Formeln formuliert. Man kann sich aber leicht durch ein Lifting–Lemma davon überzeugen, daß die hier formulierte erweitert bereichskorrekte Unifikation die Korrektheit und Vollständigkeit nicht verletzen.

Im obigen Satz ist der Ableitungsbegriff, den wir zum intuitionistischen logischen Programmieren entwickelt haben, auf den „normalen“ Ableitungsbegriff in der intuitionistischen Logik zurückgeführt. Dann kann aber direkt der Begriff der logischen Konsequenz mittels *intuitionistischer Kripke–Modelle* [Kri65, Fit69, vD86] auf diesen Ansatz übertragen werden, das heißt, wenn $\Vdash$ die intuitionistische logische Konsequenzrelation bezeichnet, kann in dem obigen Satz $\vdash_{\mathrm{I}}$ durch $\Vdash$ ersetzt werden.

2.5 Vorüberlegungen zur Implementierung

Bevor in dem nächsten Kapitel mit dem Entwurf einer abstrakten Maschine begonnen wird, die Instruktionen anbietet, die zur Ausführung übersetzter Programme geeignet sind, wollen wir uns ein paar Gedanken darüber machen, wie die Berechnung von erweitert bereichskorrekten Unifikatoren und die Ableitungsregeln für eine Implementierung angepaßt werden müssen.

Bei der Berechnung von erweitert bereichskorrekten Unifikatoren haben wir die Anpassung von Schachtelungsindices dadurch formuliert, daß neue Variablen mit den „passenden“ Schachtelungsindices erzeugt werden. In einer Implementierung werden wir keine neuen Variablen erzeugen, sondern die Anpassung direkt vornehmen. Direkt heißt, daß in den Datenobjekten, die Variablen darstellen, die Komponente, die den Schachtelungsindex der Variablen darstellt, einfach mit dem angepaßten Schachtelungsindex überschrieben wird.

Bei der Auswahl einer Klausel aus einem Programm durch die Ableitungsregeln $\vdash^{\mathcal{C}}$ und $\vdash^{\mathcal{A}}$ haben wir vollkommene Freiheit, welche Klausel wir auswählen. In einer konkreten Implementierung wird diese Auswahl nach einer festgelegten Reihenfolge vorgenommen werden. Wir werden das Vorgehen, wie es die Programmiersprache PROLOG vorsieht, benutzen: Die Klausel eines Prädikates, die textuell zuerst im Programm vorkommt, wird zuerst benutzt; ist der Beweis mit der ausgewählten Klausel nicht zu erreichen, wird die jeweils nächste benutzt.

Zusätzlich haben wir jetzt aber auch noch Klauseln, die durch die Abarbeitung von hypothetischen Implikationen dem Programm hinzugefügt wurden. Hier werden wir die folgende

Strategie anwenden: Die zuletzt hinzugefügte Klausel wird zuerst benutzt; schlagen alle hinzugefügten Klauseln fehl, so werden in der oben beschriebenen Reihenfolge die „originalen“ Klauseln ausgewählt. Dabei ist zu beachten, daß wenn mehrere Klauseln desselben Prädikates bei der Abarbeitung einer hypothetischen Implikation hinzugefügt werden, die textuell zuerst erscheinenden Klauseln zuletzt hinzugefügt werden: Dadurch wird erreicht, daß die Auswahl der hinzugefügten Klauseln mit der textuellen Reihenfolge übereinstimmt.

Auch bei der Abarbeitung der Konjunktion und der Disjunktion nach den Ableitungsregeln $\vdash^{\wedge}$ und $\vdash^{\vee}$ haben wir die Freiheit, uns zu entscheiden, welche Teilformel zuerst betrachtet wird. Auch hier werden wir analog zu der Programmiersprache PROLOG vorgehen: Die jeweils linke Teilformel wird zuerst betrachtet.

Bei der Optimierung des Entwurfs werden wir das in dem folgenden Lemma formulierte Faktum heranziehen.

Lemma 2.4 *Sei die Anfrage G aus dem Programm $\mathbf{P}$ mit*

$$\langle\{\langle G, \mathbf{P}, 0\rangle\}, \{\}\rangle = \langle\mathcal{S}_0, \theta_0\rangle \vdash_1 \langle\mathcal{S}_1, \theta_1\rangle \vdash_2 \cdots \vdash_k \langle\mathcal{S}_k, \theta_k\rangle = \langle\{\}, \theta_k\rangle$$

ableitbar. Dann gilt für $i = 0, \ldots, k$ und $j = 1, \ldots, p_i$:

$$\max\left(\{m \mid x^m \in \mathrm{free}(G_{ij})\} \cup \{m \mid c^m \in \mathrm{con}(G_{ij})\}\right) \leq n_{ij},$$

mit

$$\mathcal{S}_i = \{\langle G_{ij}, \mathbf{P}_{ij}, n_{ij}\rangle \mid 1 \leq j \leq p_i\}.$$

Beweis Wir zeigen die folgende allgemeinere Behauptung mit Induktion über i: Für $j = 1, \ldots, p_i$ gilt

$$\max\left(\{m \mid x^m \in \mathrm{free}(G_{ij}) \cup \mathrm{free}(\mathbf{P}_{ij})\} \cup \{m \mid c^m \in \mathrm{con}(G_{ij}) \cup \mathrm{con}(\mathbf{P}_{ij})\}\right) \leq n_{ij} \qquad (2.3)$$

Diese Relation (2.3) ist für $i = 0$ aufgrund der Definitionen 2.10 und 2.12 erfüllt.

Jetzt betrachten wir beim Übergang von i nach $i + 1$ die einzelnen Ableitungsregeln. Die Regeln $\vdash^{\wedge}$, $\vdash^{\vee}$ und $\vdash^{\rightarrow}$ betreffen die Relation (2.3) überhaupt nicht. Bei den Regeln $\vdash^{\forall}$ und $\vdash^{\exists}$ werden Konstanten beziehungsweise Variablen erzeugt, die gerade dem aktuellen Schachtelungsindex entsprechen, somit gilt auch hier (2.3).

Da bei der Regel $\vdash^{\mathcal{A}}$ nichts übrig bleibt, gilt (2.3) trivialerweise. Bei $\vdash^{\mathcal{C}}$ sichert die Induktionsvorraussetzung zu, daß freie Variablen und Konstanten die geforderten Eigenschaften haben. Unter Berücksichtigung, daß die *new*–Operation nur zulässige freie Variablen für eventuell gebundenen Variablen erzeugt, folgt damit die Gültigkeit der Relation (2.3).

□

Kapitel 3

Entwurf der I–WAM

Das Herz der Arbeit ist dieses Kapitel, in dem die Erweiterung der Warren Abstract Machine [War83] — kurz als WAM bezeichnet — auf eine abstrakte Maschine für das intuitionistische logische Programmieren — dem zu Folge kurz als I–WAM bezeichnet — entworfen wird. Im Vordergrund stehen dabei die Ideen und Prinzipien, die bei der Erweiterung Eingang finden.

3.1 Vorüberlegungen

Für das Verständnis dieses Kapitels ist eine grobe Kenntnis der Konzepte der Maschine von Warren unerläßlich, neben der originalen Arbeit [War83] sind zum Einstieg in das Thema auch [AK90, Nei86] zu empfehlen.

Ausgangspunkt ist die Definition 2.12 der Ableitbarkeit auf Seite 17. Daraus werden alle nötigen Mechanismen für die I–WAM herauskristallisiert, umgesetzt und verfeinert. Vergleicht man den Ableitbarkeitsbegriff für das klassische logische Programmieren mit dem in Kapitel 2 für das intuitionistische logische Programmieren entwickelten, treten als Punkte, in denen die Erweiterung erfolgen muß, folgende hervor:

Unifikation Um erweitert bereichskorrekte Unifikatoren berechnen zu können, ist der Entwurf einer geeigneten Datenstruktur für Terme nötig. Weiterhin muß der Algorithmus 2.1 auf Seite 13 zur Berechnung erweitert bereichskorrekter Unifikatoren umgesetzt und in die Maschine integriert werden.

Explizite Quantifizierung Die Umsetzung der expliziten Quantifizierung wird vor allen Dingen eine Erweiterung des Instruktionssatzes um spezialisierte Unifikationsbefehle mit sich bringen. Dieses wird sich an zwei Punkten manifestieren: Einmal bei der Bereitstellung von Argumenten im Rumpf, das andere Mal bei der Unifikation von Köpfen von Klauseln.

Hypothetische Implikationen Dieser Punkt besitzt ebenso wie der obige zwei Hauptaspekte. Der eine ist, wie die Klauseln, die zu der Prämisse einer hypothetischen Implikation gehören, bei der Abarbeitung den schon bereits vorhandenen Klauseln hinzugefügt werden. Wie diese verfügbar gemachten Klauseln dann von anderen Klauseln per Prozeduraufruf benutzt werden, das bildet den zweiten Aspekt.

Eine umfassende Definition aller Operationen und Instruktionen der I–WAM ist im Anhang A zu finden.

3.2 Entwurfskriterien

Bevor mit der Vorstellung der Erweiterung begonnen werden kann, muß man sich darüber klar werden, welche Kriterien diese zu erfüllen hat. Einmal umfassen diese *äußere* Kriterien, die die Schnittstelle der Maschine, also den Instruktionssatz, betreffen; andererseits auch Kriterien, die die *interne Realisierung* der Maschine betreffen.

Das wohl wichtigste Kriterium bedarf eines Blickes über den „Zaun“: Das intuitionistische logische Programmieren stellt nur *eine* mögliche Erweiterung des klassischen logischen Programmierens mit Hornklauseln dar. Neben dieser gibt es auch andere Erweiterungen, deren Implementierungen auch auf der WAM aufsetzen, jedoch dem Paradigma des Programmierens in Logik andere Aspekte hinzufügen. Beispielhaft seien hier das für das ordnungssortierte logische Programmieren LOPSTER [Wei89] und für das logisch–funktionale Programmieren KANN [Sch91a] genannt. Es ist also von außerordentlicher Bedeutung, die WAM so um Mechanismen zum intuitionistischen logischen Programmieren zu erweitern, daß eine Integration anderer Ansätze möglich bleibt. Dieses Kriterium wollen wir als *Orthogonalität* bezeichnen.

Zu entwerfen ist eine Maschine, deren Instruktionssatz als Zielsprache für die Übersetzung intuitionistischer logischer Programme geeignet ist. Geeignet meint natürlich nicht nur, daß es zu jedem intuitionistischen logischen Programm eine Folge von Instruktionen gibt, die die gleiche Bedeutung haben, sondern daß es möglich ist, eine *effiziente* Folge von Instruktionen zu erzeugen. Bei dem Entwurf sind Kriterien, die diesen Aspekt betreffen, entsprechend zu berücksichtigen. Im Einzelnen werden bei dem Entwurf folgende Kriterien berücksichtigt:

Spezialisierung Treten spezielle Situationen häufig auf, so ist es wünschenswert, Instruktionen oder Datenstrukturen dieser Spezialisierung anzupassen, wenn dadurch Zeit oder Speicher eingespart werden kann, auch wenn allgemeinere Konzepte die nötige Funktionalität erbringen können.

Übersichtlichkeit Aus Gründen der Wartbarkeit und Verstehbarkeit der Maschine sollen alle in die Maschine eingehenden Konzepte und Mechanismen sich in dem Entwurf von Instruktionen und Datenstrukturen explizit widerspiegeln.

Verzögerung Operationen sollen erst dann durchgeführt werden, wenn es unbedingt nötig ist. Dies ist sehr wichtig, da wegen der Möglichkeit des Fehlschlagens von Klauseln nicht davon ausgegangen werden kann, daß wirklich alle Instruktionen einer Klausel von Anfang bis Ende ausgeführt werden.

3.3 Unifikation

Wie in Kapitel 2 schon deutlich wurde, ist zur Umsetzung eine Modifikation von Termen und der auf ihr operierenden Algorithmen — speziell des Unifikationsalgorithmus — nötig. Die Datenrepräsentation von Termen in der I–WAM ist das Thema, dem wir uns zuerst widmen.

$$\text{Tag} = (\text{Free}, \text{All}, \text{Ref}, \text{Const}, \text{Struct}, \text{List}, \text{Functor});$$

Abbildung 3.1: Definition der Objekttypen

3.3.1 Repräsentation von Termen

In der WAM gibt es als Objekte, die Terme repräsentieren, freie Variablen, Referenzen und Funktoren. Als Spezialisierung gibt es noch zusätzlich Konstanten und Listen.

Wir benötigen jetzt aber auch Konstanten, die bei der Abarbeitung eines Allquantors entstehen. Diese Konstanten müssen sowohl ein Datum besitzen, das es gestattet, die Konstante eindeutig zu identifizieren, als auch eines, das den Schachtelungsindex dieser Konstanten enthält. Prinzipiell bestehen zwei Möglichkeiten: Wir bilden diese Konstanten auf die schon vorhandenen Konstanten der WAM ab, oder führen einen gänzlich neuen Typ von Konstanten ein.

Sinnvoll ist das Verfolgen der zweiten Alternative: Bei der Unifikation müssen normale Konstanten, das heißt also Konstanten mit dem Schachtelungsindex null, nicht besonders berücksichtigt werden. Denn diese können keinesfalls bei der Berechnung von Unifikatoren zur Verletzung der Bereichskorrektheit führen. Deshalb soll die Datenrepräsentation so ausgelegt sein, daß Konstanten, die bei der Abarbeitung eines Allquantors erzeugt werden, eine eigenen Objekttyp darstellen. Diese Überlegung berücksichtigt das Kriterium *Spezialisierung*.

Dieser Objekttyp, der eine intern erzeugte Konstante darstellt, muß, wie oben schon erwähnt, auch ein Datum enthalten, welches eine eindeutige Identifizierung der erzeugten Konstante ermöglicht. Wir werden das durch eine ganze Zahl bewerkstelligen, die bei dem Erzeugen dieser Konstante eindeutig vergeben wird. Um im weiteren die Terminologie zu vereinfachen, wird eine Konstante, die aus einer allquantifizierten Variable hervorgeht, einfach wieder als allquantifizierte Variable bezeichnet.

Zusätzlich muß auch eine freie Variable, das heißt eine existentiell quantifizierte Variable, einen Schachtelungsindex besitzen. In der WAM ist die Trennung von freien Variablen und Referenzen nur implizit vorhanden: Beide enthalten als Datum eine Adresse. Zeigt diese Adresse auf das Objekt selber, so ist das Objekt eine freie Variable, sonst eine Referenz auf ein anderes Objekt. Da aber Referenzen nur zur Verwaltung von Bindungen freier Variablen untereinander dienen, sie also keinen Schachtelungsindex enthalten müssen, ist es vorteilhaft, einen eigenen Objekttyp für Referenzen bereitzustellen. Dennoch bleibt auch bei einer freien Variablen die Adressinformation als ein Datum erhalten: Sie dient einerseits zur eindeutigen Identifikation dieser Variable, andererseits gibt sie die Adresse an, wo diese Variable erzeugt wurde. Diese Information ist für viele Operationen unerläßlich, wie sich später zeigen wird.

Die oben geschilderten Anforderungen an die Datenrepräsentation von Termen führt zu der in Abbildung 3.1 dargestellten Definition von Objekttypen.

Die eigentliche Definition des Datentypes ist ein variabler Verbund. Die Bedeutung der einzelnen Komponenten ist im wesentlichen gleich geblieben, oder schon weiter oben erläutert worden. Die interne Darstellung von Konstanten und Funktoren ist hier nicht von Interesse. Damit ergibt sich die Definition, wie sie in Abbildung 3.2 zu sehen ist.

```
Obj = record
        case Tag : Tag of
          Free:                Addr : ↑Obj;
                               Nest : integer;
          All:                 IdNr : integer;
                               Nest : integer;
          Ref, Struct, List:   Addr : ↑Obj;
          Const:               Val  : ConstType;
          Functor:             Val  : FunctorType;
        esac;
      end;
```

Abbildung 3.2: Definition der Objekte

3.3.2 Unifikationsalgorithmus

Nach der Vorstellung des Datentypes für Terme, wenden wir uns dem Entwurf des Unifikationsalgorithmus zu. Um den Algorithmus 2.1 auf Seite 13 in eine konkrete Implementierung zu integrieren, müssen Vorkehrungen getroffen werden, die das Wiederherstellen von Variablen bei dem Rücksetzen der Maschine ermöglichen.

In der WAM wird das dadurch erreicht, daß die Adresse einer Variablen, die an einen Term gebunden wird, vor dem Überschreiben der Variablenzelle auf den Spurstapel gesichert wird. Dieses Vorgehen reicht jetzt nicht mehr aus. Denn neben der Adresse enthält eine freie Variable auch einen Schachtelungsindex, der mitgesichert werden muß. Also muß die Operation *Trail*(*Var*) das komplette Variablenobjekt *Var* auf dem Spurstapel ablegen.

Wie in der abschließenden Diskussion des Algorithmus 2.1 auf Seite 13 zur Berechnung erweiterter bereichskorrekter Unifikatoren schon angedeutet, wird die Anpassung von Schachtelungsindices nicht durch das Erzeugen neuer Variablen mit dem entsprechenden Schachtelungsindex vollzogen, sondern durch direktes Abändern des Schachtelungsindexes. Auch dieses Abändern macht das Sichern der Variablen mittels der Operation *Trail* auf dem Spurstapel nötig.

Die Implementierung des Algorithmus zerfällt in zwei Teile: Ein Teil dient der eigentlichen Unifikation, der andere der Untersuchung eines Termes, an den eine freie Variable gebunden werden soll.

Der erste Teil wird durch die Operation *Unify*(*TermA*, *TermB* : Obj) realisiert. Wie üblich werden zuerst die Terme dereferenziert, dann wird eine Fallunterscheidung je nach Objekttyp der beiden Terme gemacht, wenn die beiden Terme nicht ohnehin identisch sind.

Zwei typische Passagen dieser Operation sind in Abbildung 3.3 skizziert. Im ersten Teil wird eine freie Variable an eine andere freie Variable gebunden. Wie üblich wird die Variable, die an der höheren Adresse im Speicher liegt, durch eine Referenz auf die andere Variable überschrieben, nachdem der Inhalt der Variablen auf den Spurstapel gesichert wurde. Um die erweiterte Bereichskorrektheit im weiteren Verlauf der Unifikation zuzusichern, wird der Schachtelungsindex der frei bleibenden Variable nötigenfalls an den Schachtelungsindex der überschriebenen Variable angepaßt, jedoch nicht ohne zuvor die Variable zu sichern.

```
...;
⟨Free, Free⟩:
  if TermA.Addr > TermB.Addr then
    Trail(TermA);
    TermA.Addr↑ ← ⟨Ref, TermB.Addr⟩;
    if TermA.Nest < TermB.Nest then
      Trail(TermB);
      TermB.Addr↑.Nest ← TermA.Nest;
    fi;
  else
    ...;
  fi;
...;
⟨Free, All⟩:
  if TermA.Nest < TermB.Nest then
    DoFail();
  else
    Bind(TermA, TermB);
  fi;
...;
```

Abbildung 3.3: Teile des Unifikationsalgorithmus

Die zweite Passage betrifft den Fall, wenn eine freie Variable mit einer allquantifizierten Variablen unifiziert wird. Verletzt die vorzunehmende Bindung die Bereichskorrektheit, so wird mittels der Operation *DoFail*() ein Rücksetzen der Maschine ausgelöst. Andernfalls wird die freie Variable an die allquantifizierte Variable mittels der Operation *Bind* gebunden, wobei diese Operation auch die Sicherung der zu überschreibenden Variable vornimmt.

In beiden Passagen tritt jetzt auch klar hervor, wieso eine freie Variable die Adresse, an der sie erzeugt wurde, als Datum enthält: Dadurch kann das Überschreiben der Variablen direkt erfolgen.

Der zweite Teil der Implementierung des Algorithmus beinhaltet neben dem eigentlichen Occur–Check auch die Überprüfung auf die Bereichskorrektheit und die Anpassung von Schachtelungsindices für nicht atomare Terme. Die Realisierung dieser Operation verläuft genauso wie für die oben schon geschilderten Fälle der Unifikation.

Vollständig sind die Operationen, die beide Teile der Unifikation realisieren, in dem Abschnitt A.5 beschrieben. Die hier nicht beschriebenen Teile betrachten wir in einem Beispiel.

Beispiel 3.1 (Unifikation in der I–WAM) Auf Ebene der I–WAM wollen wir die Terme $f(x^2, [g(c^1, y^3)])$ und $f(z^1, z^1)$ miteinander unifizieren. Die Repräsentation der Terme ist in Abbildung 3.4 zu sehen. Die beiden Register `X[1]` und `X[2]` enthalten die Termwurzeln, die Argumente sind auf der Halde dargestellt. Die Notation $\uparrow i$ soll ein Wert vom Typ ↑Obj bezeichnen.

Ganz links in der Abbildung ist die Ausgangssituation. Bei der Unifikation werden zunächst

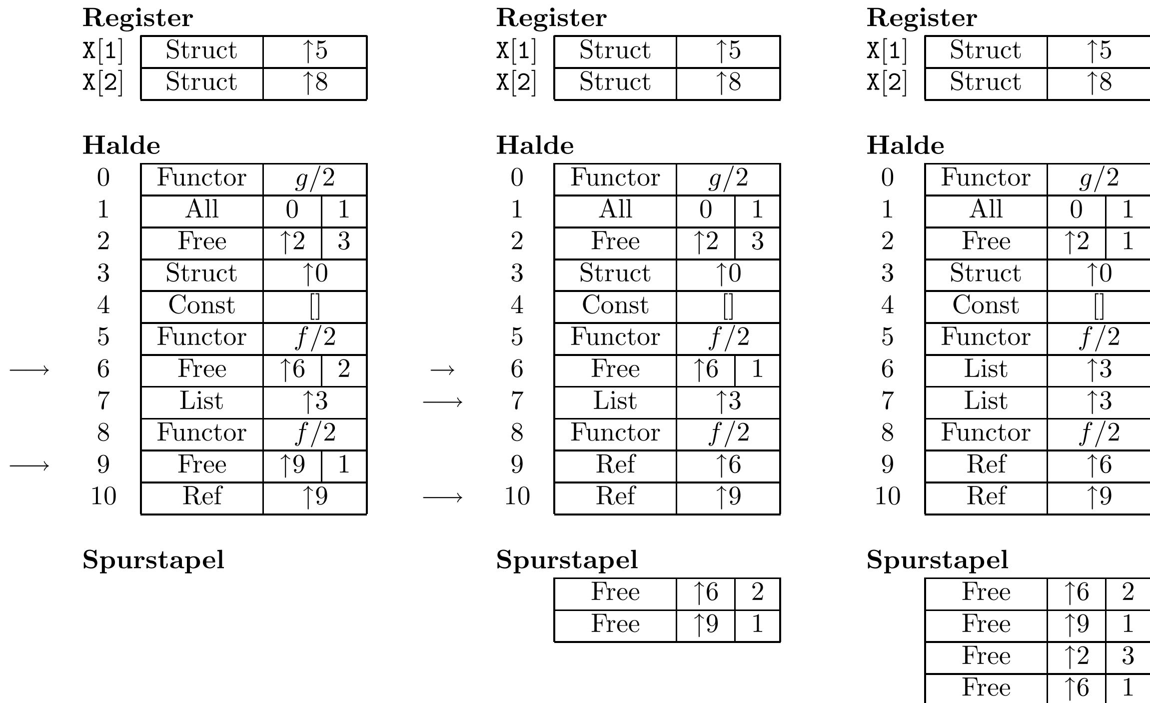


Abbildung 3.4: Unifikation von $f(x^2, [g(c^1, y^3)])$ und $f(z^1, z^1)$

die Funktoren in den Zellen 5 und 8 überprüft, auf die die Termwurzeln zeigen. Da beide gleich sind, werden jeweils die ersten Argumente miteinander unifiziert. Diese sind durch die langen Pfeile ⟶ gekennzeichnet. Nun wird die Variable in Zelle 9 an die Variable in Zelle 6 gebunden, vorher wird jedoch der Schachtelungsindex der Variablen in Zelle 6 angepaßt.

Danach haben wir die Situation, wie in der Mitte skizziert. Jetzt werden die beiden Zellen 7 und 10 miteinander unifiziert. Durch Dereferenzieren wird der Wert von der Zelle 10 ermittelt: Dies ist die freie Variable in Zelle 6. Nun wird die allquantifizierte Variable in Zelle 1 überprüft, und der Schachtelungsindex der freien Variable in Zelle 2 angepaßt. Anschließend wird jetzt die Variable in Zelle 6 durch den Listenkopf überschrieben. □

Man beachte, daß in dem vorangehenden Beispiel ein Variablenobjekt mehrmals auf dem Spurstapel abgelegt wurde. In Abschnitt 5.1 wird dieses Phänomen noch näher untersucht werden.

3.4 Explizite Quantifizierung

Die Umsetzung von expliziter Quantifizierung in Instruktionen der I–WAM ist der Inhalt dieses Abschnittes. Wie weiter oben schon erwähnt, besitzt diese Umsetzung zwei Aspekte: Einen, der die Bereitstellung von Argumenten, und einen, der die Übernahme von Argumenten beinhaltet. Ein formaler Ansatz für den Entwurf von Instruktionen, die die Unifikation realisieren, läßt sich in [Kur86] finden. Anstatt der dort beschriebenen formalen Methode der partiellen Evaluation zu folgen, werden wir hier die Spezialisierungen argumentativ herausarbeiten.

Tabelle 3.1: Instruktionen der WAM zur Bereitstellung von Argumenten

Art	Atomar		Verschachtelt[1]	
Freie Variable	`put_variable`	`V`[n]`, A`[i]	`set_variable`	`V`[n]
	`put_void`	`A`[i]	`set_void`	`V`[n]
Initialisierte Variable	`put_value`	`V`[n]`, A`[i]	`set_value`	`V`[n]
	`put_unsafe_value`	`V`[n]`, A`[i]	`set_local_value`	`V`[n]
Struktur	`put_structure`	`F, A`[i]	`put_structure`	`F, X`[n]
Liste	`put_list`	`A`[i]	`put_list`	`X`[n]
Konstante	`put_constant`	`C, A`[i]	`set_constant`	`C`

3.4.1 Bereitstellung von Argumenten

Vor der Beschäftigung mit der Bereitstellung von Argumenten in der I–WAM, betrachten wir noch einmal die Instruktionen der WAM, die Argumente bereitstellen. Schematisch sind diese in Tabelle 3.1 skizziert. Je nach Objekttyp des bereitzustellenden und nach Art der Bereitstellung in einem Argumentregister für Termwurzeln und atomare Terme oder auf der Halde für verschachtelte Terme werden diese Befehle verwendet.

Unter den Aspekt der Bereitstellung von Argumenten fällt auch das direkte Umsetzen von explizit in den Rümpfen von Klauseln auftretenden Quantoren in Instruktionen der I–WAM. Bei der Formulierung der Ableitbarkeit hatten wir als ein entscheidendes Bestimmungsstück den Schachtelungsindex. Jedesmal, wenn ein Allquantor abzuleiten ist, wird dieser Schachtelungsindex um eins erhöht und eine neue Konstante erzeugt. Das werden wir auch in der I–WAM so handhaben: Wir führen ein neues Register NL[2] ein, das den aktuellen Schachtelungsindex enthält. Das Register muß bei einem Rücksetzvorgang der Maschine wiederhergestellt werden können, also muß der Wert dieses Registers bei Anlegen eines Wahlpunktes in diesem abgelegt werden.

Betrachten wir nun den Klauselrumpf $\forall x\, G$, so muß der zugehörige Code folgendes leisten:

1. Der Schachtelungsindex muß um eins erhöht werden. Diese Funktionalität besitzt die neue Instruktion `quantify`.

2. Wenn die Variable x das erste Mal in G benutzt wird, so muß eine allquantifizierte Variable mit dem gerade inkrementierten Schachtelungsindex erzeugt werden.

3. Anschließend muß der Schachtelungsindex wieder um eins erniedrigt werden: Dieses erledigt dann die neue Instruktion `dequantify`.

Das Anpassen des aktuellen Schachtelungsindexes ist jetzt klar, was noch fehlt ist, wie die allquantifizierte Variable erzeugt wird. Wird eine existentiell quantifizierte Variable bei der Parameterübergabe das erste Mal benutzt, so wird sie durch die Instruktion `put_variable V`[n]`,`

[1] Wir verwenden hier nicht `unify`–Instruktionen wie in [War83], sondern `set`–Instruktionen wie in [AK90]. Dies liegt daran, daß die Operationen zur Bereitstellung von Termen in Rümpfen nichts mit Unifikation zu tun haben. Somit besteht eine deutliche Differenz zwischen der Semantik einer `set`–Instruktion und der einer `unify`–Instruktion im Schreibmodus.

[2] NL für Schachtelungsindex, englisch nesting level

```
quantify                                      ∀x = X[4] (
                                                ∃y = X[5] (
quantify                                          ∀z = X[6] (
put_variable_all      X[4], A[1]                    p(x,
put_variable          X[5], A[2]                       y,
put_variable_all      X[6], A[3]                        z
call                  ···, p/3                            )
dequantify                                          )
                                                  )
dequantify                                      )
```

Abbildung 3.5: Fehlerhafte Codesequenz für Beispiel 3.2

A[i] in dem Speicherplatz V[n] erzeugt, und eine Referenz auf diese Variable in das Argumentregister A[i] geschrieben. Analog dazu führen wir jetzt eine neue Instruktion `put_variable_all V[`n`], A[`i`]` ein, die eine neue allquantifizierte Variable für die Parameterübergabe erzeugt. Als Schachtelungsindex dieser Variablen wird jetzt der Wert des Registers *NL* und als Identifikationsnummer eine bisher noch nicht verwendete ganze Zahl eingetragen. Im Gegensatz dazu, daß bei der `put_variable`–Instruktion nur eine Referenz auf die erzeugte Variable in das Argumentregister geschrieben wird, kann hier die erzeugte allquantifizierte Variable direkt in das Argumentregister geschrieben werden. Dies liegt daran, daß an allquantifizierte Variablen keine Terme gebunden werden können. Es ist also nicht nötig, für eine Variable nur einen Speicherplatz zu belegen.

Es ist jedoch nicht nur bei den allquantifizierten Variablen nötig, beim Erzeugen derselben den Schachtelungsindex abzuspeichern. Auch bei existentiell quantifizierten Variablen muß beim Erzeugen der Schachtelungsindex gespeichert werden. Dieses Ablegen des Schachtelungsindexes ist gerade ein Teil der *new*–Operation aus Definition 2.11 auf Seite 16 für implizit oder ein Teil der Ableitungsregel $\vdash^{\exists}$ für explizit existentiell quantifizierte Variablen.

Die Übersetzung explizit auftretender Existenzquantoren ist jetzt trivial: Wird die quantifizierte Variable das erste Mal verwendet, so bekommt sie automatisch den gerade aktuellen Wert von *NL* als Schachtelungsindex.

Treten mehrere Allquantoren hintereinander auf, so ist es nicht nötig, für jeden der Quantoren eine `quantify`– und die entsprechende `dequantify`–Instruktion zu erzeugen. Die erweiterte Bereichskorrektheit bleibt auch dann gewahrt, wenn einer Folge von Allquantoren derselbe Schachtelungsindex zugeordnet wird, wie man sich leicht klarmacht. Dadurch, daß jede neu erzeugte allquantifizierte Variable sowieso eine eindeutige Identifikationsnummer erhält, ist auch die Unterscheidbarkeit der Variablen gesichert.

Doch die bisherigen Instruktionen reichen noch nicht aus, alle Fälle bei der Quantifizierung korrekt zu behandeln. Betrachten wir dazu das folgende Beispiel.

Beispiel 3.2 Wir haben das Stück eines Rumpfes:

$$\forall x \exists y \forall z\, p(x, y, z).$$

```
quantify                                         ∀x = X[4] (
                                                   ∃y = X[5] (
quantify                                             ∀z = X[6] (
put_variable_all_offset   X[4], A[1], 1                p(x,
put_variable_offset       X[5], A[2], 1                   y,
put_variable_all          X[6], A[3]                       z
call                      ..., p/3                          )
dequantify                                               )
                                                       )
dequantify                                         )
```

Abbildung 3.6: Korrekte Codesequenz für Beispiel 3.2

Übersetzen wir dieses nach unserem Schema, erhalten wir die Codesequenz aus Abbildung 3.5. Das bedeutet aber, daß die allquantifizierte Variable für x und die existentiell quantifizierte Variable für y mit einem Schachtelungsindex erzeugt werden, der um eins zu hoch ist. □

In diesem speziellen Fall können wir das Dilemma dadurch beheben, daß wir die Reihenfolge der Instruktionen vertauschen. Generell geht dies jedoch nicht, wie wir später noch sehen werden. Eine einfache Lösung ist, daß wir die I–WAM um die zusätzlichen Instruktionen `put_variable_offset V`[n]`, A`[i]`, O` und `put_variable_all_offset V`[n]`, A`[i]`, O` erweitern. Wie die Namen schon andeuten, wird bei dem Erzeugen einer Variablen jetzt der Wert von *NL*, vermindert um den Offset `O` als Schachtelungsindex eingetragen. Die korrekte Codesequenz für Beispiel 3.2 ist jetzt in Abbildung 3.6 dargestellt.

Nun kann auch der Fall eintreten, daß eine Variable für die Parameterübergabe erzeugt wird, jedoch später nicht mehr benutzt wird. Bei der WAM gibt es dafür die durch Spezialisierung aus der `put_variable`–Instruktion hervorgegangene Instruktion `put_void A`[i], die eine Variable auf der Halde erzeugt und eine Referenz auf diese in das Argumentregister `A`[i] schreibt. Diese Instruktion muß jetzt dahingehend erweitert werden, daß beim Erzeugen auch der Schachtelungsindex eingetragen wird. Aus den oben diskutierten Gründen führen wir jetzt auch die Instruktion `put_void_offset A`[i]`, O` ein, deren Semantik klar ist. Analoge Instruktionen führen wir auch für allquantifizierte Variablen ein: `put_void_all A`[i] und `put_void_all_offset A`[i]`, O` erzeugen eine allquantifizierte Variable direkt in dem Argumentregister `A`[i] mit dem jeweils entsprechenden Wert für den Schachtelungsindex, da allquantifizierte Variablen im Gegensatz zu existentiell quantifizierten Variablen frei kopierbar sind.

Die übrigen Instruktionen, die Strukturen, Listen, Konstanten oder bereits früher initialisierte Variablen in einem Argumentregister ablegen, bedürfen natürlich keiner Änderung, da sie eine reine Kopierfunktion besitzen.

Nun müssen wir noch untersuchen, wie zusammengesetzte Terme als Argumente übergeben werden. Die Termwurzel eines Argumentes wird in das Argumentregister geschrieben, dann folgen Instruktionen, die auf der Halde die Argumente des zu konstruierenden Terms ablegen — siehe hierzu auch die Tabelle 3.1. Sind diese Argumente selber wieder zusammengesetzte Terme, so wiederholt sich das Verfahren. Zu betrachten sind wiederum die Befehle, die mit

```
quantify                                                ∀x = X[3] (
                                                          ∃y = X[4] (
quantify                                                    ∀z = X[5] (
                                                            ∀u = X[6] (
put_structure               f/3, A[1]                         p(f(
set_variable_all            X[5]                                  z,
set_variable_all_offset     X[3], 1                                 x,
set_variable_all            X[6]                                      u),
put_variable_offset         X[4], A[2], 1                               y
call                        ···, p/2                                      )
dequantify                                                     )
                                                             )
dequantify                                                 )
```

Abbildung 3.7: Code für Beispiel 3.3

dem Erzeugen von Variablen zu tun haben. In einer normalen WAM sind dies die Befehle `set_variable` V[n] und `set_void`.

Der gleichen Argumentation wie oben folgend, müssen wir jetzt diese Befehle so erweitern, daß die Schachtelungsindices beim Erzeugen mit abgelegt werden. Der bei der WAM für freie Variablen benutzte Befehl `set_variable` V[n] wird dann also dahingehend erweitert, daß der Schachtelungsindex mit abgelegt wird. Aber auch allquantifizierte Variablen müssen nun erzeugt und dann auf der Halde abgelegt werden können. Also müssen wir eine Instruktion einführen, die dieses leistet: Das Erzeugen und Ablegen einer allquantifizierten Variablen, die in einem Term das erste Mal auftritt, übernimmt die Instruktion `set_variable_all` V[n]. Da auch hier die Möglichkeit besteht, daß zu der Erzeugung von Variablen ein anderer Wert für den Schachtelungsindex als der des Registers *NL* herangezogen werden muß, führen wird nun auch die Befehle `set_variable_offset` V[n], `O` und `set_variable_all_offset` V[n], `O` ein.

Beispiel 3.3 Betrachten wir nun einmal den folgenden Rumpf einer Klausel:

$$\forall x \exists y \forall z \forall u \, p(f(z, x, u), y).$$

Die Übersetzung ist in Abbildung 3.7 zu sehen. Dies ist aber eine Situation, in der durch Verschieben der `quantify`–Instruktionen nicht erreicht werden kann, daß jede Variable mit dem richtigen Schachtelungsindex erzeugt wird.

Dies liegt daran, daß hier die Instruktionen zur Ablage von Argumenten von Termen im Gegensatz zu den Instruktionen zur Ablage von Argumenten von Prädikaten in ihrer Reihenfolge nicht beliebig verändert werden können. Genau das ist der weiter oben erwähnte Grund, wieso die `offset`–Befehle zwingend notwendig sind. □

Analog zu der Ablage von nur einmal verwendeten Variablen in einem Argumentregister durch die `put_void`–Instruktionen sehen wir auch Instruktionen vor, die einmal verwendete Variablen auf der Halde erzeugen und ablegen. Die schon vorher vorhandene Instruktion

Tabelle 3.2: Instruktionen der I–WAM zur Bereitstellung von Argumenten

Art	Atomar		Verschachtelt	
Freie Variable	`put_variable`	V[n], A[i]	`set_variable`	V[n]
	`put_variable_offset`	V[n], A[i], O	`set_variable_offset`	V[n], O
	`put_void`	A[i]	`set_void`	V[n]
	`put_void_offset`	A[i], O	`set_void_offset`	O
Universelle Variable	`put_variable_all`	V[n], A[i]	`set_variable_all`	V[n]
	`put_variable_all_offset`	V[n], A[i], O	`set_variable_all_offset`	V[n], O
	`put_void_all`	A[i]	`set_void_all`	V[n]
	`put_void_all_offset`	A[i], O	`set_void_all_offset`	O
Initialisierte Variable	`put_value`	V[n], A[i]	`set_value`	V[n]
	`put_unsafe_value`	V[n], A[i]	`set_local_value`	V[n]
Struktur	`put_structure`	F, A[i]	`put_structure`	F, X[n]
Liste	`put_list`	A[i]	`put_list`	X[n]
Konstante	`put_constant`	C, A[i]	`set_constant`	C

`set_void` wird dahingehend erweitert, daß der Schachtelungsindex beim Erzeugen abgelegt wird. Für die anderen möglichen Fälle führen wir nun noch die Instruktionen `set_void_offset O`, `set_void_all` und `set_void_all_offset O` ein.

Befehle, die bereits initialisierte Variablen oder eine Konstante auf der Halde ablegen, bedürfen auf Grund ihrer reinen Kopierfunktion wieder keiner Änderung.

Zur besseren Übersicht sind in Tabelle 3.2 noch einmal alle Instruktionen zusammengefaßt, die zur Erzeugung von Termen in den Argumentregistern und auf der Halde benötigt werden. Die genaue Definition der `put`–Instruktionen ist in Abschnitt A.8, die der `set`–Instruktionen in Abschnitt A.9 zu finden.

3.4.2 Unifikation von Köpfen

Nachdem wir wissen, wie Terme in Rümpfen von Klauseln konstruiert werden, ist zu überlegen, wie die Unifikation von Klauselköpfen vor sich geht. Es gibt keine neuen Instruktionen, da in dem Kopf einer Klausel nur existentiell quantifizierte Variablen auftreten. Die Aufgabe ist, die Spezialinstruktionen, die die Unifikation vornehmen, so zu erweitern, daß sie die erweiterte Bereichskorrektheit gewährleisten. In Tabelle 3.3 sind zur Übersicht alle Instruktionen zusammengefaßt, die zur Unifikation von Köpfen dienen.

Betrachten wir zuerst die `get`–Instruktionen, die die Termwurzeln aus den Argumentregistern übernehmen. Mit der Instruktion `get_variable` V[n], A[i] wird der Term aus dem Argumentregister A[i] in die Variablenzelle V[n] kopiert. Das heißt, eine Variable mit dem aktuellen Schachtelungsindex wird erzeugt und sofort an den Term aus A[i] gebunden. Da aber nach Lemma 2.4 auf Seite 20 der aktuelle Schachtelungsindex nicht kleiner als der Schachtelungsindex einer existentiellen oder universellen Variable in dem Term aus A[i] sein kann, kann der Term aus A[i] ohne weitere Untersuchung übernommen werden. Da alle Variablen einer gerade bearbeiteten Klausel nicht gesichert werden müssen, braucht keine Variable angelegt zu werden: Der Term aus A[i] wird direkt in die Variablenzelle V[n] kopiert. Somit haben wir die gleiche Semantik dieser Instruktion wie in der normalen WAM.

Tabelle 3.3: Instruktionen zur Unifikation von Köpfen

Art	Atomar		Verschachtelt	
Freie Variable	get_variable	V[n], A[i]	unify_variable	V[n]
Initialisierte Variable	get_value	V[n], A[i]	unify_value	V[n]
	—		unify_local_value	V[n]
Struktur	get_structure	F, A[i]	get_structure	F, X[n]
Liste	get_list	A[i]	get_list	X[n]
Konstante	get_constant	C, A[i]	unify_constant	C

Tritt eine durch `get_variable V`[n], `A`[i] erzeugte Variable `V`[n] nochmals in einem Argument nicht verschachtelt auf, so bewirkt die Instruktion `get_value V`[n], `A`[i], daß die Terme aus `V`[n] und `A`[i] miteinander unifiziert werden. Also kann auch diese Instruktion unverändert übernommen werden, da der Unifikationsalgorithmus alle zusätzlich nötigen Operationen vornimmt. Die Unifikation eines Argumentterms mit einer Konstanten durch die Instruktion `get_constant C, A`[i] bleibt natürlich auch unverändert.

Treten nun im Kopf nicht atomare Terme auf, das heißt also eine Struktur oder eine Liste, so ist die Situation etwas komplizierter. Betrachten wir die Instruktion `get_structure F, A`[i]. Je nach Typ des Terms in `A`[i] verhält sich diese Instruktion. Sei also dieser Typ auch eine Struktur. Zeigen beide Strukturobjekte auf unterschiedliche Funktoren, so wird Fehlschlag ausgelöst. Sind die beiden Funktoren gleich, so geht die Maschine in den Lesemodus, das heißt Terme werden nicht konstruiert, sondern die Argumente der Terme werden miteinander unifiziert. Kommt in dem gerade betrachteten Kopfterm dann eine Variable vor, so wird die Instruktion `unify_variable V`[n] für das erste Auftreten, für alle weiteren dann die Instruktionen `unify_value V`[n] oder `unify_local_value V`[n] eingesetzt. Die Funktionalität dieser drei Instruktionen für den Lesemodus der Maschine kann unverändert bleiben: Die Situation ist die gleiche wie für die `get_variable`– und `get_value`–Instruktionen, wie oben schon geschildert. Trivialerweise bleibt natürlich auch die Semantik der Instruktion `unify_constant C` im Lesemodus der Maschine gleich.

Ist aber nun der Term in `A`[i] eine freie Variable, so haben wir eine gänzlich andere Situation. Die Maschine geht bei der Ausführung der `get_structure`–Instruktion in den Schreibmodus über und legt bei der Ausführung einer folgenden `unify_variable`–Instruktion eine existentiell quantifizierte Variable ab, die nun auch mit dem Schachtelungsindex erzeugt werden muß, die die als Argument übergebene Variable besitzt.

Das heißt, wenn wir jetzt Terme bei der Unifikation von Köpfen konstruieren, die freie Variablen enthalten, so müssen wir die Variablen mit dem Schachtelungsindex der Variablen erzeugen, die als Argument übergeben wurde! Dies ist genau die Situation, die auch bei der Unifikation eintritt, wenn eine existentiell quantifizierte Variable an einen Term gebunden wird: Die eventuelle Anpassung von Schachtelungsindices von Variablen in dem Term nimmt dann der Occur–Check vor. Damit muß die `get_structure` Instruktion beim Schalten in den Schreibmodus zusätzlich ein Register *WNL*[3] mit dem Schachtelungsindex der Variablen aus dem Argumentregister initialisieren. Wird dann die Instruktion `unify_variable V`[n] ausgeführt, wenn die Maschine

[3] *WNL* für Schreibschachtelungsindex, englisch write nesting level

Code für den Rumpf $\exists x \forall y\, p(x,y)$:

```
                                                ∃x = X[3](
quantify                                          ∀y = X[4](
put_variable_offset     X[3], A[1], 1                p(x,
put_variable_all        X[4], A[2]                      y
call                    ···, p/2                          )
dequantify                                            )
                                                )
```

Code für die Klausel $p(f(z),z)$:

```
get_structure           f/1, A[1]        p(f(
unify_variable          X[3]                  z),
get_value               X[3], A[2]              z
proceed                                          )
```

Abbildung 3.8: Code für Beispiel 3.4

sich im Schreibmodus befindet, wird diese Variable mit dem Wert des Registers *WNL* erzeugt.

Beispiel 3.4 In Abbildung 3.8 ist der Code für einen Rumpf und für eine Klausel zu sehen. In diesem Beispiel wird die atomare Klausel für p mit einer freien Variablen als erstem Argument aufgerufen, die einen Schachtelungsindex besitzt, der um eins kleiner als der aktuelle Schachtelungsindex im Register *NL* ist.

Von links nach rechts sind in Abbildung 3.9 Ausschnitte aus dem Maschinenzustand zu sehen, jeweils nach Abarbeitung der darüber stehenden Instruktion aus dem Code der atomaren Klausel für p. Nach Ansprung der Klausel für p haben wir die ganz links skizzierte Situation. In der Mitte ist das Ergebnis der Ausführung von `put_structure f/1, A[1]` zu sehen. Die freie Variable an der Haldenzelle 0 ist durch ein Strukturobjekt ersetzt, das auf die nächste Haldenposition zeigt, an dem der Funktor $f/1$ steht. Das Register *WNL* ist jetzt mit dem Wert 0 belegt. Ganz rechts haben wir an der Haldenspitze eine neue freie Variable, die den Schachtelungsindex $0 = WNL$ besitzt. Erzeugt wurde diese von der `unify_variable X[3]`–Instruktion.

Wird jetzt der Befehl `get_value X[3], A[2]` ausgeführt, so wird Rücksetzen der Maschine ausgelöst: Denn die Variable in der Haldenzelle 2 mit Schachtelungsindex 0 darf nicht an die allquantifizierte Variable im Register `A[2]` mit Schachtelungsindex 1 gebunden werden. □

Befindet sich die Maschine im Schreibmodus, so wird durch die Instruktion `unify_constant C` einfach die Konstante `C` an die gerade aktuelle Haldenposition geschrieben. Damit bleibt diese Instruktion ungeändert. Legen wir jedoch einen Term durch die Instruktion `unify_value V[n]` oder `unify_local_value V[n]` ab, so müssen die abzulegenden Terme überprüft werden: Freie Variablen müssen eventuell in ihrem Schachtelungsindex an den Wert von *WNL* angepaßt werden, der Schachtelungsindex allquantifizierter Variablen muß mit dem Wert von *WNL* verglichen werden und zusätzlich muß sichergestellt werden, daß keine zyklischen Terme auf der Halde erzeugt werden. Diese Überprüfung in den beiden Instruktionen wird durch die Ope-

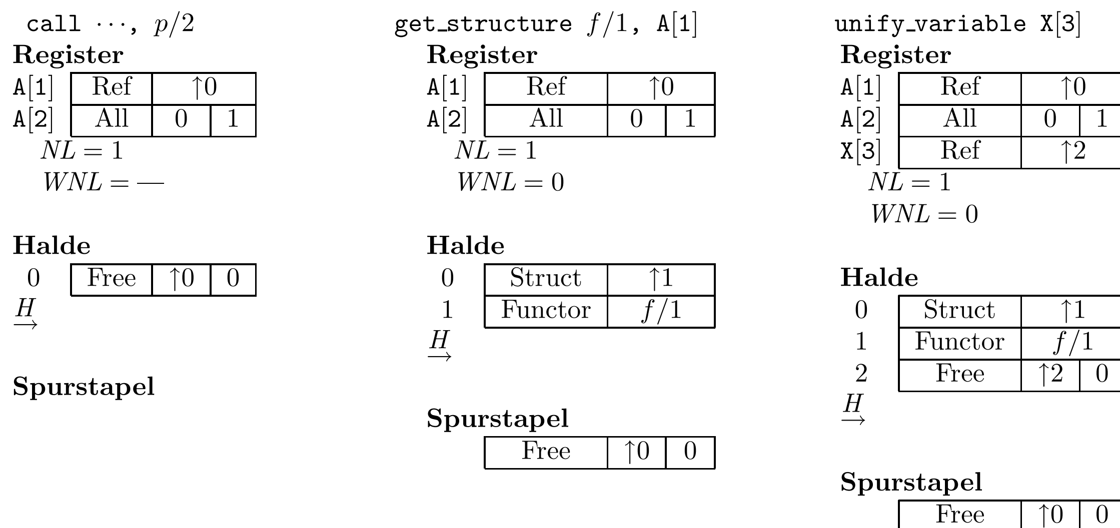


Abbildung 3.9: Ablaufverfolgung für Beispiel 3.4

ration *WriteOccurs* realisiert, die diese drei Aufgaben übernimmt. Genau aufgeführt ist diese Operation in Abschnitt A.5.

Die beiden noch zu untersuchenden Instruktionen `get_list A`[i] und `unify_void`, die Spezialisierungen der Instruktionen `get_structure` und `unify_variable` darstellen, sind natürlich genau so zu erweitern, wie wir es oben gerade für die allgemeineren Operationen durchgeführt haben.

Damit sind die Instruktionen, die zur Unifikation von Köpfen benötigt werden, komplett. Die zuletzt besprochenen `get`– und `unify`–Instruktionen sind in den Abschnitten A.6 und A.7 genau definiert.

3.5 Hypothetische Implikationen

Bei der Umsetzung von hypothetischen Implikationen in die I–WAM drängen sich sofort die folgenden Fragen auf:

- Wie wird bei der Abarbeitung einer hypothetischen Implikation dem Programm Code hinzugefügt, das heißt, eine *dynamische Klausel* aktiviert?

- Wie können die hinzugefügten dynamischen Klauseln benutzt werden?

- Wie entfernen wir nach Abarbeitung der hypothetischen Implikation die hinzugefügten dynamischen Klauseln?

- Wie können beim Rücksetzen der Maschine hinzugefügte Klauseln gelöscht, oder bereits entfernte Klauseln wieder verfügbar gemacht werden?

Der erste Schritt in der Beantwortung dieser Fragen ist die Feststellung, daß der Maschine bei der Aktivierung von Klauseln Information verfügbar gemacht werden muß, die die spätere

Benutzung der Klauseln zuläßt. Wie diese Information aussieht, und wie sie verwaltet wird, damit befaßt sich der nächste Teilabschnitt.

3.5.1 Implikationspunkte

Die Information über eine aktivierte dynamische Klausel muß in einer Form gespeichert werden, die es gestattet, daß wenn sie nach erfolgreicher Abarbeitung der hypothetischen Implikation, die die dynamische Klausel aktiviert hat, entfernt wird, es beim Rücksetzen der Maschine jedoch möglich ist, diese Information wieder zugänglich zu machen. Diese Anforderungen sind genau die, die auch an Prozedurumgebungen der WAM gestellt werden. Also ist der Ort, an dem wir die Information ablegen, der Kontrollstapel. Neben den Prozedurumgebungen und Wahlpunkten führen wir eine dritte Art von Daten auf dem Kontrollstapel ein: Die *Implikationspunkte*.

In Abschnitt 2.5 hatten wir überlegt, daß dynamisch aktivierte Klauseln in der umgekehrten Reihenfolge ihrer Aktivierung verwendet werden sollen: Das bedeutet bei der Benutzung der Information, daß der zuletzt bereitgestellte Implikationspunkt zuerst berücksichtigt wird. Also werden die Implikationspunkte — genauso wie die Prozedurumgebungen und Wahlpunkte — mit einer Verkettungsinformation versehen. Wir führen ein neues Register *IP*[4] ein, das auf den zuletzt hinzugefügten Implikationspunkt zeigt. Jeder Implikationspunkt enthält dann einen Zeiger auf seinen Vorgänger. Das neue Register *IP* wird auch bei Anlegen eines Wahlpunktes in diesem gespeichert.

Nutzen wir die Information aus den Implikationspunkten bei dem Prozeduraufruf für ein Prädikat p/n, so wollen wir nicht auf den letzten Implikationspunkt zugreifen, sondern vielmehr auf den letzten Implikationspunkt, der bei der Aktivierung einer dynamischen Klausel für dieses Prädikat p/n angelegt wurde. Zumindest muß also in einem Implikationspunkt gespeichert werden, für welches Prädikat dieser Implikationspunkt erzeugt wurde. Nun könnten wir den letzten Implikationspunkt für ein bestimmtes Prädikat durch lineares Suchen innerhalb der verketteten Liste der Implikationspunkte bestimmen. Dies ist aber viel zu ineffizient. Wünschenswert ist es, wenn wir für jedes Prädikat durch Direktzugriff den zuletzt angelegten Implikationspunkt für dieses Prädikat ermitteln können. Wir gehen jetzt davon aus, daß es zu jedem Prädikat p/n einen Prozedurbeschreibungsblock gibt, der verschiedene Informationen enthält[5]. Hier legen wir jetzt einen Zeiger auf den zuletzt für dieses Prädikat aktivierten Implikationspunkt ab. Als Information, zu welchem Prädikat dieser Implikationspunkt gehört, verwenden wir dann gerade den Zeiger auf den Prozedurbeschreibungsblock. Zusätzlich verketten wir die Implikationspunkte miteinander, die zum selben Prädikat „gehören". Damit können wir jetzt sequentiell auf alle Implikationspunkte eines Prädikates zugreifen, ohne Implikationspunkte betrachten zu müssen, die zu anderen Prädikaten gehören.

Wenn wir eine dynamische Klausel benutzen wollen, müssen wir natürlich wissen, an welcher Adresse im Speicher sich der Code für diese Klausel befindet: Diese Adresse muß auch in einem Implikationspunkt gespeichert werden. Zusätzlich wird in einem Implikationspunkt noch der Wert des Registers E, das einen Zeiger auf die gerade aktuelle Prozedurumgebung enthält, und eine Menge von Objekttypen, das heißt ein Wert vom Typ „**set of** Tag", gespeichert.

[4] *IP* für Implikationspunkt, englisch implication point

[5] So etwas wird es in jeder Implementierung geben, um Informationen zu den einzelnen Klauseln eines Prädikates zu verwalten.

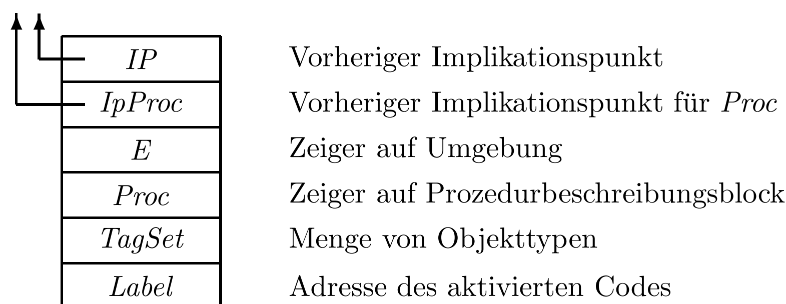


Abbildung 3.10: Ein Implikationspunkt auf dem Kontrollstapel

Wozu die beiden letzteren Daten gespeichert werden, wird später erklärt. Schematisch ist ein Implikationspunkt in Abbildung 3.10 gezeigt.

Bisher haben wir erst diskutiert, wie und wo ein Implikationspunkt erzeugt wird und mit welcher Information er versehen wird. Nun haben wir uns aber auch Gedanken darüber zu machen, wie ein Implikationspunkt gelöscht werden kann, und wie Implikationspunkte durch das Rücksetzen verwaltet werden. Löschen können wir einen Implikationspunkt einfach dadurch, daß wir ihn aus der Liste der Implikationspunkte ausketten. Bei Prozedurumgebungen oder Wahlpunkten reicht es aus, den Zeiger hierauf einfach auf den Vorgänger zurückzusetzen. Bei Implikationspunkten müssen wir sowohl das Register *IP*, als auch den Zeiger auf den letzten Implikationspunkt für dasselbe Prädikat im Prozedurbeschreibungsblock auf die im Implikationspunkt gespeicherten Werte zurücksetzen. Den Zeiger auf den Prozedurbeschreibungsblock entnehmen wir dabei auch dem Implikationspunkt. Wichtig ist, daß der Implikationspunkt aus der Liste ausgegliedert ist, jedoch weiterhin vor dem Überschreiben geschützt ist. Benötigt wird dieser Implikationspunkt beim Rücksetzen der Maschine, wenn ein Wahlpunkt auf dem Kontrollstapel nach diesem Implikationspunkt existiert. Aber durch den Wahlpunkt ist der Implikationspunkt vor Überschreiben geschützt, da zum Neuanlegen von Daten auf dem Kontrollstapel erst Zellen herangezogen werden, die hinter diesem Wahlpunkt liegen.

Beim Rücksetzen der Maschine sind zwei Fälle zu unterscheiden. Einmal kann es nötig sein, Implikationspunkte zu löschen, andererseits kann es auch sein, daß Implikationspunkte wieder sichtbar gemacht werden müssen. Das Löschen von Implikationspunkten ist genauso zu bewerkstelligen wie oben diskutiert: Solange der Implikationspunktzeiger *IP* noch nicht den im Wahlpunkt gespeicherten Wert von *IP* erreicht hat, werden Implikationspunkte durch das oben beschriebene Umsetzen von *IP* und des Zeigers im Prozedurbeschreibungsblock ausgegliedert. Um Implikationspunkte wieder aufzubauen, gehen wir wie folgt vor: Von dem Implikationspunkt ausgehend, auf den der Wert von *IP* im Wahlpunkt zeigt, durchlaufen wir die Liste der Implikationspunkte, bis wir auf den Implikationspunkt treffen, der durch *IP* referenziert wird. Für jeden betrachteten Implikationspunkt stellen wir den Prozedurbeschreibungsblock fest und setzen in diesem den Zeiger auf den gerade betrachteten Implikationspunkt, wenn dieser Zeiger auf einen Implikationspunkt zeigt, der früher als der gerade betrachtete angelegt wurde. „Früher“ ist dieser dann, wenn er tiefer in dem Kontrollstapel liegt. Der Grund für diese komplizierte Operation liegt daran, daß wir die Liste der Implikationspunkte nur von hinten nach vorne

durchlaufen können, wir aber die Implikationspunkte eigentlich von vorne nach hinten wieder aufbauen müssen. Die Operation *LinkImpPts*, die die Implikationspunkte beim Rücksetzen verwaltet, ist in Abschnitt A.3.2 definiert.

Beispiel 3.5 In Abbildung 3.11 sind Implikationspunkte auf dem Kontrollstapel skizziert. Zu sehen ist die Ausgangssituation und drei Folgezustände, wobei der letzte den Endzustand darstellt. Oben in jedem Zustand sind die Prozedurbeschreibungsblöcke dargestellt, darunter die Implikationspunkte. *IP* gibt das Implikationspunktregister, *A* den gerade betrachteten Implikationspunkt an. Am Anfang haben wir drei Implikationspunkte, die eingegliedert werden müssen: Diese sind gestrichelt dargestellt. Nach und nach werden nun die Implikationspunktzeiger aus den Prozedurbeschreibungsblöcken aktualisiert. Zuletzt zeigt dann auch *IP* auf die wieder eingegliederten Implikationspunkte. □

3.5.2 Geteilte Variablen

Wie wir in den Beispielen 1.1 auf Seite 2 und 2.3 auf Seite 18 gesehen haben, kann dieselbe Variable sowohl in der Prämisse einer hypothetischen Implikation als auch in der Klausel, die die hypothetische Implikation enthält, auftreten. Bei der Formulierung der Ableitbarkeit in Definition 2.12 haben wir dieser Tatsache in den Ableitungsregeln $\vdash^{\mathcal{C}}$ und $\vdash^{\mathcal{A}}$ dadurch Rechnung getragen, daß wir die Substitution σ sowohl auf die Anfragen als auch auf die Programme der resultierenden Tripel anwenden. Wie diese Situation auf der I–WAM–Ebene gelöst werden kann, wird in diesem Abschnitt entwickelt.

Betrachten wir dazu eine Klausel mit einer hypothetischen Implikation, in der eine Variable wie oben beschrieben auftritt; also zum Beispiel:

$$p(x, y) \leftarrow (q(x) \rightarrow r(x, y)).$$

Hier ist x so eine Variable. Diese Variable wollen wir *geteilt* nennen. Aber nicht nur solche Variablen sind geteilt, sondern alle Variablen, die in der Prämisse einer hypothetischen Implikation frei vorkommen. Also ist z in $p(z) \rightarrow q$ geteilt, in $\forall z\, p(z) \rightarrow q$ jedoch nicht. Zu klären ist, wie in der Klausel $q(x)$ auf diese Variable zugegriffen werden kann, und wie die Variable in der aktivierenden Klausel behandelt werden muß.

Diese Situation gibt es natürlich ähnlich auch in imperativen, blockorientierten Programmiersprachen. Eine Situation ähnlich zu der oben angegebenen Klausel in einer blockorientierten Programmiersprache ist in Abbildung 3.12 zu sehen. Links sind die Prozedurvereinbarungen angegeben, rechts das Kontourmodell [WG85, section 2.5], wenn die Prozedur q aus der Prozedur p aufgerufen wurde. Der Pfeil gibt den dynamischen Vorgänger der Prozedur q, das heißt also p, an. Wird die Variable x in q verwendet, so wird diese Variable dem Zustand des *statischen Vorgängers* p entnommen. Hier kann die Prozedur q nur innerhalb der Prozedur p aufgerufen werden: Dies ist der entscheidende Unterschied zwischen der Situation bei blockorientierten Sprachen und der Situation bei den hypothetischen Implikationen. Denn bei hypothetischen Implikationen kann ein Aufruf von q auch von einer anderen Prozedur aus erfolgen.

Anhand dieser Analogie können wir jedoch folgendes feststellen: Eine dynamische Klausel mit geteilten Variablen braucht Zugriff auf die Variablen ihres *statischen* Vorgängers. Wird also eine dynamische Klausel betreten, so müssen die Variablen des statischen Vorgängers dieser Klausel verfügbar gemacht werden. Nun wird auch klar, wieso wir in einem Implikationspunkt

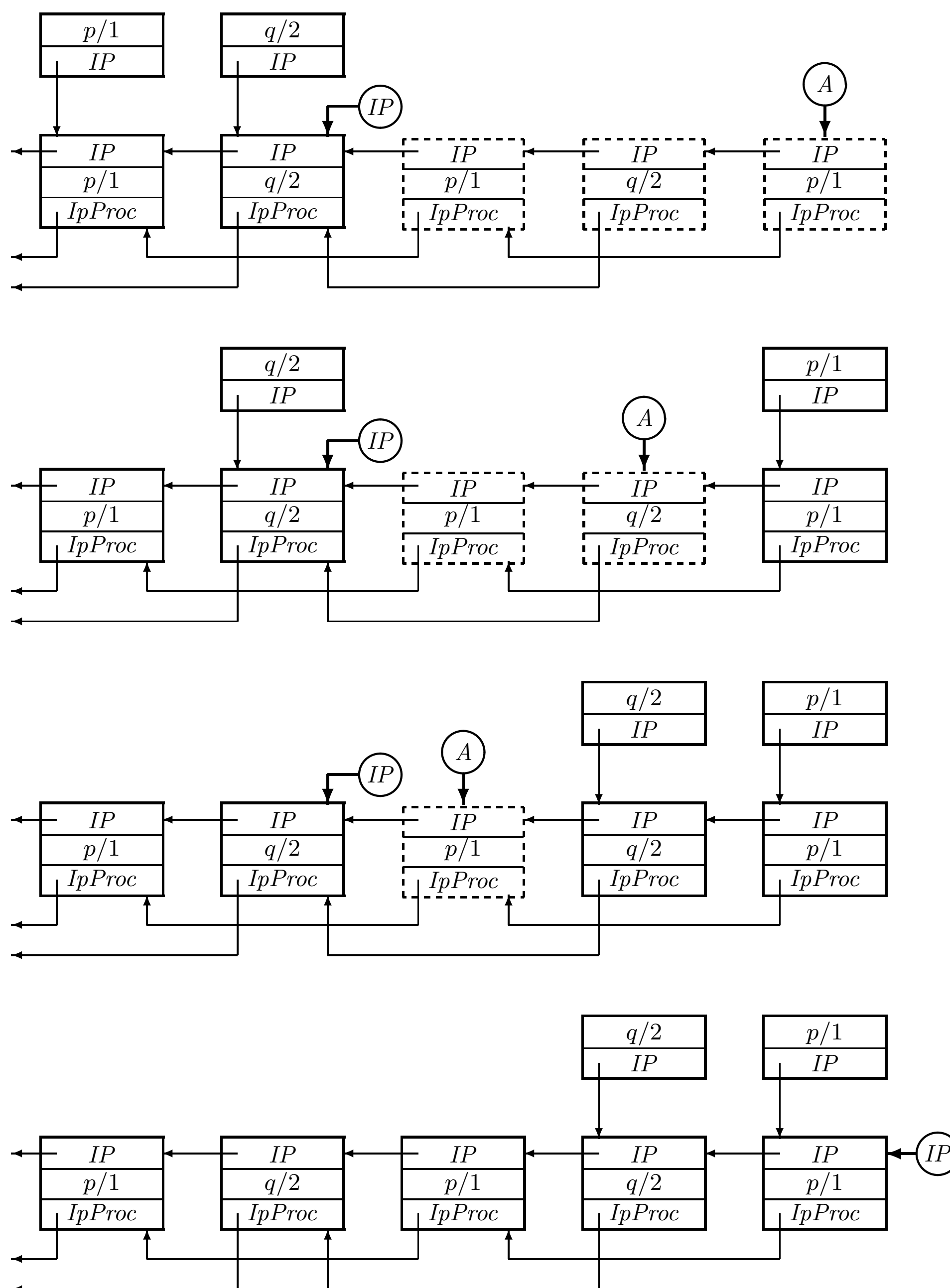


Abbildung 3.11: Wiederaufbau von Implikationspunkten beim Rücksetzen

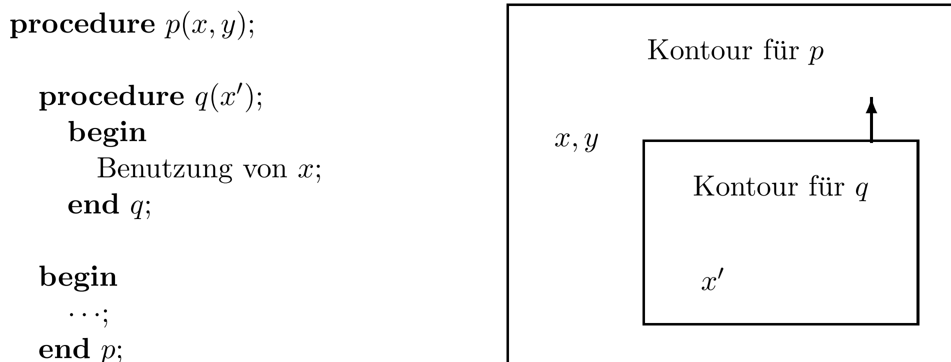


Abbildung 3.12: Prozeduren in einer blockorientierten Programmiersprache

den Umgebungszeiger der aktivierenden Klausel speichern: Dieser gibt gerade den Zugriffspfad auf die Variablen des statischen Vorgängers. Bevor mit der Ausführung einer dynamischen Klausel begonnen wird, wird dem Implikationspunkt für diese Klausel der darin gespeicherte Umgebungszeiger entnommen und in einem neuen Register *EIP*[6] abgelegt.

Wird während der Ausführung der dynamischen Klausel mit diesem neuen Register *EIP* auf geteilte Variablen zugegriffen, so muß der Inhalt dieses Registers bei einem Prozeduraufruf mit `call` durch die Operation `allocate` gespeichert und vor der Rückkehr mit `proceed` durch den Befehl `deallocate` wieder gesetzt werden. Das bedeutet, daß in einer Prozedurumgebung zusätzlich dieses Register gespeichert werden muß. Doch das ist überhaupt nicht wünschenswert, da nur relativ selten dieses Register *EIP* einen sinnvollen Wert besitzt: Eben nur bei der Ausführung der Instruktionen einer dynamischen Klausel. In allen anderen Fällen wird also eine Menge Zeit und Speicher unnütz verschwendet. Das ist um so schlimmer, da dies bei jedem Auf- und Abbauen einer Prozedurumgebung, einer extrem häufigen Aktion, ins Gewicht fällt.

Dieses Dilemma wollen wir auf andere Art und Weise lösen. Anstatt den Inhalt des Registers *EIP* zu retten, lassen wir zu, daß dieser eventuell nach Rückkehr von einem Prozeduraufruf zerstört ist. Dann müssen wir alle Zugriffe auf Variablen des statischen Vorgängers innerhalb einer dynamischen Klausel noch vor dem ersten Prozeduraufruf abwickeln, und die so ermittelten Werte eventuell in der aktuellen Prozedurumgebung abspeichern. Dafür müssen wir zwar zusätzlich Speicher aufwenden, aber danach gibt es in den dynamischen Klauseln keinen Unterschied mehr zu den permanenten Variablen. Insbesondere ist es überflüssig, alle Unifikationsinstruktionen mit einer neuen Adressierungsart für geteilte Variablen auszurüsten.

Dieses Prinzip, das wir herausgearbeitet haben, kann man auch als einen Mechanismus zur erweiterten Parameterübergabe auffassen: Anstatt nach dem Prozeduraufruf nur die normalen Parameter auszuwerten und eventuell abzuspeichern, werden dann auch „Parameter" betrachtet, die aus der aktivierenden Prozedur kommen — via dem Register *EIP*. Diese Art von Zugriff wird [ASU88, Abschnitt 7.4] als „Flacher Zugriff" bezeichnet, im Gegensatz zu „Tiefer Zugriff", der weiter oben geschilderten Alternative.

Wie die geteilten Variablen in der Klausel, in der die hypothetische Implikation auftritt,

[6] *EIP* für Implikationspunktumgebungszeiger, englisch environment implication point pointer

behandelt werden müssen, ist nun trivial. Diese Variablen müssen alle Prozeduraufrufe der Konklusion der hypothetischen Implikation überleben: Sie sind also permanent, und ihr Speicherplatz darf durch die Beschneidung der Prozedurumgebung erst dann frei gegeben werden, wenn die aktivierten Klauseln wieder deaktiviert werden.

3.5.3 Bereitstellung dynamischer Klauseln

Das Prinzip, wie dynamische Klauseln bereitgestellt werden, ist klar: Es müssen Implikationspunkte angelegt werden. Wie dieses sich jedoch in konkreten Instruktionen der I–WAM niederschlägt, ist noch zu klären. Das wollen wir in diesem Abschnitt tun.

Um eine hypothetische Implikation $C_1 \wedge \cdots \wedge C_k \rightarrow G$ zu übersetzen, übersetzen wir zuerst die dynamischen Klauseln C_i für $i = 1, \ldots, k$. Dabei ermitteln wir Proc_i, TagSet_i und Label_i für $i = 1, \ldots, k$. Proc_i bezeichnet den Zeiger auf den Prozedurbeschreibungsblock für das Prädikatensymbol des Kopfes von C_i und Label_i ist die Startadresse der Codesequenz, die C_i implementiert. Ist $\mathit{Tag}_i \in \{\mathrm{Const}, \mathrm{Struct}, \mathrm{List}\}$ der Objekttyp des ersten Argumentes des Kopfatoms von C_i, so ist $\mathit{TagSet}_i = \{\mathit{Tag}_i, \mathrm{Free}\}$. Die Bedeutung dieser Information wird erst im nächsten Teilabschnitt erklärt. Ist die Stelligkeit des Kopfatoms jedoch null oder $\mathit{Tag} = \mathrm{Free}$, so ist $\mathit{TagSet}_i = \{\mathrm{Free}, \mathrm{All}, \mathrm{Const}, \mathrm{Struct}, \mathrm{List}\}$.

Um jetzt die Konklusion G der hypothetischen Implikation zu übersetzen, erzeugen wir die Instruktion `activate N,` k, $[\langle \mathit{Proc}_k, \mathit{TagSet}_k, \mathit{Label}_k \rangle, \ldots, \langle \mathit{Proc}_1, \mathit{TagSet}_1, \mathit{Label}_1 \rangle]$. Bei der Ausführung dieser Instruktion geschieht dann das folgende: Ist der letzte Datenbereich auf dem Kontrollstapel eine Prozedurumgebung, so werden für diese `N` Zellen für die permanenten Variablen reserviert, direkt nach diesen Zellen beginnt der freie Speicher auf dem Kontrollstapel[7]. Sonst beginnt der freie Speicher auf dem Kontrollstapel direkt nach der Stapelspitze.

Beginnend an dieser ersten freien Speicherzelle auf dem Kontrollstapel werden jetzt k Implikationspunkte aufgebaut. Zuerst wird der Implikationspunkt für die Klausel C_k aufgebaut, dann für C_{k-1}, bis zum Implikationspunkt für C_1. Damit kommt der Implikationspunkt für C_1 an die Spitze der Liste der Implikationspunkte: Dies, damit die oben in 3.5.1 spezifizierte Reihenfolge der Implikationspunkte mit der lexikalischen Reihenfolge der C_i übereinstimmt. Die Information aus der Liste der $\langle \mathit{Proc}_i, \mathit{TagSet}_i, \mathit{Label}_i \rangle$ wird jeweils in die Implikationspunkte eingetragen, diese, wie in Teilabschnitt 3.5.1 beschrieben, verkettet, und auch der Umgebungszeiger E in dem Implikationspunkt abgelegt. Danach wird noch das Register IP auf den letzten Implikationspunkt gesetzt.

Der Code für G wird mit der Instruktion `deactivate` k beendet. Bei Ausführung dieser Instruktion werden die letzten k Implikationspunkte auf dem Kontrollstapel, wie in Teilabschnitt 3.5.1 beschrieben, ausgegliedert.

Hier haben wir für jede dynamische Klausel einen Implikationspunkt angelegt. Wir könnten andererseits auch erst alle dynamischen Klauseln einer hypothetischen Implikation mit demselben Kopfprädikat zusammenfassen, für diese Klauseln Indexcode erzeugen, und dann dafür nur einen Implikationspunkt anlegen. Dagegen sprechen aber folgende Argumente: In der Regel werden nur sehr wenige Klauseln mit demselben Prädikat abgelegt. In diesem Fall wird die Effizienz eingeschränkt, da für die Auswahl der Klauseln, die jetzt alle zu einem Implikationspunkt gehören, ein weiterer Wahlpunkt angelegt werden muß. Der Vorteil, statischen Indexcode zu

[7] Dieser Parameter `N` hat somit die gleiche Bedeutung wie bei der Instruktion `call N,` p/k.

haben, wird jetzt sehr teuer durch einen weiteren Wahlpunkt erkauft. In Abschnitt 5.2.2 werden diese beiden Alternativen anhand einer Laufzeitmessung miteinander verglichen.

Die Übersetzung von G läuft jetzt im wesentlichen wie gewohnt ab. Wir müssen nur sicherstellen, daß die geteilten Variablen in G schon initialisiert sind, bevor der erste Prozeduraufruf in G ausgeführt wird. Ist die geteilte Variable `Y`[n] schon vorher durch eine `get`–, `unify`–, `put`– oder `set`–Instruktion initialisiert worden, so bleibt nichts zu tun. Ist dem nicht so, brauchen wir aber eine Instruktion, die nichts anderes macht, als eine permanente Variable in der aktuellen Prozedurumgebung zu erzeugen. Dies leistet die Instruktion `put_shared Y`[n]: Sie erzeugt eine freie Variable in `Y`[n] mit dem Schachtelungsindex *NL*. Analog dazu erzeugt `put_shared_all Y`[n] in `Y`[n] eine allquantifizierte Variable mit dem Schachtelungsindex *NL*. Aus den gleichen Gründen, wie in Abschnitt 3.4.1 erläutert, benötigen wir noch zusätzlich die Instruktionen `put_shared_offset Y`[n]`, O` und `put_shared_all_offset Y`[n]`, O`.

Bei dem Initialisieren der geteilten Variablen muß beachtet werden, daß die erzeugten existentiellen Variablen *unsicher* sind. Das heißt, vor dem Zerstören der Prozedurumgebung durch Verlassen der Klausel, in der G auftritt, muß diese Variable auf die Halde verlegt werden, damit sie weiter erhalten bleibt. Werden die geteilten Variablen später noch benutzt, so kann das Verlegen durch die Instruktionen `put_unsafe_value Y`[n]`, A`[i] oder `set_local_value Y`[n] erfolgen. Da es aber auch geteilte Variablen gibt, die nur initialisiert, aber nicht verwendet werden, brauchen wir eine Instruktion, die solche geteilten Variablen *sicher* erzeugt. Dieses leistet die Instruktion `put_unsafe_shared Y`[n]: Sie erzeugt die freie Variable auf der Halde und schreibt in `Y`[n] eine Referenz auf diese Variable. Natürlich brauchen wir auch noch eine Instruktion, die es gestattet, einen Offset zum aktuellen Schachtelungsindex *NL* anzugeben: `put_unsafe_shared_offset Y`[n]`, O`.

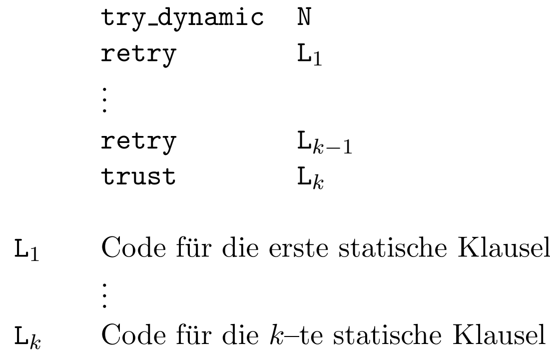
Bei den `put_shared`–Instruktionen sollte darauf geachtet werden, daß diese erst nach Bereitstellung der Argumente für den ersten Prozeduraufruf in G erzeugt werden. Da bei der Bereitstellung der Argumente eventuell noch geteilte Variablen initialisiert werden, können so Instruktionen eingespart werden.

3.5.4 Benutzung dynamischer Klauseln

Im vorigen Abschnitt haben wir betrachtet, welche Instruktionen für die Übersetzung der Konklusion G der hypothetischen Implikation $C_1 \wedge \cdots \wedge C_k \rightarrow G$ benötigt werden. Hier wird betrachtet, wie eine dynamische Klausel C_i übersetzt und benutzt wird. Zuerst wollen wir uns der Benutzung zuwenden.

Haben wir für ein Prädikat p/n statische Klauseln, so werden diese Klauseln durch *Indexinstruktionen* zu einer Prozedur verbunden. Findet ein Prozeduraufruf für p/n statt, so wird anhand dieser Indexinstruktionen eine Klausel zur Ausführung ausgewählt. Setzt die Maschine bis zu dieser Prozedur zurück, wird die nächste Klausel ausgewählt, und bei dieser die Ausführung fortgesetzt.

Gibt es zu diesem Prädikat p/n jedoch aktivierte dynamische Klauseln, so muß zur Abarbeitung die zuletzt aktivierte dynamische Klausel herangezogen werden. Entsprechend muß bei einem Rücksetzvorgang die Abarbeitung bei der davor aktivierten dynamischen Klausel wieder aufgenommen werden. Erst wenn alle aktivierten dynamischen Klauseln fehlgeschlagen sind, werden die statischen Klauseln betrachtet. Die Instruktion, die dynamische Klauseln auswählt, ist `try_dynamic N`. Wie diese Instruktion verwendet wird, ist in Abbildung 3.13 zu sehen.



```
        try_dynamic  N
        retry        L1
        ⋮
        retry        Lk-1
        trust        Lk

L1      Code für die erste statische Klausel
        ⋮
Lk      Code für die k–te statische Klausel
```

Abbildung 3.13: Indexcode für Prozeduren mit dynamischen Klauseln

Wird die Instruktion `try_dynamic N` durch einen Prozeduraufruf für p/n angesprungen, so wird zuerst ein Wahlpunkt auf dem Kontrollstapel angelegt. In diesem Wahlpunkt werden die ersten `N`, also hier n, Argumentregister abgelegt. Dann werden alle anderen Register gespeichert. Ein Wahlpunkt enthält bei der normalen WAM eine *Alternative*. Trifft die WAM beim Rücksetzen auf diesen Wahlpunkt, so wird nach Restaurieren des Maschinenzustandes die Abarbeitung an dieser Adresse fortgesetzt. Analog dazu enthält ein Wahlpunkt bei der I–WAM eine *dynamische* und eine *statische Alternative*. Die dynamische Alternative enthält die Adresse des nächsten zu betrachtenden Implikationspunktes, falls vorhanden, sonst den Wert **nil**. Die statische Alternative enthält die Adresse auf ein alternatives Codestück.

Das Register *PROC* enthält einen Zeiger auf den Prozedurbeschreibungsblock der angesprungenen Prozedur. Mit Hilfe dieses Registers legen wir in *DynAlt* die Adresse des ersten, in *NextDynAlt* die Adresse des darauf folgenden Implikationspunktes für diese Prozedur ab. Hat *DynAlt* den Wert **nil**, so gibt es keine aktivierte dynamische Klausel: In dem Wahlpunkt wird als dynamische Alternative **nil** eingetragen und die Abarbeitung bei der folgenden Instruktion, also bei `retry` L_1, fortgesetzt. Diese Instruktion wird dann die statische Alternative in gewohnter Weise setzen, und die Ausführung der statischen Klauseln weiter steuern.

Ist der Wert von *DynAlt* ungleich **nil**, so muß die Abarbeitung bei der dynamischen Klausel fortgesetzt werden, die durch den Implikationspunkt, auf den *DynAlt* zeigt, beschrieben wird. Bevor dies geschieht, wird als dynamische Alternative *NextDynAlt* und als statische Alternative die Adresse der folgenden Instruktion, `retry` L_1, in dem Wahlpunkt abgelegt. Dann wird das Register *EIP* mit dem Wert aus dem Implikationspunkt *DynAlt*↑ initialisiert, um den Zugriffspfad auf die geteilten Variablen einzustellen. Anschließend folgt ein Sprung auf die Codeadresse der dynamischen Klausel; auch diese wird dem Implikationspunkt *DynAlt*↑ entnommen.

Beim Rücksetzen wird nach dem Restaurieren des Maschinenzustandes aus den Daten des Wahlpunktes analog zu oben verfahren. Gibt es noch eine dynamische Alternative im Wahlpunkt, so wird als neue dynamische Alternative der darauf folgende Implikationspunkt im Wahlpunkt eingetragen und die Abarbeitung bei der vorher gespeicherten dynamischen Alternative fortgesetzt. Dies natürlich erst, nachdem das Register *EIP* und die Sprungadresse dem Implikationspunkt entnommen wurden. Gibt es keine dynamische Alternative, so wird bei der statischen Alternative fortgesetzt.

Bisher waren wir davon ausgegangen, daß es auch statische Klauseln gibt. Gibt es diese nicht, so reduziert sich der ganze Indexcode für ein Prädikat auf die Instruktion `trust_dynamic N`. Deren Semantik ist im wesentlichen gleich der von `try_dynamic N`, die Unterschiede sind nur kleine Optimierungen. Gibt es keinen Implikationspunkt für diese Prozedur, so wird gleich ein Rücksetzen der Maschine ausgelöst. Gibt es nur einen Implikationspunkt, so wird kein Wahlpunkt mehr angelegt, sondern nach dem Einstellen des Registers *EIP* direkt die dynamische Klausel angesprungen.

In der obigen Beschreibung werden *alle* Implikationspunkte für eine Prozedur sequentiell nacheinander für die Ausführung betrachtet. Im folgenden werden wir dieses Konzept noch verfeinern. In einem Implikationspunkt wird auch die Menge der Objekttypen *TagSet* des ersten Argumentes der dynamischen Klausel, wie im vorigen Teilabschnitt beschrieben, abgelegt. Wird beim Prozeduraufruf der Objekttyp des ersten Argumentes, also des Argumentregisters `A[1]`, in *Tag* bestimmt, so brauchen nur Implikationspunkte mit $Tag \in TagSet$ betrachtet werden; alle anderen Implikationspunkte führen zur Ausführung einer dynamischen Klausel, deren erstes Argument sich nicht mit dem ersten Argument des Prozeduraufrufs unifizieren läßt. Das ist die gleiche Idee, die bei der WAM der `switch_on_type`–Instruktion zugrundeliegt.

Bisher haben wir nur diskutiert, wie die Ausführung des Codes von dynamischen Klauseln gestartet wird. Es ist noch zu klären, wie in dynamischen Klauseln der Zugriff auf geteilte Variablen des statischen Vorgängers abgewickelt wird. Dazu führen wir die beiden Instruktionen `get_shared_variable V[`n`], SY[`m`]` und `get_shared_value V[`n`], SY[`m`]` ein. Diese Formulierung betont noch einmal die Idee des erweiterten Mechanismus zur Parameterübergabe. Die Bezeichnung `SY[`m`]` steht dabei für eine permanente Variable, die via dem Register *EIP* adressiert wird. Die Instruktion `get_shared_variable V[`n`], SY[`m`]` kopiert den *Wert* von `SY[`m`]` in die Variable `V[`n`]`. Mit Wert ist gemeint, daß wenn `SY[`m`]` eine freie Variable enthält, die ja nicht kopiert werden darf, eine Referenz auf diese Variable in `V[`n`]` abgelegt wird. `get_shared_value V[`n`], SY[`m`]` wird dagegen benutzt, wenn `V[`n`]` bereits initialisiert ist: `V[`n`]` und `SY[`m`]` werden einfach miteinander unifiziert. Die Überlegung, ob diese Vorgehensweise korrekt ist, entspricht genau der für die Korrektheit von `get_variable` und `get_value`.

3.6 Sonstiges

Bei der Auswahl von Implikationspunkten werden die Objekttypen des ersten Argumentes betrachtet. Insbesondere werden nur dynamische Klauseln mit einer freien Variable im ersten Argument des Kopfatoms betrachtet, wenn das erste Argument des Prozeduraufrufs eine allquantifizierte Variable ist. Denn alle Klauseln mit einem anderen Objekttyp sind von vornerein zum Fehlschlagen verurteilt. Diese Berücksichtigung der allquantifizierten Variablen bei der Auswahl von Implikationspunkten übertragen wir jetzt auch auf die `switch_on_type`–Instruktion.

Dazu führen wir die Instruktion `switch_on_ext_type LE, LA, LS, LL, LC` ein. Bei Ausführung dieser Instruktion erfolgt ein Sprung zu der Adresse `LE`, `LA`, `LS`, `LL` oder `LC`, je nachdem ob der Objekttyp des Terms im Argumentregister `A[1]` gleich Free, All, Struct, List oder Const ist. Ist die ausgewählte Adresse gleich **nil**, so wird Rücksetzen der Maschine ausgelöst.

3.7 Zwei abschließende Beispiele

Um den Entwurf noch einmal zu verdeutlichen, werden hier zwei Beispiele vorgestellt. Der Code ist jeweils so angegeben, wie der Übersetzer des KAIN–Prolog Systems — siehe dazu auch Kapitel 4 — ihn erzeugt.

Beispiel 3.6 Als erstes Beispiel wollen wir eine Klausel des Programms von Beispiel 2.4 auf Seite 18 in I–WAM Instruktionen übersetzen.

Klausel

$$steril(x) \leftarrow \forall y\,(in(y,x) \rightarrow tot(y))$$

Allokation

Name	Statische Klausel	Dynamische Klausel	Geteilt
x	Y[1]	A[2]	+
y	Y[2]	A[1]	+

Code

```
    allocate                                                 steril(
    get_variable         Y[1], A[1]                                x) ←
    quantify                                                       ∀y (
    activate             2, 1, [⟨in/2, {Free, All}, L⟩]                  ··· →
    put_variable_all     Y[2], A[1]                                          tot(y
    call                 2, tot/1                                                 )
    deactivate           1
    dequantify                                                     )
    deallocate
    proceed

L   get_shared_value     A[1], SY[2]                         in(y,
    get_shared_value     A[2], SY[1]                              x
    proceed                                                        )
```

□

Beispiel 3.7 Das zweite Beispiel ist rein künstlich. In ihm sind viele geteilte Variablen unterschiedlichster Art enthalten.

Klausel

$$\begin{array}{rcl} p(f(x),y) & \leftarrow & \forall x\, q([x],y) \\ & \wedge & \forall z\ (\forall y\ (q(x,y) \leftarrow r(z) \wedge s(u)) \rightarrow s(y) \wedge t(x)) \end{array}$$

Allokation

Name	Gebunden	Statische Klausel	Dynamische Klausel	Geteilt
x	$-$	`Y[1]`	`A[1]`	$+$
x	$+$	`void`	—	$-$
y	$-$	`Y[4]`	—	$-$
y	$+$	—	`void`	$-$
z	$+$	`Y[2]`	`X[3]`	$+$
u	$-$	`Y[3]`	`Y[1]`	$+$

Code

	Instruktion	Argumente	Klausel
	`get_structure`	$f/1$, `A[1]`	$p(f($
	`allocate`		
	`unify_variable`	`Y[1]`	$x),$
	`get_variable`	`Y[4]`, `A[2]`	$y) \leftarrow$
	`quantify`		$\forall x\,($
	`put_list`	`A[1]`	$q([$
	`set_void_all`		x
	`set_constant`	`[]`	$],$
			y
	`call`	`4`, $q/2$	$)$
	`dequantify`		$)\wedge$
	`quantify`		$\forall z\,($
	`activate`	`4`, `1`, $[\langle q/2,$ $\{\mathrm{Free}, \mathrm{All}, \mathrm{Struct}, \mathrm{List}, \mathrm{Const}\},$ `L`$\rangle]$	$\cdots \rightarrow ($
	`put_value`	`Y[4]`, `A[1]`	$s(y$
	`put_unsafe_shared_offset`	`Y[3]`, `1`	
	`put_shared_all`	`Y[2]`	
	`call`	`3`, $s/1$	$)\wedge$
	`put_value`	`Y[1]`, `A[1]`	$t(x$
	`call`	`3`, $t/1$	$)$
	`deactivate`	`1`	$)$
	`dequantify`		$)$
	`deallocate`		
	`proceed`		
`L`	`get_shared_value`	`X[1]`, `SY[1]`	$q(x, y) \leftarrow$
	`allocate`		
	`get_shared_variable`	`X[3]`, `SY[2]`	
	`get_shared_variable`	`Y[1]`, `SY[3]`	
	`put_value`	`X[3]`, `A[1]`	$r(z$
	`call`	`1`, $r/1$	$)\wedge$
	`put_value`	`Y[1]`, `A[1]`	$s(u$

```
deallocate
execute              s/1                    )
```

Die beiden unterstrichenen Instruktionen sind nur aus Gründen der Verstehbarkeit des Beispiels vorhanden, beide zusammen können wegoptimiert werden.

□

Kapitel 4

Das KAIN–PROLOG System

In diesem Kapitel wird die Realisierung des KAIN–PROLOG Systems — Karlsruher intuitionistisches PROLOG — beschrieben. Ausgehend von einem Schichtenmodell werden die zentralen Aspekte, die dieser Erweiterung des KA–PROLOG Systems [BBL+91, Nei87] zugrundeliegen, diskutiert und in ihrer Funktionalität vorgestellt. Dieses Kapitel ist jedoch keine Sprachdefinition oder Bedienungsanleitung, dies läßt sich in Anhang B finden.

4.1 Schichtenmodell

Bevor mit der Beschreibung begonnen wird, wollen wir ein Blick auf die Architektur des KAIN–PROLOG Systems — im weiteren einfach nur als KAIN System bezeichnet — werfen. In Abbildung 4.1 sind verschiedene Schichten des Systems skizziert.

Die *Basisschicht* ist hier natürlich kein Gegenstand weiterer Untersuchung. Die KAIN–Maschine der *Maschinenschicht* ist eine Realisierung der in dem Kapitel 3 und dem Anhang A ausführlich beschriebenen I–WAM. Somit entfällt in diesem Kapitel eine weitere Beschäftigung hiermit. Die zweite Komponente der Maschinenschicht besteht aus dem Laufzeitsystem. Dessen Funktionalität umfaßt solche Aufgaben wie Speicherverwaltung, Symboltabellenverwaltung oder die Organisation der Codeablage. Diese im wesentlichen implementierungsabhängigen Teile werden hier nicht besprochen. Denn diese Komponente des KAIN Systems hat gegenüber dem KAP System kaum eine Erweiterung erfahren, als Beschreibung kann somit [Nei87] dienen.

Die *Abbildungsschicht* wird auch nur kurz untersucht. Den Hauptteil dieses Kapitels wird die Beschreibung des Übersetzers und des Debuggers ausmachen. Als Grundlage der Beschreibung dient die *Programmiersprache* des KAIN Systems; ihr wollen wir uns zuerst zuwenden.

4.2 Die Programmiersprache

Das syntaktische Grundgerüst der Programmiersprache bilden in KAIN–PROLOG die *Terme*. Die Term–Grammatik ist genauso definiert, wie in der Sprachdefinition von KA–PROLOG beschrieben [BBL+91].

Anstatt eine Grammatik für Anfragen und Klauseln zu formulieren, werden wir hier nur eine Abbildung der KAIN–Programmklauseln und KAIN–Anfragen auf die bisher betrachteten logischen Formeln angeben. Eine genaue Definition läßt sich in Abschnitt B.1.2 finden.

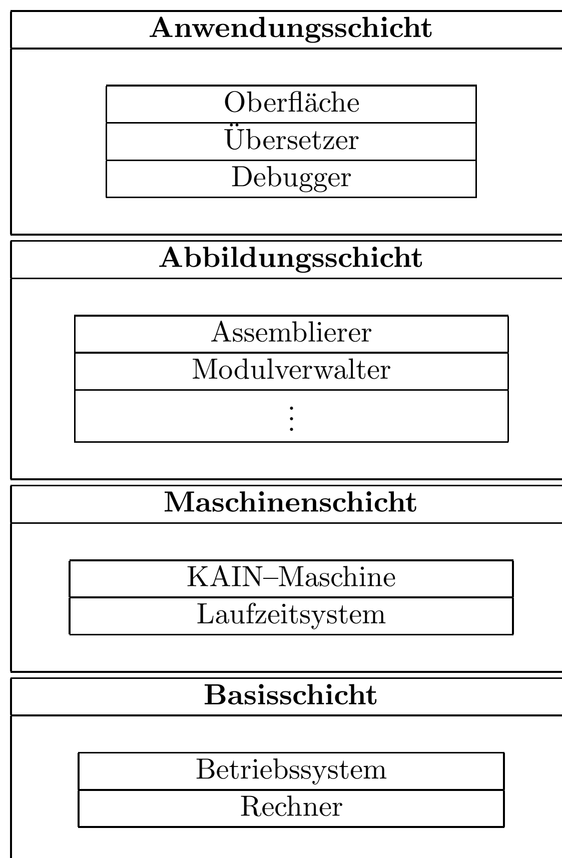


Abbildung 4.1: Schichtenmodell des KAIN–PROLOG Systems

In Abbildung 4.2 ist skizziert, wie die Entsprechung der KAIN–Klauseln und –Anfragen zu den bisherigen Formeln ist. Dabei ordnen die Abbildungen K, R und A jeweils KAIN–Klauseln, KAIN–Anfragen und KAIN–Atomen die entsprechenden logischen Formeln zu. Fast alle Konstrukte sind nur andere Schreibweisen, etwas neues ist nur die Möglichkeit, bei einer verschachtelten Klausel ein `all` voranzustellen: Dies entspricht gerade dem Allabschluß.

4.3 Der Übersetzer

In diesem Abschnitt wird der Übersetzer des KAIN–Systems dargestellt. Im Vordergrund stehen dabei die Aspekte, die eine Erweiterung des normalen Übersetzungsprozesses für PROLOG ausmachen. Die hier nicht genannten Aspekte lassen sich dann der Beschreibung des Übersetzers für das KA–PROLOG System [Lin87] entnehmen.

Der Übersetzer besteht aus zwei Komponenten: Einmal dem Deklarationsübersetzer, der das Modulkonzept realisiert. Dessen etwas veränderte Aufgaben sind in dem Abschnitt 4.4 näher beschrieben. Zum anderen aus dem eigentlichen Übersetzer, der die Klauseln übersetzt und diese zu Prozeduren zusammenfügt. Dieser wird in diesem Abschnitt beschrieben, dargestellt

$$
\begin{array}{lcl}
K[\![\texttt{all}\ (\texttt{X}_1,\ldots,\texttt{X}_\texttt{k})\ :\ \mathit{Klausel}]\!] & = & \forall x_1 \cdots \forall x_k\, K[\![\mathit{Klausel}]\!] \\
K[\![\texttt{all}\ \mathit{Klausel}]\!] & = & \tilde{\forall}\, K[\![\mathit{Klausel}]\!] \\
K[\![\mathit{Atom}\ :-\ \mathit{Rumpf}]\!] & = & A[\![\mathit{Atom}]\!] \leftarrow R[\![\mathit{Rumpf}]\!] \\
K[\![\mathit{Atom}]\!] & = & A[\![\mathit{Atom}]\!] \\
\\
R[\![\mathit{Rumpf}\ ,\ \mathit{Rumpf}]\!] & = & R[\![\mathit{Rumpf}]\!] \wedge R[\![\mathit{Rumpf}]\!] \\
R[\![\mathit{Rumpf}\ ;\ \mathit{Rumpf}]\!] & = & R[\![\mathit{Rumpf}]\!] \vee R[\![\mathit{Rumpf}]\!] \\
R[\![\texttt{all}\ (\texttt{X}_1,\ldots,\texttt{X}_\texttt{k})\ :\ \mathit{Rumpf}]\!] & = & \forall x_1 \cdots \forall x_k\, R[\![\mathit{Rumpf}]\!] \\
R[\![\texttt{exists}\ (\texttt{X}_1,\ldots,\texttt{X}_\texttt{k})\ :\ \mathit{Rumpf}]\!] & = & \exists x_1 \cdots \exists x_k\, R[\![\mathit{Rumpf}]\!] \\
R[\![\mathit{Klausel}_1,\ldots,\mathit{Klausel}_k\ -:\ \mathit{Rumpf}]\!] & = & K[\![\mathit{Klausel}_1]\!] \wedge \ldots \wedge K[\![\mathit{Klausel}_k]\!] \rightarrow R[\![\mathit{Rumpf}]\!] \\
R[\![\mathit{Atom}]\!] & = & A[\![\mathit{Atom}]\!]
\end{array}
$$

Abbildung 4.2: Bedeutung der Sprachkonstrukte

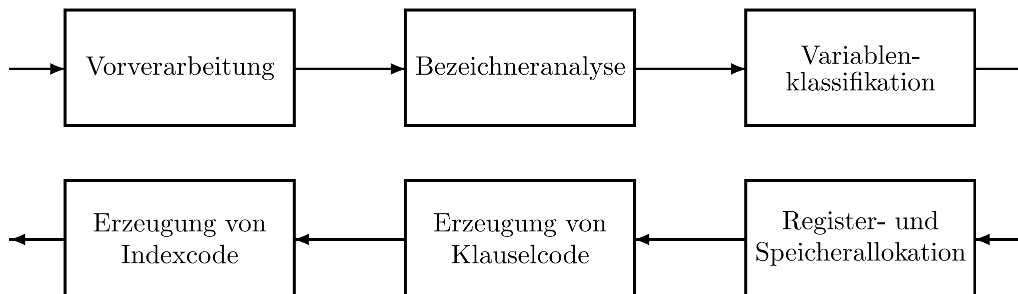


Abbildung 4.3: Struktur des Übersetzers

sind dessen verschiedene Teilaufgaben und deren Reihenfolge in Abbildung 4.3.

Die lexikalische und syntaktische Analyse der eingelesenen Programme wird wie sonst auch von dem Systemteil übernommen, das die Terme einliest. Die eingelesenen Terme werden dann einer *Vorverarbeitung* unterzogen. Dabei wird eine Transformation der eingelesenen Klauseln in eine Darstellung vorgenommen, die zur Durchführung des Übersetzungsprozesses besser geeignet ist. Dieser Teil ist rein technischer Natur, für eine Einsicht in den Übersetzungsvorgang ist das nicht weiter von Interesse.

Auch die Register- und Speicherallokation hat sich nicht geändert. Bei der Beschreibung der anderen Komponenten wollen wir jetzt etwas weiter ins Detail gehen.

4.3.1 Bezeichneranalyse

Bei einem PROLOG Programm bezeichnet ein Variablenname immer dasselbe Objekt. Das ist jetzt nicht mehr so. Einerseits kann durch das Auftreten von Quantoren ein und derselbe Variablenname verschiedene Variablen bezeichnen, andererseits bezeichnen gleiche Variablennamen in einer Klausel und einer darin enthaltenen dynamischen Klausel verschiedene Objekte hinsichtlich der Behandlung bei der Speicherallokation.

Die Aufgabe der Bezeichneranalyse ist es jetzt, festzustellen, welcher Variablenname welches Objekt bezeichnet. Zusätzlich wird auch festgestellt, welche Variablen geteilt sind. Dazu wird zwischen der *Definition* und der *Benutzung* eines Variablennamens unterscheiden. Definitionen werden einmal von Quantoren eingeführt: Die quantifizierten Variablen werden definiert. Zum anderen gibt es auch implizite Definitionen: Wird eine Variable das erste Mal benutzt, ohne vorher explizit durch einen Quantor definiert zu werden, so ist dieses Auftreten gleichzeitig eine Definition.

Um jetzt für eine Klausel alle definierten Objekte festzustellen, und das Auftreten von Namen den entsprechenden Objekten zuzuordnen, wird wie folgt vorgegangen. Es wird für diese Klausel eine Liste von Paaren berechnet, wobei ein Paar aus dem Namen und dem entsprechenden Objekt besteht. Ein Auftreten eines Variablennamens wird dann durch das bezeichnete Objekt ersetzt; dieses Objekt wird in der Paarliste nachgeschlagen, wobei das passende Paar gewählt wird, das in der Liste am weitesten vorne steht. Eine Definition fügt der Liste ein Paar am Anfang hinzu, wird der Bereich dieser Definition wieder verlassen, so wird das Paar wieder entfernt. Bei der Analyse einer verschachtelten Klausel wird dann wieder mit einer leeren Liste begonnen. Diese Technik zur Bezeichneranalyse ist also wie bei blockorientierten Programmiersprachen realisiert, siehe dazu zum Beispiel [WG85, subsection 9.1.1].

Nachdem die Bezeichner durch die eigentlichen Objekte ersetzt worden sind, wird noch berechnet, welche Variablenobjekte einer dynamischen Klausel geteilt sind. Dazu werden zuerst für jede dynamische Klausel die Variablenobjekte bestimmt, die frei in ihr auftreten. Genau das sind die geteilten Variablen, siehe dazu auch Abschnitt 3.5.2. Um später Instruktionen zu erzeugen, die diese Variablenobjekte mit den Werten der Objekte aus der Klausel, in der die dynamische Klausel enthalten ist, initialisieren, werden diese initialisierenden Objekte bestimmt. Dies geht mit Hilfe der oben berechneten Paarlisten für jede Klausel: Die unterschiedlichen Objekte haben den gleichen Variablennamen.

4.3.2 Variablenklassifikation

Unter dem Begriff der Variablenklassifikation versteht man, festzustellen, ob eine Variable vom Typ „void“, „permanent“ oder „temporär“ ist. Variablen vom Typ „void“ erhalten keinen benannten Speicherplatz innerhalb einer Klausel, permanente Variablen werden auf dem Kontrollstapel in der aktuellen Prozedurumgebung und temporäre Variablen in einem Datenregister abgelegt.

Die Klassifikation findet nach den Vorkommen einer Variablen in einer Klausel statt. Die neuen Aspekte der Variablenklassifikation hatten wir schon in Kapitel 3 dargelegt. Hier wollen wir die Kriterien zur Variablenklassifikation noch einmal komplett zusammenfassen:

void Die Variable tritt innerhalb der Klausel nur einmal auf.

permanent Die Variable tritt in mehr als zwei Prozeduraufrufen auf, oder tritt in einer dynamischen Klausel der Prämisse einer hypothetischen Implikation auf, die in der Klausel enthalten ist.

temporär Dies sind alle sonstigen Variablen.

Bei einer permanenten Variable muß zusätzlich bestimmt werden, ob diese *sicher* ist: Sicher ist sie dann, wenn sie kein existentiell quantifiziertes Variablenobjekt enthalten kann, das

direkt in der Prozedurumgebung und nicht auf der Halde erzeugt wurde. Dieses Kriterium ist also unverändert geblieben, es muß jetzt nur zusätzlich bei der Erzeugung von Instruktionen beachtet werden, die geteilte Variablen vor dem ersten Prozeduraufruf in der Konklusion einer hypothetischen Implikation initialisieren.

An die Klassifikation der Variablen schließt sich die Registerallokation für temporäre Variablen und die Speicherallokation für permanente Variablen an. Die Klassifikation liefert in derselben Art und Weise wie bei KA–PROLOG die für die Allokation nötige Information, deshalb ist diese auch unverändert geblieben.

4.3.3 Erzeugung von Klauselcode

In Kapitel 3 hatten wir bei dem Entwurf des erweiterten Instruktionssatzes schon gesehen, wie Klauseln und Anfragen in Instruktionssequenzen übersetzt werden. In diesem Abschnitt wollen wir die Erzeugung von Code anhand einer Reihe von Übersetzungsschemata noch einmal zusammenfassen.

4.3.3.1 Übersetzungsschemata

Atomare Klauseln Eine Klausel $p(t_1, \ldots, t_k)$, die nur aus einem Atom besteht, wird nach dem folgenden Schema übersetzt:

$\langle$Code für $p(t_1, \ldots, t_k)\rangle =$
    $\langle$Unifikation von $t_1\rangle$
    $\vdots$
    $\langle$Unifikation von $t_k\rangle$
    `proceed`

Normale Klauseln Eine Klausel $p(t_1, \ldots, t_k) \leftarrow G$ wird wie folgt übersetzt:

$\langle$Code für $p(t_1, \ldots, t_k) \leftarrow G\rangle =$
    `allocate`
    $\langle$Unifikation von $t_1\rangle$
    $\vdots$
    $\langle$Unifikation von $t_k\rangle$
    $\langle$Initialisierung geteilter Variablen$\rangle$
    $\langle$Code für $G\rangle$
    `deallocate`
    `proceed`

Die Initialisierung der geteilten Variablen in dem obigen Schema wird dadurch realisiert, daß `get_shared_variable V`$[n]$`, SY`$[m]$ Instruktionen erzeugt werden. Dabei werden nur Variablen betrachtet, die noch nicht bei der Unifikation der Kopfargumente initialisiert wurden.

Tabelle 4.1: Initialisierung geteilter Variablen

Quantifizierung	Offset	Sicher	Instruktion	
$\exists$	$-$	$+$	`put_shared_variable`	Y[n]
		$-$	`put_unsafe_shared_variable`	Y[n]
	$+$	$+$	`put_shared_variable_offset`	Y[n], O
		$-$	`put_unsafe_shared_variable_offset`	Y[n], O
$\forall$	$-$	$+$	`put_shared_variable_all`	Y[n]
	$+$	$+$	`put_shared_variable_all_offset`	Y[n], O

Konjunktionen Eine Konjunktion $G_1 \wedge G_2$ von zwei Rümpfen wird nach folgendem Schema übersetzt:

⟨Code für $G_1 \wedge G_2$⟩ =
  ⟨Code für G_1⟩
  ⟨Code für G_2⟩

Allquantoren Die Übersetzung von $\forall \cdots G$ folgt dem nächsten Schema:

⟨Code für $\forall \cdots G$⟩ =
  `quantify`
  ⟨Code für G⟩
  `dequantify`

Existenzquantoren Wie schon erwähnt, schlagen sich Existenzquantoren nicht in zusätzlichen Instruktionen nieder:

⟨Code für $\exists \cdots G$⟩ =
  ⟨Code für G⟩

Hypothetische Implikationen Eine hypothetische Implikation $C_1 \wedge \cdots \wedge C_k \to G$ wird so übersetzt:

⟨Code für $C_1 \wedge \cdots \wedge C_k \to G$⟩ =
  `activate` ⟨Größe der Prozedurumgebung⟩,
    k,
    ⟨Information über die C_i⟩
  ⟨Initialisierung der geteilten Variablen⟩
  ⟨Code für G⟩
  `deactivate` k

Wie die Information über die C_i aussehen, ist ausführlich in Abschnitt 3.5.3 beschrieben. Die Initialisierung der geteilten Variablen für die dynamischen Klauseln C_i erfolgt durch die Instruktionen in Tabelle 4.1.

Prozeduraufruf Schließlich wird ein Rumpf $q(t_1, \ldots, t_k)$, der nur aus einem Atom besteht, folgendermaßen übersetzt:

⟨Code für $q(t_1, \ldots, t_k)$⟩ =
⟨Bereitstellung von t_1⟩
⋮
⟨Bereitstellung von t_k⟩
`call` ⟨Größe der Prozedurumgebung⟩, q/k

Unifikation von Kopfargumenten Die Instruktionsfolge, die für die Unifikation eines Kopfterms t erzeugt wird, richtet sich nach der Art des Terms t. Nachfolgend sind die verschiedenen Schemata aufgeführt.

⟨Unifikation von $t = f(t_1, \ldots, t_k)$⟩ =
`get_structure` f/k, `A`$[i]$
⟨Extraktion von t_1⟩
⋮
⟨Extraktion von t_k⟩
⟨Unifikation von t_1, geschachtelt⟩
⋮
⟨Unifikation von t_k, geschachtelt⟩

⟨Unifikation von $t = [t_1|t_2]$⟩ =
`get_list A`$[i]$
⟨Extraktion von t_1⟩
⟨Extraktion von t_2⟩
⟨Unifikation von t_1, geschachtelt⟩
⟨Unifikation von t_2, geschachtelt⟩

⟨Unifikation von t_i = `C`, `C` Konstante⟩ =
`get_constant C, A`$[i]$

Wie die entsprechenden Instruktionssequenzen für die Unifikation mit einer Variablen aussehen ist Tabelle 4.2 zu entnehmen.

Unifikation von geschachtelten Kopfargumenten Das Aufbrechen von Strukturen wird durch eine Codesequenz vorgenommen, die die Argumente eines Terms entweder direkt unifizieren, oder diese erst in einem Register abspeichern, um später mit der geschachtelten Unifikation fortzufahren.

⟨Extraktion von $t = f(t_1, \ldots, t_k)$⟩ =
`unify_variable X`$[n]$

⟨Extraktion von $t = [t_1|t_2]$⟩ =
`unify_variable X`$[n]$

Tabelle 4.2: Unifikation mit einer Variable

Auftreten	Geteilt	Instruktionssequenz	
Einziges	+	`get_shared_value`	A[i], SY[m]
Erstes	−	`get_variable`	V[n], A[i]
	+	`get_variable`	V[n], A[i]
		`get_shared_value`	V[n], SY[m]
Weiteres	egal	`get_value`	V[n], A[i]

Tabelle 4.3: Unifikation mit einer verschachtelten Variable

Auftreten	Geteilt	Instruktionssequenz	
Einziges	−	`unify_void`	
Erstes	−	`unify_variable`	V[n]
	+	`unify_variable`	V[n]
		`get_shared_value`	V[n], SY[m]
Weiteres	egal	`unify_value`	V[n]
Letztes unsicheres	egal	`unify_local_value`	Y[n]

Die oben genannten Register X[n] sind die, aus denen bei der späteren verschachtelten Unifikation die Teilterme entnommen werden.

Die entsprechenden Codesequenzen für das Aufbrechen von Argumenten, die Variablen sind, können Tabelle 4.3 entnommen werden. Der noch fehlende Fall für die Extraktion einer Konstanten schlägt sich in dem folgenden Schema nieder:

⟨Extraktion von $t =$ C, C Konstante⟩ =
    unify_constant C

Nach der Extraktion der Argumente folgen noch die Schemata für die eigentliche Unifikation mit einem verschachtelten Term. Nur in den Fällen, in denen der verschachtelte Term wieder keine Variable oder Konstante ist, muß die Unifikation fortgesetzt werden. Diese beiden Schemata lauten wie folgt:

⟨Unifikation von $t = f(t_1, \ldots, t_k)$, geschachtelt⟩ =
    get_structure f/k**,** X[n]
    ⟨Extraktion von t_1⟩
    ⋮
    ⟨Extraktion von t_k⟩
    ⟨Unifikation von t_1, geschachtelt⟩
    ⋮
    ⟨Unifikation von t_k, geschachtelt⟩

$\langle$Unifikation von $t = [t_1|t_2]$, geschachtelt$\rangle =$

    get_list X[n]
    $\langle$Extraktion von $t_1\rangle$
    $\langle$Extraktion von $t_2\rangle$
    $\langle$Unifikation von t_1, geschachtelt$\rangle$
    $\langle$Unifikation von t_2, geschachtelt$\rangle$

Bereitstellung von Rumpfargumenten Die Instruktionsfolge für die Bereitstellung eines Terms t richtet sich wiederum nach der Art des Terms t.

$\langle$Bereitstellung von $t = f(t_1, \ldots, t_k)\rangle =$

    $\langle$Bereitstellung von t_1, geschachtelt$\rangle$
    $\vdots$
    $\langle$Bereitstellung von t_k, geschachtelt$\rangle$
    put_structure f/k, A[i]
    $\langle$Ablegen von $t_1\rangle$
    $\vdots$
    $\langle$Ablegen von $t_k\rangle$

$\langle$Bereitstellung von $t = [t_1|t_2]\rangle =$

    $\langle$Bereitstellung von t_1, geschachtelt$\rangle$
    $\langle$Bereitstellung von t_2, geschachtelt$\rangle$
    put_list A[i]
    $\langle$Ablegen von $t_1\rangle$
    $\langle$Ablegen von $t_2\rangle$

$\langle$Bereitstellung von $t_i =$ C, C Konstante$\rangle =$

    put_constant C, A[i]

Wie die entsprechenden Instruktionssequenzen für die Bereitstellung einer Variablen aussehen, ist Tabelle 4.4 zu entnehmen.

Bereitstellung von geschachtelten Rumpfargumenten Bei der Bereitstellung von geschachtelten Termen werden zuerst die nicht atomaren Teilterme bereitgestellt. Die Schemata für die zu erzeugenden Instruktionen sehen dann wie folgt aus:

$\langle$Bereitstellung von $t = f(t_1, \ldots, t_k)$, geschachtelt$\rangle =$

    $\langle$Bereitstellung von t_1, geschachtelt$\rangle$
    $\vdots$
    $\langle$Bereitstellung von t_k, geschachtelt$\rangle$
    put_structure f/k, X[n]
    $\langle$Ablegen von $t_1\rangle$
    $\vdots$
    $\langle$Ablegen von $t_k\rangle$

Tabelle 4.4: Instruktionen für Bereitstellung einer Variablen

Auftreten	Quantifizierung	Offset	Instruktion	
Einziges	$\exists$	$-$	put_void	A[i]
		$+$	put_void_offset	A[i], O
	$\forall$	$-$	put_void_all	A[i]
		$+$	put_void_all_offset	A[i], O
Erstes	$\exists$	$-$	put_variable	V[n], A[i]
		$+$	put_variable_offset	V[n], A[i], O
	$\forall$	$-$	put_variable_all	V[n], A[i]
		$+$	put_variable_all_offset	V[n], A[i], O
Weiteres	egal	egal	put_value	V[n], A[i]
Letztes unsicheres	egal	egal	put_unsafe_value	V[n], A[i]

$\langle$Bereitstellung von $t = [t_1|t_2]$, geschachtelt$\rangle =$
    $\langle$Bereitstellung von t_1, geschachtelt$\rangle$
    $\langle$Bereitstellung von t_2, geschachtelt$\rangle$
    put_list X[n]
    $\langle$Ablegen von $t_1\rangle$
    $\langle$Ablegen von $t_2\rangle$

Das Ablegen der Termwurzeln bereits bereitgestellter Terme wird durch die beiden folgenden Schemata beschrieben:

$\langle$Ablegen von $t = f(t_1, \ldots, t_k)\rangle =$
    set_value X[n]

$\langle$Ablegen von $t = [t_1|t_2]\rangle =$
    set_value X[n]

Die oben genannten Register X[n] sind die, in denen die Wurzeln der nicht atomaren Teilterme abgelegt wurden.

Die entsprechenden Instruktionen für das Ablegen von Variablen können Tabelle 4.5 entnommen werden. Der noch fehlende Fall für das direkte Ablegen einer Konstanten schlägt sich in dem folgenden Schema nieder:

$\langle$Ablegen von $t =$ C, C Konstante$\rangle =$
    set_constant C

4.3.3.2 Optimierungen

Nachdem nun das Vorgehen zur Codeerzeugung bekannt ist, werden Optimierungen vorstellen, die der Übersetzer des KAIN–PROLOG Systems vornimmt.

Tabelle 4.5: Instruktionen für Bereitstellung einer geschachtelten Variable

Auftreten	Quantifizierung	Offset	Instruktion	
Einziges	∃	−	`set_void`	
		+	`set_void_offset`	`0`
	∀	−	`set_void_all`	
		+	`set_void_all_offset`	`0`
Erstes	∃	−	`set_variable`	`V`[n]
		+	`set_variable_offset`	`V`[n]`, 0`
	∀	−	`set_variable_all`	`V`[n]
		+	`set_variable_all_offset`	`V`[n]`, 0`
Weiteres	egal	egal	`set_value`	`V`[n]
Letztes unsicheres	egal	egal	`set_local_value`	`V`[n]

Eliminierung des letzten Prozeduraufrufs Die Eliminierung des letzten Prozeduraufrufs — auch als „Last Call Optimization (LCO)“ bezeichnet — wird dann vorgenommen, wenn auf den letzten Prozeduraufruf `call N,` q/k einer Klausel nur noch die Instruktionen `deallocate` und `proceed` folgen. Diese drei Instruktionen werden durch die Sequenz `deallocate` und `execute` q/k ersetzt. Das bedeutet, daß die Prozedurumgebung vor dem letzten Aufruf zerstört wird, und ein direkter Sprung auf die Prozedur q/k folgt. Man beachte, daß diese Optimierung durch das Auftreten der Instruktionen `dequantify` und `deactivate N` nach der letzten `call`–Instruktion verhindert wird: Hier darf die Prozedurumgebung noch nicht zerstört werden, da ein Rücksprung in die Klausel nach dem letzten Prozeduraufruf nötig ist, um den Schachtelungsindex zu vermindern, beziehungsweise um Implikationspunkte abzubauen.

Zusammenfassen der Unifikation des Kopfatoms mit dem ersten Rumpfatom Hierdurch wird erreicht, daß überflüssige Kopieraktionen nicht stattfinden. Die dazu nötige Information wird bei der Registerallokation schon berechnet: Bei der Unifikation der Kopfterme werden die Termwurzeln vorzugsweise in den Registern abgelegt, in denen diese bei Aufruf des ersten Rumpfatoms benötigt werden.

Ausführungsreihenfolge Durch eine geschickte Festlegung der Ausführungsreihenfolge kann der Code stark verbessert werden. Es steht ja nicht immer fest, daß der Code einer Klausel bis zur letzten Instruktion durchlaufen wird: Vorher kann Rücksetzen ausgelöst worden sein. Der Compiler trägt diesem Aspekt dadurch Rechnung, daß er gewisse Instruktionen soweit verschiebt, bis deren Ausführung zwingend notwendig wird. Welche Instruktionen davon in welcher Form betroffen sind, ist in Tabelle 4.6 zu sehen.

Eliminierung von Instruktionen Als eine weitere Optimierung werden direkt aufeinanderfolgende Paare von `dequantify` und `quantify` Instruktionen unterdrückt.

Tabelle 4.6: Verschieben von Instruktionen

Instruktionen	Verschieben möglich bis
`allocate`	Erster Zugriff auf permanente Variable, erster Prozeduraufruf oder erstes Anlegen von Implikationspunkten
`put_shared`	Erster Prozeduraufruf in der Konklusion der hypothetischen Implikation
`get_shared`	Erster Prozeduraufruf in dem Rumpf der dynamischen Klausel

4.3.4 Erzeugung von Indexcode

Bei der Erzeugung von Indexcode wird jetzt berücksichtigt, daß ein Prozeduraufruf mit einer allquantifizierten Variable im ersten Argument stattfinden kann. Dazu hatten wir in Abschnitt 3.6 die `switch_on_ext_type`–Instruktion eingeführt. Diese Instruktion verzweigt je nach dem Objekttyp des ersten Argumentes zu einer anderen Adresse. Der Code, der für die bisherigen Argumenttypen erzeugt wird, bleibt ungeändert. Die Instruktionssequenz, zu der verzweigt wird, wenn das erste Argument eine allquantifizierte Variable ist, ist auch ganz einfach aufgebaut. Hier werden alle Klauseln sequentiell mit Hilfe der normalen Indexinstruktionen verknüpft, die im ersten Argument des Kopfatoms eine freie Variable haben: Nur für diese kann eine Unifikation des Aufrufes mit dem Klauselkopf überhaupt erfolgreich sein.

Zusätzlich muß jetzt bei der Erzeugung des Indexcodes feststehen, ob es nur statische, nur dynamische oder beide Arten von Klauseln für diese Prozedur gibt. Im ersten Fall wird eine `try N`, im zweiten Fall eine `trust_dynamic N` und im letzten Fall eine `try_dynamic N` Instruktion ausgewählt, mit der dann der Indexcode beginnt. Diese Auswahl richtet sich nach der Deklaration einer Klausel, hierfür sei auf den nächsten Abschnitt verwiesen.

4.4 Modulverwaltung

Das Modulkonzept von KA–PROLOG ist in das KAIN–PROLOG System fast unverändert übernommen worden. Es gibt zwei kleine Modifikationen, diese beiden werden hier vorgestellt.

Bei KA–PROLOG war es so, daß es zu einem Prädikat nur in einem Modul implementierende Klauseln geben durfte. Mit anderen Worten, zu einem aus einem anderen Modul importierten Prädikat durften keine weiteren Klauseln angegeben werden. Bei dem KAIN–PROLOG System ist das dahingehend aufgehoben, daß zu einem importierten Prädikat noch dynamische Klauseln definiert werden dürfen, das heißt also Klauseln, die in der Prämisse einer hypothetischen Implikation auftreten. Um sicherzustellen, daß diese Klauseln auch ausgeführt werden können, muß der Indexcode, der die statischen Klauseln untereinander verbindet mit einer `try_dynamic N` oder einer `trust_dynamic N` Instruktion beginnen. Das ist gerade die Situation, die im letzten Abschnitt erwähnt wurde.

Es ist jetzt möglich, auch in Anfragen an das System dynamische Klauseln auftreten zu lassen. Wünschenswert ist dabei, daß dies auch Klauseln sein können, die ein vollkommen

neues Kopfprädikat haben. Deshalb ist es möglich, auf der Anfrageebene des Systems noch explizit neue Prädikate zu deklarieren.

4.5 Der Debugger

Der Debugger dient dazu, den Benutzer des Systems bei der Abarbeitung einer Anfrage mit Information darüber zu versorgen, *was* bei der Abarbeitung gerade vor sich geht.

4.5.1 Anforderungen

In einem herkömmlichen System für das logische Programmieren sind bei der Abarbeitung nur die Anfragen einer Änderung unterworfen. Bei der Formulierung des Ableitungsbegriffes in Definition 2.12 auf Seite 17 gingen jedoch drei jeweils veränderliche Komponenten bei der Abarbeitung mit ein: Die Anfragen, das jeweilige Programm und der aktuelle Schachtelungsindex.

Der Debugger im KAIN–PROLOG System wird also den Anforderungen gerecht werden müssen, den Benutzer sowohl über die Änderung von Anfragen, als auch über Änderungen des Programms und des Schachtelungsindexes zu informieren.

Die Idee, die bei der Extraktion der Information aus der gerade stattfindenden Abarbeitung zugrundeliegt, ist, daß bei der Abarbeitung der Instruktionen diese „on the fly“ dekompiliert werden. Eine Behandlung dieser Dekompilation ist in [Bue86] zu finden. Im weiteren wollen wir uns überlegen, welche Instruktionen bei der Abarbeitung die gewünschte Information liefern können, und wie diese Information dem Benutzer verfügbar gemacht werden kann.

Die Extraktion und Darstellung von Prozeduraufrufen, die ja gerade die Änderung der Anfrage darstellen, aus der gerade bearbeiteten Instruktionsfolge wird genauso wie bei dem KA–PROLOG System gehandhabt. In den nächsten beiden Abschnitten werden wir uns mit der expliziten Quantifizierung und hypothetischen Implikationen beschäftigen.

4.5.2 Explizite Quantifizierung

Die Instruktionen, die die explizite Quantifizierung auf der Ebene der I–WAM realisieren, sind einerseits die `quantify`– und `dequantify`–Instruktionen, andererseits aber auch die `put`– und `set`–Instruktionen. Es geht jetzt darum, die Wirkung der Abarbeitung dieser Instruktionen sichtbar zu machen. Für den Benutzer ist es natürlich nicht interessant, zu erfahren, daß jetzt der Schachtelungsindex erhöht wurde oder gerade eine allquantifizierte Variable in einem Argumentregister abgelegt wurde.

Vielmehr kann die Wirkung dieser Befehle dadurch klar gemacht werden, daß bei einem Prozeduraufruf die Ausgabe der Terme, die als Argumente übergeben werden, erweitert wird: Werden diese Argumentterme ausgegeben, so werden bei existentiellen und universellen Variablen einfach die entsprechenden Schachtelungsindices mit ausgegeben. Dieses Vorgehen hat den Vorteil, alle Informationen im Zusammenhang mit der expliziten Quantifizierung darzustellen, ohne eine wahre Flut an Ausgaben vornehmen zu müssen. Man denke insbesondere daran, daß bei dieser Ausgabe auch die Anpassung der Schachtelungsindices durch die Unifikation sichtbar wird.

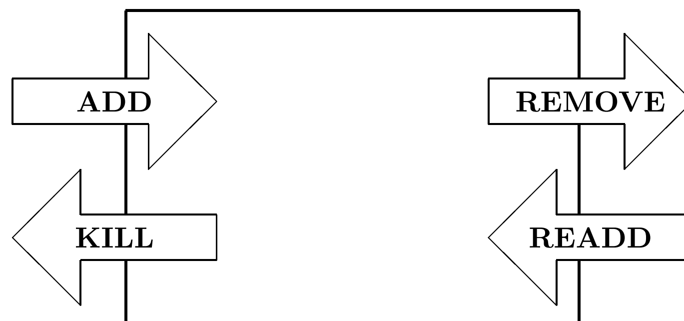


Abbildung 4.4: Vier–Tor Modell für dynamische Klauseln

4.5.3 Hypothetische Implikationen

Die Repräsentation der Information, welche Klauseln das Programm gerade enthält, wird genauso vorgenommen, wie die Repräsentation der Prozeduraufrufe. Prozeduraufrufe werden durch ein „Vier–Tor“ Modell sichtbar gemacht: Man hat ein CALL–Tor für das erstmalige Betreten der Klausel, ein EXIT–Tor für das Verlassen der Klausel, ein REDO–Tor für das Wiederbetreten der Klausel beim Rücksetzen und ein FAIL–Tor, durch daß die Klausel nach dem endgültigen Fehlschlagen wieder verlassen wird.

Analog dazu legen wir der Abarbeitung von hypothetischen Implikationen auch ein Vier–Tor Modell zu Grunde. Schematisch ist dieses Vier–Tor Modell für hypothetische Implikationen in Abbildung 4.4 zu sehen.

Wir wollen jetzt dieses Vier–Tor Modell näher erläutern:

ADD Wird eine hypothetische Implikation bei der Abarbeitung betreten, so werden die Klauseln in der Prämisse dieser Implikation dem Programm hinzugefügt. Dann wird mit der Ausführung der Konklusion der Implikation fortgefahren. Die entsprechende Instruktion der I–WAM, bei deren Abarbeitung die nötige Information zur Verfügung steht, ist die `activate`–Instruktion. Der Benutzer wird darüber informiert, welche Klauseln hinzugefügt wurden.

REMOVE Nach erfolgreicher Abarbeitung der Konklusion werden die Klauseln der Prämisse entfernt, und dieses dem Benutzer mitgeteilt. Das entspricht gerade der Abarbeitung der `deactivate`–Instruktion.

READD Schlägt die Abarbeitung einer folgenden Anfrage fehl, so wird die hypothetische Implikation wieder „betreten“. Die dynamischen Klauseln werden erneut wieder hinzugefügt und Alternativen innerhalb der Konklusion der Implikation ausgewählt. Die nötige Information läßt sich bei dem Rücksetzen extrahieren, genauer bei der Wiederherstellung von Implikationspunkten durch die *LinkImpPts*–Operation, siehe dazu auch die Abschnitte 3.5.1 und A.3.2. Ausgegeben wird, welche Klauseln wieder hinzugefügt werden.

KILL Sind alle Alternativen bei der Abarbeitung der Konklusion der hypothetischen Implikation fehlgeschlagen, so schlägt auch die hypothetische Implikation fehl. Hier wird die

nötige Information auch durch die Operation *LinkImpPts* beim Rücksetzen erhalten, aber bei dem Zerstören von Implikationspunkten. Auch hier wird der Benutzer wieder über die zerstörten Klauseln informiert.

Die Ausgabe der jeweiligen Klausel bei dem Betreten der obigen Tore beschränkt sich bei dem KAIN–PROLOG System darauf, welches Prädikatensymbol der Kopf der Klausel hat. Denn ein explizites Ausgeben der Definition der Klausel ist ohne weiteres nicht möglich, da dazu eine komplette Dekompilation der ganzen Klausel nötig ist. Prinzipiell ist dies möglich, würde jedoch umfangreiche Erweiterungen des Systems mit sich bringen.

Kapitel 5

Praktikabilität des KAIN–PROLOG Systems

In diesem Kapitel geht es darum, die Praktikabilität des in dieser Arbeit entworfenen und implementierten KAIN–PROLOG Systems zu untersuchen. Dabei werden zwei Hauptaspekte berücksichtigt. Der eine ist, wie sich das implementierte System hinsichtlich Laufzeit und Speicherbedarf verhält. Der andere Aspekt betrifft die eigentliche *Praktikabilität*: Es werden Beispielprogramme angegeben und untersucht, die Paradigmen des intuitionistischen logischen Programmierens aufgreifen. Es wird gezeigt, daß es viele interessante, allgemein einsetzbare Paradigmen gibt und daß ihr Einsatz entscheidende Vorteile hinsichtlich Effizienz oder Verstehbarkeit mit sich bringt. Salopp gesagt, steht die folgende Frage im Vordergrund: „Bringt es mehr ein, als es kostet?“

5.1 Speicherbedarf

Der Speicherbedarf von KAIN–PROLOG ist leicht zu überschauen. Die Datenobjekte enthalten mehr Information, in der Implementierung belegen sie 8 Byte, im Gegensatz zu 4 Byte bei KA–PROLOG. Das heißt, der Speicherbedarf auf der Halde und für die Datenregister ist doppelt so groß.

Die Prozedurumgebungen auf dem Kontrollstapel, die auch Datenobjekte enthalten, nehmen dadurch an Umfang in diesem Maße zu. Die Wahlpunkte werden durch Ablegen zweier neuer Register[1] und der zusätzlichen dynamischen Alternative auch etwas größer. Bei KA–PROLOG werden sieben Register in einem Wahlpunkt abgespeichert, bei KAIN–PROLOG sind es zehn[2]. Außerdem ist zu berücksichtigen, daß sich auf dem Kontrollstapel auch noch Implikationspunkte befinden.

Bei dem Spurstapel ist die Situation nicht ganz so transparent. Prinzipiell sind die dort abgelegten Daten jetzt auch doppelt so groß wie bei KA–PROLOG, da ganze Variablenobjekte anstatt von Adressen abgelegt werden. In Beispiel 3.1 auf Seite 25, wo die Unifikation von Termen auf der Ebene der I–WAM dargestellt wurde, werden Variablen *mehrmals* auf den

[1] Das sind die Register *NL* für den aktuellen Schachtelungsindex und *IP* für den letzten Implikationspunkt.

[2] Eine Zusammenstellung aller Register läßt sich in der Tabelle A.1 auf Seite 88 finden.

Spurstapel gesichert: Zum einen wegen der Veränderung des Schachtelungsindexes, zum anderen wegen des Überschreibens von Variablen. Also müssen wir uns überlegen, wie oft Variablen durch die *Trail*–Operation auf dem Spurstapel abgelegt werden. Diese Überlegung faßt das nächste Lemma zusammen.

Lemma 5.1 *Seien t_1, t_2 Terme mit $k = \#(\mathrm{var}(t_1) \cup \mathrm{var}(t_2))$ Variablen und $l = \#\{n \mid x^n \in (\mathrm{var}(t_1) \cup \mathrm{var}(t_2))\}$ verschiedenen Schachtelungsindices. Für die Anzahl der Trail–Operationen zur Anpassung der Schachtelungsindices T_{Nest} bei der Unifikation von t_1 und t_2 gilt:*

$$T_{Nest} = l - 1. \tag{5.1}$$

Weiter gilt für die Gesamtanzahl der Trail–Operationen T bei der Unifikation von t_1 und t_2:

$$T \leq 2k - 1. \tag{5.2}$$

Beweis Sei o.B.d.A. $k \neq 0$, also auch $l \neq 0$. Der Beweis für (5.1) wird mit Induktion über l geführt.

Ist $l = 1$, so besitzen alle Variablen den gleichen Schachtelungsindex; das heißt, es ist kein Sichern nötig, und es gilt: $T_{Nest} = 0$. Für alle l' mit $l' < l$ gelte (5.1). Sei l die Anzahl der verschiedenen Schachtelungsindices. Wird die Variable x^n an einen Term t mit $p = \#\{m \mid x^m \in \mathrm{var}(t), m > n\}$ gebunden, so werden p Schachtelungsindices angepaßt. Die Anzahl l' der noch verschiedenen Schachtelungsindices ist $l - p$. Nach Induktionsvorraussetzung sind noch $l' - 1 = l - p - 1$ *Trail*–Operationen nötig, und es wurden schon p Operationen durchgeführt. Wegen $(l - p - 1) + p = l - 1$ gilt (5.1).

Die Aussage (5.2) gilt, da l maximal so groß wie k sein kann, und jede Variable höchstens einmal an einen Term gebunden wird.

□

Somit ist auch der Speicherbedarf des Spurstapels unkritisch. Bei gleichzeitiger Betrachtung aller freien Variablen während der Abarbeitung eines Programms kann sich die Anzahl der *Trail*–Operationen, somit die der belegten Zellen des Spurstapels, im Vergleich zu KA–PROLOG maximal verdoppeln.

Als Fazit läßt sich angeben, daß der Speicherbedarf von KAIN–PROLOG in etwa doppelt so groß ist wie der von KA–PROLOG. Bei den heute zur Verfügung stehenden Speichergrößen ist dies ein durchaus akzeptabler Faktor.

5.2 Laufzeitverhalten

Dieser Abschnitt beschäftigt sich damit, das Laufzeitverhalten des KAIN–PROLOG Systems mit dem des KA–PROLOG Systems zu vergleichen. Die beiden Systeme werden jetzt kurz als KAIN beziehungsweise KAP bezeichnet.

Die zur Untersuchung herangezogenen Programme sind einerseits Benchmarks, die es gestatten, spezielle Aspekte der beiden Systeme miteinander zu vergleichen. Es wird analysiert, wie die Unterschiede im Laufzeitverhalten zu Stande kommen: Das bietet eine Grundlage zur

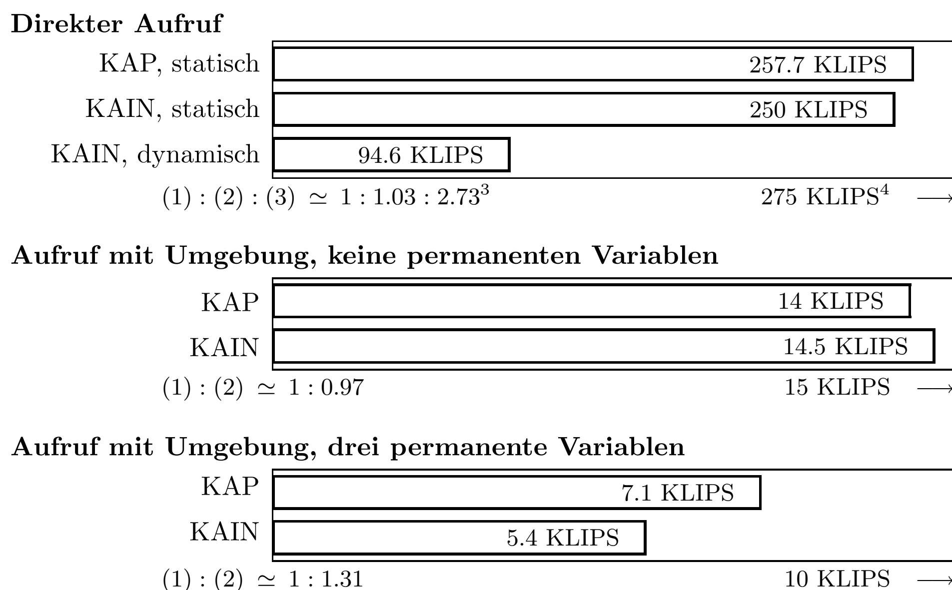


Abbildung 5.1: Laufzeitverhalten von Prozeduraufrufen

Bewertung des KAIN Systems und zeigt Punkte auf, an denen das System der Verbesserung bedarf. Auch Aspekte, die aufgrund der größeren Funktionalität des KAIN Systems keine Entsprechung im KAP System haben, werden untersucht.

Andererseits wird auch angegeben, was für eine Effizienzeinbuße bei Benutzung des KAIN Systems für normale PROLOG–Programme gegenüber dem KAP System *durchschnittlich* zu erwarten ist.

5.2.1 Prozeduraufrufe

Zuerst wird untersucht, welche Unterschiede sich bei der mit Abstand häufigsten Operation, dem Prozeduraufruf, ergeben. Die gemessenen Werte für diesen Abschnitt sind alle in Abbildung 5.1 dargestellt.

Direkter Aufruf Hier wird die Abarbeitungsgeschwindigkeit normaler Prozeduraufrufe gemessen: Klauseln, die nur aus jeweils einem nullstelligen Kopf- und Rumpfprädikat bestehen, werden aufgerufen. Bei „KAP, statisch“ und „KAIN, statisch“ sind dieses normale Klauseln, bei „KAIN, dynamisch“ sind das dynamische Klauseln, die mittels einer hypothetischen Implikation aktiviert wurden. Das Aktivieren der Klauseln wurde nicht mitgemessen.

Wie zu erwarten, ist so gut wie kein Unterschied zwischen den beiden statischen Varianten von KAP und KAIN festzustellen: Denn hier werden beidesmal dieselben Operationen, die ja

[3]Bei den angegebenen Verhältnissen bedeutet eine Zahl, die größer als eins ist, daß das entsprechende Programm um diesen Faktor *langsamer* abläuft.

[4]KLIPS steht für tausend Inferenzschritte pro Sekunde, englisch kilo logical inferences per second. Alle KLIPS–Angaben beziehen sich auf Messungen an einer Sun SPARCstation 1 unter SunOS 4.1.2.

keine Erweiterung erfahren haben, ausgeführt. Die dynamische Variante von KAIN ist um den Faktor 2.73 langsamer. Dieser Faktor beschreibt, was die Benutzung dynamischer Klauseln kostet. Dies liegt daran, daß hier der Zugriff auf die Klausel über mehrere Indirektionsstufen geht. Bei beiden Varianten wird zuerst auf den Prozedurbeschreibungsblock zugegriffen, dann die Startadresse des Indexcodes bestimmt. Bei der statischen Variante besteht dieser Indexcode aus einem direkten Sprung auf den eigentlichen Code der Klausel. Im dynamischen Fall besteht der Indexcode hier aus einer `trust_dynamic`–Instruktion: Bei Ihrer Abarbeitung wird erst der letzte Implikationspunkt für das anzuspringende Prädikat ermittelt, der Implikationspunktumgebungszeiger gesetzt, geprüft, ob ein Wahlpunkt angelegt werden muß, und erst dann wird zu dem Klauselcode gesprungen, dessen Adresse aus dem Implikationspunkt extrahiert wurde. Angesichts dieser komplexen Operation mit Zugriff über mehrere Indirektionsstufen ist der gemessene Faktor eher überraschend klein.

Aufruf mit Umgebung Hier werden wir das Laufzeitverhalten untersuchen, wenn Prozeduren aufgerufen werden, die mehr als ein Rumpfatom besitzen. In diesem Fall müssen die Rücksprung- und Umgebungsadresse mittels einer `allocate`–Instruktion gesichert und einer `deallocate`–Instruktion restauriert werden.

Wir betrachten sowohl Klauseln mit nullstelligen als auch dreistelligen Kopf- und Rumpfprädikaten. Für die nullstelligen Prädikate stimmen die gemessenen Werte überein, da auch hier keine Änderung der zugrundeliegenden Instruktionen stattgefunden hat. Anders dagegen bei den dreistelligen Prozeduraufrufen: In dem gemessenen Programm werden vor jedem Prozeduraufruf noch Variablen in die Argumentregister geschrieben. Die Datenobjekte bei KAIN sind doppelt so groß wie bei KAP, ihr Kopieren in die Argumentregister beansprucht entsprechend mehr Zeit. In Verbindung mit dem gleich gebliebenen Aufwand für die Abwicklung des eigentlichen Aufrufs resultiert dann der Faktor 1.31.

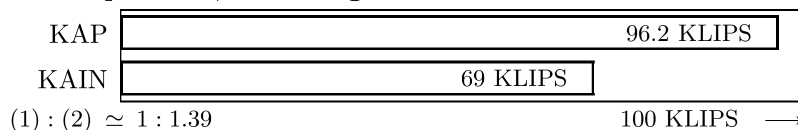
5.2.2 Verwaltung der Wahlpunkte

In diesem Abschnitt wollen wir das Anlegen von Wahlpunkten und deren Verwaltung durch das Rücksetzen untersuchen. Die gemessenen Werte sind der Abbildung 5.2 zu entnehmen.

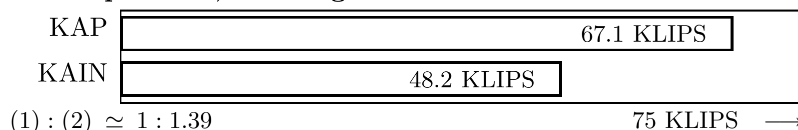
Anlegen von Wahlpunkten Grundlage der Untersuchung bildet ein Programm, das aus mehreren Prozeduren besteht, die jeweils zwei Klauseln umfassen. Bei Ausführung dieser Prozeduren wird dann für jeden Aufruf ein Wahlpunkt auf dem Kontrollstapel angelegt. Es werden sowohl Prozeduraufrufe für nullstellige als auch für dreistellige Klauseln untersucht: Letzteres macht das zusätzliche Abspeichern von drei Argumentregistern beim Anlegen eines Wahlpunktes nötig.

Beide Messungen liefern denselben Faktor 1.39, um den KAIN langsamer als KAP ist. In der Diskussion des Speicherbedarfs wurde erwähnt, daß die Wahlpunkte im Verhältnis 7 zu 10 von KAP zu KAIN größer sind. Das Verhältnis 7 : 10 entspricht in etwa dem Verhältnis 1 : 1.43. Das „mehr“ an abzulegender Information ist also ziemlich genau das „mehr“ an Laufzeit. Zu bemerken ist außerdem, daß das Speichern der Argumentregister nicht meßbar in Erscheinung tritt, obwohl die in den Argumentregistern zu speichernde Information jetzt doppelt so groß ist. Erklären läßt sich dieses Phänomen dadurch, daß der Aufwand beim Ablegen der anderen Register und bei den Prozeduraufrufen dieses verdeckt.

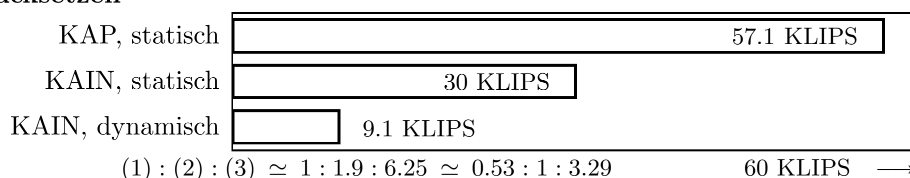
Anlegen von Wahlpunkten, keine Argumente



Anlegen von Wahlpunkten, drei Argumente



Rücksetzen



Indizierte Klauselauswahl

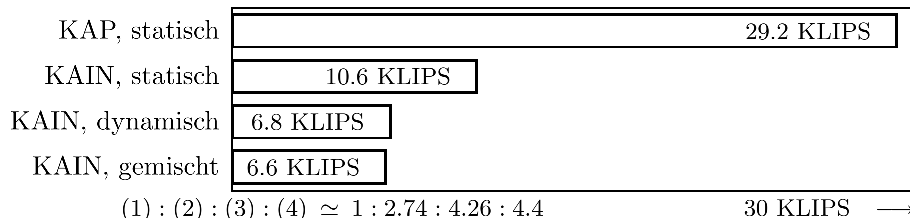


Abbildung 5.2: Laufzeitverhalten der Wahlpunktverwaltung

Rücksetzen Hier untersuchen wir, wie sich das Verhältnis von KAP zu KAIN bei dem Rücksetzen verhält. Das Problem wird durch ein Programm modelliert, das 20 Klauseln mit dreistelligem Kopfprädikat enthält, die bei einem Aufruf fehlschlagen, und eine, die erfolgreich ist. Bei der dynamischen Variante von KAIN sind diese Klauseln wieder bei der Abarbeitung einer hypothetischen Implikation hinzugefügt worden, wobei auch hier das Anlegen der Implikationspunkte nicht mitgemessen wurde.

Vergleicht man die beiden statischen Varianten, so ergibt sich ein Faktor von 1.9, um den KAIN in der Ausführung langsamer ist. Dieser Faktor läßt sich folgendermaßen erklären: Einmal ist das Anlegen von Wahlpunkten, wie oben schon diskutiert, langsamer. Denn auch diese Operation schlägt sich in den Messungen nieder. Zusätzlich ist das Restaurieren des Maschinenzustandes wesentlich komplexer. Neben den Operationen, die die zusätzlichen Daten aus dem Wahlpunkt entnehmen, werden jetzt die auf dem Spurstapel gespeicherten Variablenbindungen gelöst, und es muß überprüft werden, ob eine statische oder dynamische Alternative vorliegt, und ob Implikationspunkte zu verwalten sind. Das Lösen der Variablenbindungen verbraucht jetzt mehr Zeit, da wiederum Objekte kopiert werden müssen, die doppelt so groß sind.

Dem Vergleich der Laufzeiten von der statischen Variante für KAP und der dynamischen

für KAIN wollen wir uns jetzt zuwenden. Auf den ersten Blick wirkt der sich ergebende Faktor von 6.25 sehr groß. Legt man jedoch zuerst den oben ermittelten Faktor zugrunde, da auch die dort beschriebenen Operationen hier mit eingehen, ist der verbleibende Faktor von 3.29 leicht zu erklären. Einerseits fallen die mehreren Indirektionsstufen bei Prozeduraufrufen für dynamische Klauseln mit ins Gewicht. Andererseits wird das Ablaufen dieser Indirektionsstufen bei *jedem* Rücksetzen wieder nötig: Die entsprechende Information wird auch bei einem Rücksetzvorgang aus den jeweiligen Implikationspunkten extrahiert. Somit ist der Faktor von 6.25 zwar unerfreulich hoch, aber doch nachvollziehbar.

In einem späteren Beispiel in Abschnitt 5.3.2 wird ein Programm untersucht, das extensiven Gebrauch vom Rücksetzen macht. Dort wird gemessen, wie der zusätzliche Aufwand in einem realistischen Beispiel ausfällt.

Indizierte Klauselauswahl Der indizierten Klauselauswahl liegt ein Programm zugrunde, das 21 Klauseln für dasselbe Prädikat umfaßt, wobei jeweils abwechselnd das erste Argument der Klauseln eine Konstante, eine Liste oder ein Funktor ist. Gemessen wurde, wie lange die Abarbeitung einer Klausel braucht, die alle 21 Klauseln in ihrem Rumpf durch Prozeduraufruf auswählt. Bei den statischen Varianten handelt es sich um 21 statische Klauseln. Bei der dynamischen Variante sind alle 21 Klauseln dynamisch, und für jede Klausel wurde ein Implikationspunkt angelegt. Bei der gemischten Variante sind auch alle Klauseln dynamisch, aber für jeweils sieben wurde nur ein Implikationspunkt angelegt. Dies entspricht gerade der in Abschnitt 3.5.3 diskutierten Alternative.

Der Faktor von 2.74, der sich zwischen den Varianten „KAP, statisch“ und „KAIN, statisch“ ergibt, liegt wiederum an der komplizierteren Wahlpunktverwaltung. Daß dieser Faktor höher ausfällt als oben, liegt daran, daß das Rücksetzen hier eine viel größere Auswahl zwischen Klauseln treffen muß.

Werden dagegen dynamische Klauseln ausgewählt, so ergibt sich ein Faktor von 4.26. Das liegt daran, daß nach den Klauseln mit passenden ersten Argumenten sequentiell gesucht wird. Sie werden also nicht per Direktzugriff wie im statischen Fall ausgewählt. Dieser Faktor ist überraschend gering, wenn man die beiden Zugriffsmethoden berücksichtigt.

In Abschnitt 3.5.3 hatten wir eine Alternative diskutiert, wie dynamische Klauseln bereitgestellt werden können. Hier zeigt sich durch den resultierenden Faktor von 4.4, daß es die Effizienz eher herabsetzt, wenn man mehrere dynamische Klauseln für dasselbe Prädikat zusammenfaßt. In der Regel wird man auf einmal sowieso eher wenige Klauseln hinzufügen, die betrachtete Anzahl von sieben Klauseln wird wohl selten überschritten.

5.2.3 Unifikation

In diesem Abschnitt wird gemessen, wie das Laufzeitverhalten bei der Unifikation ist. Die Ergebnisse der Messungen sind in Abbildung 5.3 dargestellt. Dabei gibt es jedoch noch einen Punkt zu berücksichtigen, der nicht in die Messungen mit eingeht. Bei normalen PROLOG Systemen fehlt der Occur–Check, oder er läßt sich abschalten. Das ist auch in dem KA–PROLOG und KAIN–PROLOG System so. Bei der normalen Unifikation treten jedoch nur sehr selten Situationen auf, in denen der Occur–Check wirklich benötigt wird. Bei Quantoren treten jedoch sehr schnell Situationen auf, in denen der Occur–Check die Inkorrektheit von Unifikatoren verhindert. Man denke dabei nur an das Beispiel 3.4 auf Seite 33. Generell muß

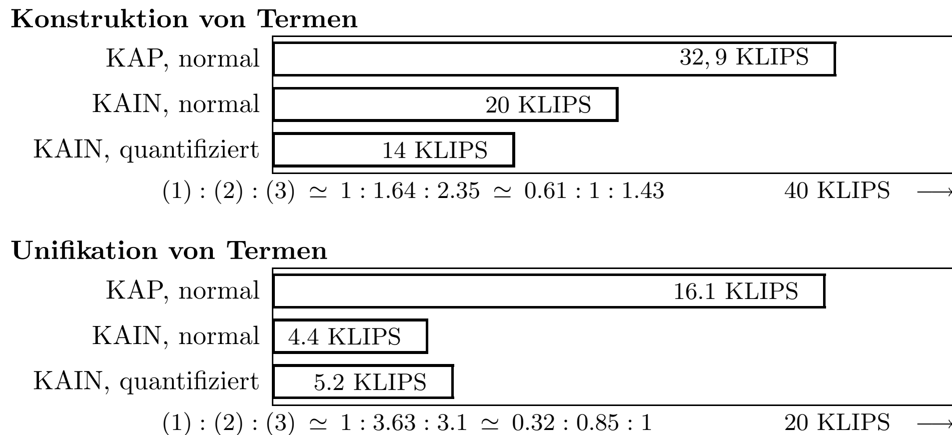


Abbildung 5.3: Laufzeitverhalten der Unifikation

also berücksichtigt werden, daß der Occur–Check nicht so ohne weiteres abgeschaltet werden kann. Bei den folgenden Messungen war jeweils der Occur–Check eingeschaltet.

Konstruktion von Termen Hier wird gemessen, wie sich die Bereitstellung von Argumenten verhält. Dazu wird ein Programm herangezogen, das bei jedem Prozeduraufruf Terme aller möglichen Bauarten konstruiert. Unter „normal“ verstehen wir im Gegensatz zu „quantifiziert“ hier die Konstruktion nicht explizit quantifizierter Terme.

Der Faktor von 1.64 zwischen der „normalen“ Konstruktion bei KAP, beziehungsweise KAIN, läßt sich einfach auf die größeren Datenobjekte zurückführen.

Der zusätzliche Faktor von 1.43 bei der Konstruktion quantifizierter Terme kommt dadurch zu Stande, daß durch das Auftreten von `dequantify`–Instruktionen keine Eliminierung des letzten Prozeduraufrufes möglich ist, siehe dazu auch Abschnitt 4.3.3.2.

Unifikation von Termen Hier wollen wir untersuchen, wie sich die Laufzeit bei der eigentlichen Unifikation verhält. Es wird die Unifikation verschachtelter Terme betrachtet. Bei der Variante „quantifiziert“ werden Terme mit universell und existentiell quantifizierten Variablen unifiziert, bei der Variante „normal“ sind die allquantifizierten Variablen durch normale Konstanten ersetzt.

Der Faktor von 3.61 zwischen den beiden normalen Varianten beschreibt jetzt den Mehraufwand bei der Unifikation. Dieser Mehraufwand tritt wegen der folgenden Punkte auf: Das Sichern von überschriebenen Variablen ist teurer, da ganze Objekte unter Umständen auch mehrmals auf dem Spurstapel abgelegt werden. Zusätzlich fällt auch das „mehr“ an zu verwaltender Information in den freien Variablen an. Das der Faktor so hoch ausfällt, liegt auch an der Architektur des zugrundeliegenden Rechners: Die Datenobjekte befinden sich nicht mehr in Registern, sondern im Speicher, da die Objekte für die Register zu groß sind.

Der Faktor von 0.85, um den die Unifikation quantifizierter Terme zu der Unifikation normaler Terme schneller ist, resultiert aus der konkreten Implementierung des Vergleichs von

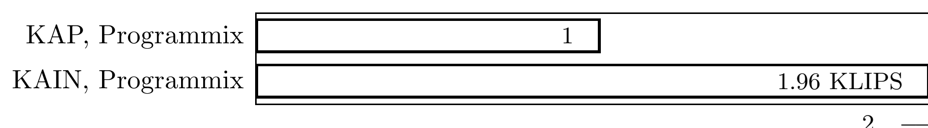


Abbildung 5.4: Durchschnittliches Laufzeitverhalten

Konstanten. Zum Vergleich von Konstanten wird intern ein Prozeduraufruf durchgeführt, der Vergleich von allquantifizierten Variablen untereinander wird direkt ausgeführt. Man sieht also, daß die hier ermittelten Werte mit gewisser Vorsicht zu behandeln sind, da auch eher sekundäre Effekte mit eingehen.

5.2.4 Programmix

Abschließend wollen wir noch untersuchen, wie sich der Unterschied im Laufzeitverhalten bei einem repräsentativen Programmix von PROLOG–Programmen auswirkt. In Abbildung 5.4 ist die Relation der Laufzeiten eines Programmixes von 8 typischen Programmen dargestellt. Im Gegensatz zu den vorherigen Messungen sind hier keine KLIPS–Zahlen aufgeführt, sondern nur die Verhältnisse der Laufzeiten, wobei der kürzesten Laufzeit die Zahl 1 zugeordnet wurde. Den Faktor von 1.96 kann man als den zugrundelegen, der generell bei dem KAIN System zu Buche schlägt. Bei der enormen Evolution der Rechnerleistung ist dieser Faktor durchaus vertretbar.

5.3 Praktikabilität

Bisher haben wir in diesem Kapitel nur diskutiert, was an zusätzlichem Aufwand für KAIN–PROLOG zu erwarten ist. In diesem Abschnitt wenden wir uns einem anderen Aspekt zu. Wir geben Beispielprogramme an, die bestimmte Paradigmen des intuitionistischen logischen Programmierens herausstellen. Wir werden zeigen, daß diese Paradigmen einerseits effizient von KAIN–PROLOG unterstützt werden, und daß ihr Einsatz andererseits einen Gewinn an Ausdruckskraft, Verstehbarkeit und Effizienz mit sich bringt. Wir werden also versuchen, die am Anfang des Kapitels gestellte Frage so zu beantworten: „Es bringt *mehr* ein, als es kostet!“

Die betrachteten Beispielprogramme werden hier nur kurz umrissen; sie sind jedoch in Anhang C aufgeführt. Dieser Anhang bietet zudem auch einen guten Überblick, wie Programme beziehungsweise Module in KAIN–PROLOG aussehen.

5.3.1 Sortieren

In diesem Abschnitt wollen wir ein Modul entwerfen, das verschiedene Sortierprädikate für Listen anbietet. Zum einen wollen wir Prädikate haben, die sowohl auf- als auch absteigend nach der normalen Termordnung sortieren, zum anderen Prädikate, die zusätzlich noch Duplikate bei dem Sortieren aus der Liste entfernen. Wir brauchen also ein Modul, das vier verschiedene Prädikate anbietet. Es soll der Quicksort–Algorithmus [SS86, page 56] verwendet werden.

Betrachten wir jetzt zuerst eine Realisierung in normalem PROLOG. Für die beiden Sortierprädikate, die die Duplikate in den Listen belassen, brauchen wir jeweils drei Klauseln für das Partitionieren einer Liste und zwei Klauseln für das eigentliche Sortieren. Um die Prädikate zu formulieren, die Duplikate beim Sortieren entfernen, brauchen wir noch jeweils eine zusätzliche Klausel, die bei der Partitionierung der Listen dafür sorgt, daß keine Duplikate in den partitionierten Listen auftreten. Insgesamt müssen wir also 22 Klauseln implementieren. Jeder Programmierer in PROLOG wird sich schon mit Recht darüber geärgert haben, daß man für viele ähnliche Aufgaben jeweils *komplett* neue Prädikate schreiben muß.

Jetzt entwickeln wir ein Modul in KAIN–PROLOG, das Prädikate mit derselben Semantik implementiert, jedoch dafür nur 15 Klauseln benötigt, wobei dynamische Klauseln sogar schon mitgezählt sind; zählt man diese nicht mit, so sind es sogar nur elf Klauseln.

Die Idee hierzu ist einfach: Die Prädikate, die entweder auf- oder absteigend sortieren, unterscheiden sich nur durch die bei der Partitionierung benutze Relation. Bei dem Schreiben des Partitionierungs–Prädikates setzen wir ein Prädikat `relation(X,Y)` zur Beschreibung der Ordnungsrelation ein. Ist `generic_sort(Xs,SXs)` das Prädikat, das die Liste `Xs` in die Liste `SXs` sortiert, formulieren wir das Prädikat `sort_term_up` für das aufsteigende Sortieren jetzt so:

```
sort_term_up(Xs, SXs) :-
    all (
        relation(X, Y) :-
            X @=< Y
    )
    -:  generic_sort(Xs, SXs).
```

Analog dazu formulieren wir das Prädikat `sort_term_down` für das absteigende Sortieren:

```
sort_term_down(Xs, SXs) :-
    all (
        relation(X, Y) :-
            X @>= Y
    )
    -:  generic_sort(Xs, SXs).
```

Wird jetzt das Prädikat `sort_term_up` oder `sort_term_down` aufgerufen, wird zuerst eine dynamische Klausel dem Programm hinzugefügt, die beschreibt, nach welcher Relation sortiert werden soll.

Nun fehlen noch die entsprechenden Prädikate für das Sortieren mit Entfernen von Duplikaten. Wie oben schon erwähnt, erreichen wir das Entfernen von Duplikaten dadurch, daß das Prädikat, welches das Partitionieren vornimmt, eine zusätzliche Klausel erhält. Diese zusätzliche Klausel sorgt dafür, daß bei Identität eines Elements aus der zu partitionierenden Liste mit dem Element, nach dem partitioniert wird, in keine der Ergebnislisten übernommen wird. Nun formulieren wir das Prädikat `generic_sort_unique` so:

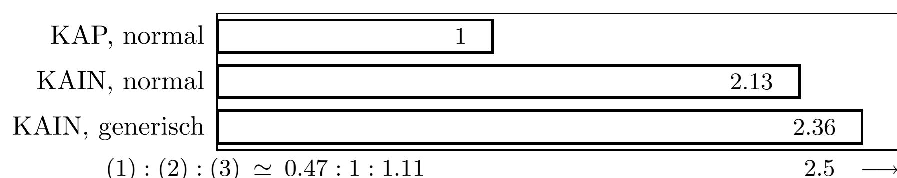


Abbildung 5.5: Relative Laufzeiten der Sortier–Prädikate

```
generic_sort_unique(Xs, SXs) :-
    all (
        partition([E | Xs], X, Ls, Gs) :-
            E == X, !,
            partition(Xs, X, Ls, Gs)
    )
    -:  generic_sort(Xs, SXs).
```

Mit Hilfe dieses Prädikats können die Prädikate, die das eigentliche Sortieren realisieren, wieder wie oben durch „Instantiieren“ der zum Sortieren herangezogenen Relation formuliert werden. Ein komplettes Modul, das die hier formulierten Prädikate enthält, ist in Abschnitt C.1 zu finden.

Dieses Beispiel dokumentiert zwei mächtige Paradigmen: *Generizität* und *Vererbung*. Die Generizität drückt sich darin aus, daß Prädikate mit „impliziten Parametern“ formuliert werden können. Die Relation, nach der in dem obigen Beispiel sortiert wird, ist so ein „impliziter Parameter“. Die Vererbung kann man so verstehen: Ein Prädikat kann durch zusätzliche Klauseln erweitert werden, es müssen nicht immer alle Klauseln eines Prädikates neu programmiert werden.

Diese Paradigmen sind vor allen Dingen im Kontext erwähnenswert, wenn das logische Programmieren zur Erzeugung von Prototypen eingesetzt wird. Eine drastische Verringerung der Entwicklungszeit läßt sich erreichen.

Daß diese Paradigmen interessant und mächtig sind, ist ohne weiteres klar. Zu Untersuchen bleibt jetzt natürlich, wie der Einsatz dieser Paradigmen sich auf den Laufzeitbedarf auswirkt. In Abbildung 5.5 sind die Ergebnisse von Messungen dargestellt. Es wurden Listen mit 100 Elementen mit allen vier Sortierprädikaten sortiert, und das arithmetische Mittel über die Laufzeiten gebildet. Die Punkte „KAP, statisch“ und „KAIN, statisch“ geben die Messungen für die lange und herkömmliche Lösung wieder, der Punkt „KAIN, dynamisch“ für die unter dem Einsatz von hypothetischen Implikationen entwickelte. Der sich ergebende Faktor 2.36 von der statischen KAP–Variante zu der dynamischen KAIN–Variante ist vollkommen akzeptabel. Legt man zudem noch die statische KAIN–Variante zu Grunde, so resultiert der Einsatz dieser Paradigmen nur in einem Faktor von 1.11. Die KAIN–Maschine bietet also eine durchaus *praktikable* Unterstützung dieser Paradigmen.

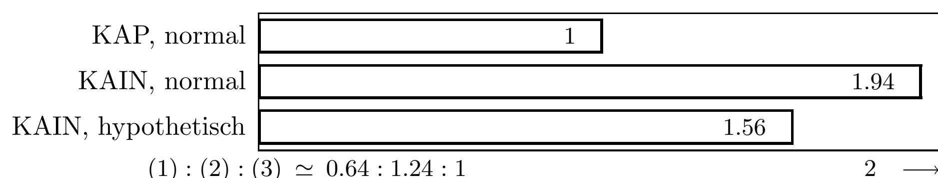


Abbildung 5.6: Relative Laufzeiten des n–Damen Problems

5.3.2 Das n–Damen Problem

Bei dem n–Damen Problem [SS86, pages 209–211] geht es darum, n Damen auf einem Schachbrett mit $n \times n$ Feldern so zu plazieren, daß diese sich nicht gegenseitig nach den üblichen Schachregeln attackieren.

Wir wollen hier eine Formulierung angeben, die rein deklarativ ist, und das Problem nach dem „Generiere und Teste" Paradigma löst. Dazu gehen wir wie folgt vor: Die `I`–te Königin wird durch das Hinzufügen des Prädikates `queen(I, Row, Col)` plaziert, und in der Konklusion der hypothetischen Implikation wird spezifiziert, welchen Anforderungen die Plazierung zu genügen hat.

Damit kommen wir zu der folgenden Programmklausel:

```
place(Queen, [q(Row, Col) | QueenList]) :-
    queen(Queen, Row, Col)            % Dame plazieren
    -:  position(Row, Col),           % Position auf dem Brett
        not attack,                   % Keine Bedrohungen
        NextQueen is Queen - 1,       % Nächste Dame...
        place(NextQueen, QueenList).  % ...plazieren
```

Durch das Prädikat `position(Row, Col)` wird spezifiziert, daß sich diese Position auf dem Brett befindet. Das Prädikat `attack` spezifiziert jetzt die Bedingung, wenn sich zwei Damen auf dem Brett bedrohen. Das Prädikat `not` hat die übliche Bedeutung der endlichen Nichtableitbarkeit, eine nähere Untersuchung hierzu im Kontext des intuitionistischen logischen Programmierens findet sich in [Gab85, BM90, Sch91b].

Die eigentliche Berechnung führt das Rücksetzen automatisch aus. Das ganze Programm läßt sich in Abschnitt C.2 finden. Man beachte dabei die außergewöhnlich elegante Formulierung. Diese Lösung ist zwar sehr ineffizient, zeigt aber sehr gut, wie hypothetische Implikationen das rein deklarative Programmieren unterstützen können.

Abschließend wollen wir wiederum die relativen Laufzeiten betrachten. In Abbildung 5.6 sind diese dargestellt. Mit „normal" sind Varianten bezeichnet, die das Problem durch das Ablegen der plazierten Königin in einer Liste lösen, wobei dann diese Liste auf Bedrohungen untersucht werden. Wenn man berücksichtigt, daß hier fast alles durch Rücksetzen berechnet wird, so ist der Faktor von 1.56 erstaunlich gering. Man beachte insbesondere, daß hier der Einsatz von hypothetischen Implikationen sowohl die Verstehbarkeit als auch die Effizienz — um den Faktor 1.24 — heraufsetzt.

5.3.3 Der Cocke–Younger–Kasami–Algorithmus

Der Cocke–Younger–Kasami– oder CYK–Algorithmus [HU90, Seite 147 ff.] dient zur Entscheidung des Elementproblems für kontextfreie Sprachen: Ist $x \in L(G)$, wobei G eine kontextfreie Grammatik ist? Dieser Algorithmus basiert auf der Technik des dynamischen Programmierens, das heißt, bei der Berechnung werden Lösungen aus vorher berechneten Teillösungen von unten nach oben zusammengesetzt.

Sei eine kontextfreie Grammatik $G = \langle T, N, P, S \rangle$ mit Terminalsymbolen T, Nichtterminalsymbolen N, Produktionen P und Startsymbol S in Chomsky–Normalform[5] gegeben. Um festzustellen, ob das Wort x der Länge k in $L(G)$ ist, wird wie folgt vorgegangen: Zuerst werden für alle Teilwörter der Länge eins von x bestimmt, aus welchen Nichtterminalsymbolen sich diese herleiten lassen. Danach werden alle Teilwörter der Längen $i = 2, \ldots, k$ betrachtet. Ein Teilwort y der Länge i ist aus dem Nichtterminalsymbol A genau dann herleitbar, wenn es eine Produktion $A \rightarrow BC \in P$ gibt, so daß $B \overset{*}{\Rightarrow} y_1$ und $C \overset{*}{\Rightarrow} y_2$ gilt, wobei $y = y_1 y_2$ und $y_1 \neq \epsilon, y_2 \neq \epsilon$ ist. Für das Teilwort y werden auf diese Art und Weise die Nichtterminalsymbole berechnet, aus denen sich y herleiten läßt. Dabei wird die Information benutzt, die vorher für die kürzeren Teilwörter schon berechnet wurde. Das Wort x ist nun in $L(G)$ enthalten, wenn das Startsymbol S in dieser Art und Weise für x selbst berechnet wurde.

Diese Formulierung wird jetzt direkt in ein KAIN–PROLOG Modul umgesetzt. Das komplette Modul läßt sich in Abschnitt C.3 finden, hier wird nur der entscheidende Schritt vorgestellt, wie sich die Nichtterminalsymbole berechnen lassen, aus denen ein Teilwort herleitbar ist. Durch das Hinzufügen des Prädikates `table(NonTermSym, Word)` bei der Abarbeitung einer hypothetischen Implikation werden die berechneten Nichtterminalsymbole gespeichert. Der Hauptschritt in der Berechnung sieht dann wie folgt aus:

```
derive_step([NonTermSym | NonTermSymS], Prefix, Word, Len) :-
    % Zuerst eine Produktion auswählen.
    production(NonTermSym, [LeftNonTermSym, RightNonTermSym]),
    % Linkes und rechtes Teilwort des Präfix...
    split(LeftPrefix, RightPrefix, Prefix),
    % ...müssen aus den Nichtterminalsymbolen ableitbar sein.
    table(LeftNonTermSym,  LeftPrefix),
    table(RightNonTermSym, RightPrefix), !,
    (   table(NonTermSym, Prefix)
        -:  derive_step(NonTermSymS, Prefix, Word, Len)
    ).
```

Die Klausel ist im wesentlichen selbsterklärend. Hier wird berechnet, ob sich das Teilwort `Prefix` aus dem Nichtterminalsymbol `NonTermSym` herleiten läßt. Ist dem so, wird mit der weiteren Berechnung fortgefahren.

Bevor jedoch diese Berechnung durchgeführt wird, wird durch eine andere Klausel überprüft, ob die Information nicht schon berechnet wurde. Dies sieht dann so aus:

[5] Dazu muß eine Produktion $p \in P$ von folgender Gestalt sein: Entweder gilt $p = A \rightarrow a$ oder $p = A \rightarrow BC$, wobei $a \in T$ und $A, B, C \in N$.

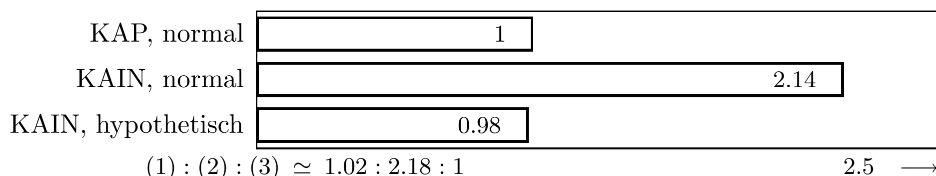


Abbildung 5.7: Relative Laufzeiten des CYK–Algorithmus

```
derive_step([NonTermSym | NonTermSymS], Prefix, Word, Len) :-
    % Dieser Tabelleneintrag ist schon berechnet!
    table(NonTermSym, Prefix), !,
    derive_step(NonTermSymS, Prefix, Word, Len).
```

Man sieht sehr schön, wie sich der Algorithmus direkt aus der verbalen Formulierung in ein leicht verständliches, intuitionistisches logisches Programm umsetzen läßt. Die Formulierung des Algorithmus wird noch dadurch erleichtert, daß Informationen über die zugrundeliegende Grammatik der Abarbeitung durch dynamische Klauseln zur Verfügung gestellt werden kann. In normalem PROLOG muß die ganze Information über die Grammatik in Argumenten übergeben werden, zusätzlich müssen die Tabelle und die Produktionen in einer Liste abgelegt werden. Das führt zum Beispiel dazu, daß das oben geschilderte Prädikat *neun* Argumente in einer normalen PROLOG Formulierung hat.

Vergleicht man die Effizienz einer Formulierung in KA–PROLOG und in KAIN–PROLOG, so ergeben sich die in Abbildung 5.7 gezeigten Werte. Die Bezeichnung der verschiedenen Varianten ist dabei genauso wie im vorigen Beispiel. Man sieht, daß die KAIN–PROLOG Variante, die von hypothetischen Implikationen Gebrauch macht, nicht nur wesentlich übersichtlicher und leichter verständlich, sondern auch effizienter ist. Hier wird nun endgültig klar, daß das KAIN System praktikabel einsetzbar ist: Effizienzsteigerung und Gewinn an Ausdruckskraft gehen Hand in Hand.

5.3.4 Weitere Beispiele

Um den Rahmen dieser Arbeit nicht zu sprengen, werden zwei weitere Beispiele für das intuitionistische logische Programmieren nur kurz vorgestellt, ohne komplette Programme und ohne das Laufzeitverhalten anzugeben. Der Fokus wird sich besonders auf den Einsatz der expliziten Quantifizierung richten. Das Laufzeitverhalten der erweiterten Unifikation ist auch bereits in Abschnitt 5.2.3 untersucht worden. Dabei wurde schon festgestellt, daß das Vorhandensein von explizit quantifizierten Variablen das Laufzeitverhalten der Unifikation nicht entscheidend beeinflußt. Dieser Abschnitt dient also eher dazu, die Verwendbarkeit der expliziten Quantifizierung zu unterstreichen.

5.3.4.1 Deduktive Datenbanken

Das intuitionistische logische Programmieren bietet sehr mächtige Konzepte zur Realisierung von deduktiven Datenbanken an, dieser Aspekt ist auch schon in der Literatur untersucht wor-

den, siehe zum Beispiel [BMV89]. Anfragen können universellen Charakter haben, das haben wir schon in Beispiel 1.2 auf Seite 2 gesehen.

Durch hypothetische Implikationen können zum Beispiel auch Konsistenzprüfungen von Datenbanken durchgeführt werden. Will man eine neue Tatsache `faktum` der Datenbank hinzufügen, ohne die Konsistenz der Datenbank zu verletzen, so kann zum Beispiel die Anfrage `faktum -: not inkonsistent` gestellt werden. Klauseln mit dem Kopfprädikat `inkonsistent` beschreiben dann, wann die Datenbasis inkonsistent ist. Dies funktioniert auch, wenn die Konsistenzbedingungen selber wieder universelle Konzepte aufgreifen: Bedingungen enormer Audruckskraft können so formuliert werden.

Generell kann man sagen, daß es durch die Formulierbarkeit und Ableitbarkeit universeller Konzepte viele Verwendungsmöglichkeiten im Bereich der Wissensrepräsentation gibt.

5.3.4.2 Generierung von Lemmata

Die Effizienz der Ableitung von Anfragen kann sehr heraufgesetzt werden, wenn Lemmata bewiesen werden, die dann bei der weiteren Ableitung eingesetzt werden. Rudimentär trat das schon in Beispiel 5.3.3 auf: Die atomaren Formeln `table/2`, die die Tabelle für die Herleitbarkeit von Wörtern aus Nichtterminalsymbolen speicherten, sind auch Lemmata. Doch diese waren sehr speziell. Durch die Möglichkeit, auch allquantifizierte Aussagen abzuleiten, können auch universelle Lemmata bewiesen und der weiteren Abarbeitung zur Verfügung gestellt werden.

Konkret läßt sich das zum Beispiel für ein effizientes Planungssystem vorstellen. Bevor der eigentliche Planungsvorgang beginnt, wird versucht, einige universelle Lemmata zu beweisen, die auf die Ausgangssituation zugeschnitten sind. Ist der Beweis eines Lemmas erfolgreich, so wird es den Klauseln des Planungsprogramms hinzugefügt. Es muß natürlich angegeben werden, welche Lemmata das System betrachten soll. Unter Umständen könnte man auch versuchen, daß das System bei der Planung bewiesene Formeln zu generalisieren versucht. Nachdem die Ableitung einer Formel für konkrete Individuen geglückt ist, könnte das System versuchen, eine generalisierte Aussage zu beweisen, und die generalisierte Aussage dann für die weitere Planung zu benutzen.

Kapitel 6

Diskussion und Ausblick

In diesem Kapitel wird das in dieser Arbeit entworfene, implementierte und empirisch untersuchte KAIN–PROLOG System diskutiert. Der erste Punkt der Diskussion ist ein Vergleich dieses Systems mit einem in der Literatur publizierten System. Darauf folgt eine Bewertung des erhaltenen Systems. Den Abschluß bildet ein Ausblick auf weitere Möglichkeiten, die sich im Zusammenhang mit dieser Arbeit ergeben.

6.1 Vergleich mit einem anderen Ansatz

In dem Artikel „Implementation Techniques for Scoping Constructs in Logic Programming“ von B. Jayaraman und G. Nadathur [JN91] wird eine Erweiterung der WAM vorgestellt, die auch zur Abarbeitung übersetzter intuitionistischer logischer Programme dient. Diese Maschine dient als Grundlage der Implementierung der logischen Programmiersprache höherer Ordnung λPROLOG [NM88]. Sie wird mit der in Kapitel 3 entworfenen I–WAM verglichen, die Unterschiede herausgearbeitet und bewertet.

6.1.1 Datenrepräsentation

Im Gegensatz zu der Datenrepräsentation bei der I–WAM werden bei dieser Maschine die Konstanten, die bei der Abarbeitung von Allquantoren entstehen, auch intern als Konstanten dargestellt. Dadurch müssen bei der Durchführung des Occur–Check nicht nur die Konstanten, die aus allquantifizierten Variablen hervorgegangen sind, betrachtet werden, sondern auch die normalen Konstanten, die sowieso nicht die erweiterte Bereichskorrektheit verletzen können. Dies führt dazu, wie auch schon in Abschnitt 3.3 diskutiert, daß die Effizienz der Unifikation herabgesetzt wird. Insbesondere verletzt dieses Vorgehen das Kriterium der *Spezialisierung* aus Abschnitt 3.2.

6.1.2 Explizite Quantifizierung

Auch bei der Realisierung der expliziten Quantifizierung unterscheiden sich die beiden Maschinen wesentlich.

Bei dem Anlegen einer Prozedurumgebung mittels einer `allocate`–Instruktion werden alle Variablen der Prozedurumgebung mit dem aktuellen Schachtelungsindex versehen. Dazu erhält

diese Instruktion einen zusätzlichen Parameter, der angibt, wieviele Variablenzellen die Umgebung maximal enthalten wird. Dieses Vorgehen schließt von vornerein viele Optimierungen aus, die sowohl in der WAM als auch in der I–WAM vorhanden sind. Man denke dabei nur an die `get_variable`–Instruktion: Bei dieser Maschine wird erst eine Speicherzelle initialisiert, dann aber direkt überschrieben. In der I–WAM findet keine Initialisierung, sondern nur das Überschreiben statt. Diese Optimierung wurde mit Hilfe des Lemmas 2.4 auf Seite 20 von der WAM auf die I–WAM übertragen.

Ein Allquantor schlägt sich jetzt auch in Instruktionen nieder, die den Schachtelungsindex erhöht, und ihn nach Abarbeitung des Allquantors wieder vermindert. Im Gegensatz zu der I–WAM werden jetzt alle explizit quantifizierten Variablen durch spezielle Instruktionen mit dem neuen Schachtelungsindex initialisiert. Diese Initialisierung findet nicht bei der ersten Verwendung der Variablen statt, sondern dann, wenn der Bindungsbereich eines Quantors betreten wird. Werden diese Variablen dann verwendet, so muß wieder eine separate Instruktion erzeugt, beziehungsweise abgearbeitet werden. Insbesondere schlagen sich dann auch Existenzquantoren in Instruktionen nieder.

Zusätzlich werden alle explizit quantifizierten Variablen in der Prozedurumgebung abgelegt, sie sind also permanent. Dies ist nötig, da zwischen der Initialisierung und Verwendung so einer Variablen Prozeduraufrufe stattfinden können.

Insgesamt weist dieses Vorgehen mehrere Nachteile auf: Der Schachtelungsindex einer Variablen wird unter Umständen mehrmals initialisiert, insbesondere werden alle Variablen initialisiert, bevor feststeht, ob diese jemals bei der Abarbeitung benutzt werden. Dies ist ein „Verstoß“ gegen das Kriterium *Verzögerung* aus Abschnitt 3.2. Zudem führt das zu sehr viel größeren Prozedurumgebungen. Hier wird unsinnigerweise die in der normalen WAM benutzte Speicherallokation umgestoßen, obwohl es möglich ist, diese vollkommen zu erhalten, wie sich bei dem Entwurf der I–WAM gezeigt hat.

6.1.3 Hypothetische Implikationen

Der wohl gravierendste Unterschied der in dem Artikel beschriebenen Maschine zur I–WAM ist, daß es keine explizit statischen Klauseln gibt. Alle Klauseln werden über Strukturen verwaltet, die den Implikationspunkten ähnlich sind. Zusätzlich kann auf die Klauseln nur sequentiell zugegriffen werden, ohne daß eine Berücksichtigung des ersten Argumentes eines Prozeduraufrufes stattfindet. Dies führt generell zu einer geringeren Effizienz bei der Auswahl von Klauseln. In Abschnitt 5.2.2 war deutlich zu erkennen, daß schon die Auswahl von Klauseln bei der I–WAM erheblich langsamer als bei einem normalen PROLOG System ist. Also sind in diesem Punkt erhebliche Effizienzeinbußen bei der in der Literatur publizierten Maschine zu erwarten.

Das Bereitstellen von dynamischen Klauseln funktioniert analog zu der beschriebenen Technik bei der I–WAM: Auch hier steht das Erzeugen von Implikationspunkten im Vordergrund. Diese werden zwar auch auf dem Kontrollstapel angelegt, aber in der gerade aktuellen Prozedurumgebung. Das führt bei der Verwaltung von Implikationspunkten durch das Rücksetzen zu komplizierteren Operationen. Denn es kann nicht mehr ohne weiteres an der Lage der verschiedenen Daten auf dem Kontrollstapel festgestellt werden, welche Implikationspunkte gelöscht oder wieder verfügbar gemacht werden müssen.

Die Verwaltung der geteilten Variablen ist auch anders gelöst. Es findet zwar auch ein „flacher Zugriff“ auf die geteilten Variablen statt, aber das Kopieren der Werte wird durch

die Interpretation einer Tabelle, die zur Übersetzungszeit angelegt wurde, gesteuert. Dadurch ist zu erwarten, daß der Zugriff auf die geteilten Variablen nicht so effizient wie bei der I–WAM stattfinden kann. Denn bei der I–WAM wird dies durch einfache Abarbeitung spezieller Instruktionen realisiert, es findet keine Interpretation einer Datenstruktur durch eine Instruktion statt.

6.1.4 Fazit

Als Fazit läßt sich also angeben, daß die I–WAM in vielen Punkten entscheidende Vorteile in ihrer Konzeption gegenüber der Maschine aus dem Artikel besitzt.

Neben den oben diskutierten Vorteilen, muß auch berücksichtigt werden, daß es zu der I–WAM bereits eine konkrete Implementierung gibt. Aufgrund dieser Implementierung konnten in Kapitel 5 konkrete Aussagen zur Leistungsfähigkeit der entworfenen Maschine gemacht werden.

6.2 Bewertung

In Kapitel 5 hat sich gezeigt, daß das erhaltene System den geforderten Ansprüchen im wesentlichen genügt. Einerseits werden die Paradigmen des intuitionistischen logischen Programmierens effizient unterstützt. Andererseits ist das System von der angebotenen Funktionalität so vollständig, daß mit ihm praktisch gearbeitet werden kann. In diesem Abschnitt wird noch einmal vertieft untersucht, in welchen Punkten die I–WAM und das KAIN–PROLOG System den praktischen Anforderungen genügen, und in welchen Punkten noch eine weitere Verbesserung angebracht ist, wobei Lösungsansätze für existierende Probleme skizziert werden.

6.2.1 I–WAM

Zuerst wollen wir die Situation auf der Maschinenseite untersuchen.

6.2.1.1 Unifikation und explizite Quantifizierung

Die Realisierung der Unifikation und der expliziten Quantifizierung kann man als zufriedenstellend gelöst betrachten. Alle Spezialfälle, die auch bei dem Entwurf der Instruktionen der originalen WAM berücksichtigt wurden, sind bei der I–WAM auch in gebührender Art und Weise behandelt worden. Insbesondere, daß die zentralen Punkte der Variablenklassifikation und Register- und Speicherallokation übertragen werden konnten, ist als Erfolg zu werten. Betrachtet man dagegen die Lösung, die in dem vorhergehenden Abschnitt vorgestellt wurde, ist erwähnenswert, daß die explizite Quantifizierung keinen Einfluß auf diese zentralen Punkte hat.

Es ist jedoch zu überlegen, ob es sich lohnt, einen anderen Unifikationsalgorithmus in das System zu integrieren. Die Verbesserungen, die in [CB83, RP89] beschrieben werden, können unter Umständen zu einer effizienteren Unifikation führen, siehe dazu auch Abschnitt 2.2. Denn diese verringern vor allen Dingen die Anzahl der Terme, die bei einem Occur–Check untersucht werden müssen: Da der Occur–Check bei der I–WAM mehr Funktionalität trägt, kann sich der Einsatz dieser Verbesserung unter Umständen entscheidend bemerkbar machen. Dabei ist zu berücksichtigen, daß nicht nur die eigentliche Unifikationsoperation erweitert werden müßte,

sondern daß auch alle spezialisierten Unifikationsinstruktionen modifiziert werden müßten: Der Aufwand hierfür ist jedoch beträchtlich.

6.2.1.2 Hypothetische Implikationen

In diesem Bereich ist die Leistung der I–WAM nicht ganz so positiv zu beurteilen. Das generelle Problem ist, einen möglichst direkten Zugriff auf die einzelnen Klauseln eines Prädikates zu haben. Bei diesem Zugriff sollte nicht nur der Objekttyp des ersten Argumentes berücksichtigt werden, sondern auch der Inhalt des Objektes selbst: Ein Direktzugriff nach einer Konstanten oder einem Funktor sollte möglich sein.

Der andere Aspekt ist, wie teuer es ist, die Datenstrukturen, die einen Direktzugriff ermöglichen, zu verwalten. Einerseits spielt dabei das Anlegen dieser Datenstrukturen eine Rolle, andererseits aber auch die Veränderung der Strukturen durch das Rücksetzen.

Die schnellste Methode beim Zugriff wäre, mittels einer geeigneten Hash–Funktion diesen zu realisieren. Doch das ist kaum zu erreichen. Denn die Hash–Funktion muß ja wegen dem Hinzufügen von dynamischen Klauseln erweiterbar sein. Aber das Verwalten von erweiterbaren Hash–Funktionen ist sehr teuer. Das fällt um so mehr ins Gewicht, da beim Rücksetzen sehr oft Klauseln wieder entfernt und hinzugefügt werden müssen. Die dabei zu verwaltenden Informationsmengen würden Kosten verursachen, die in fast allen Fällen weit über dem Nutzen des Direktzugriffs liegen.

Ein weitere Idee ist, Implikationspunkte nicht mehr linear zu verketten, sondern sie in einer Binärbaumstruktur anzuordnen. Hier hat man zwar keinen Direktzugriff mehr, kann aber doch noch sehr schnell durch binäres Suchen zugreifen. Das Problem ist hierbei, daß dieser binäre Baum sehr schnell zu einer sequentiellen Liste entarten kann, wenn die Argumente, über die zugegriffen werden sollen, beim Hinzufügen von dynamischen Klauseln eine gewisse Ordnung aufweisen: Dieses wird gerade im Zusammenhang mit Zahlen relativ häufig sein. Lösen ließe sich dieses Problem durch eine balancierte Datenstruktur, zum Beispiel AVL–Bäume [Wir83, Abschnitt 4.4].

Die Verwaltung eines Binärbaumes, ob balanciert oder nicht, durch das Rücksetzen ließe sich relativ einfach realisieren. In den Implikationspunkten wird explizit abgespeichert, welche Veränderungen an der Baumstruktur vorgenommen wurden, um eine dynamische Klausel hinzuzufügen. Beim Rücksetzen kann dann diese Information entweder dazu verwendet werden, dynamische Klauseln ohne das Wiederausführen dieser Operationen erneut verfügbar zu machen, oder um die Wirkung dieser Operationen wieder rückgängig zu machen. Die Idee ist also dieselbe, die auch dem Lösen von Variablenbindungen beim Rücksetzen zugrundeliegt.

6.2.2 Übersetzung

Ein weiterer Aspekt betrifft die Codeerzeugung für die I–WAM. Hier gibt es noch eine Menge Schwächen.

Im Gegensatz zu der Maschine befinden sich die Hauptschwächen bei der Übersetzung im Bereiche der Unifikation und der expliziten Quantifizierung. Alle Optimierungen, die der Übersetzer bis jetzt vornimmt, sind direkt bei der Übersetzung ohne wesentlichen Mehraufwand zu realisieren. Doch gerade hier wäre ein guter Punkt, die Leistungsfähigkeit des gesamten Systems dadurch zu erhöhen, daß bei der Übersetzung mehr investiert wird.

Neben üblichen Optimierungen, die der Übersetzer nicht vornimmt, wie zum Beispiel Einsparungen von Operationen bei der Unifikation von Köpfen durch Umordnung der Unifikationsbefehle [MJT91] oder Wiederverwendung von Zellen auf dem Kontrollstapel, ist besonders die Erzeugung der Befehle zur expliziten Quantifizierung noch unbefriedigend. Dieses Problem sollte so gelöst werden, daß so wenig `offset`–Befehle wie möglich erzeugt werden. Hier können die gleichen Methoden wie in [MJT91] verwendet werden: Eine Umordnung der Befehle, die die Terme ablegen. Dies ist der einzige Aspekt, der direkt die Erzeugung von Instruktionen betrifft, die neu in der I–WAM sind.

6.2.3 Standardprädikate

Probleme ergeben sich auch aus der Kombination der PROLOG Standardprädikate mit der expliziten Quantifizierung.

Problematisch ist insbesondere die interne Datenbasis. Wenn Terme, die Variablen enthalten, aus der Datenbasis extrahiert werden, ist nicht ohne weiteres klar, welche Schachtelungsindices die Variablen erhalten sollen. In der Implementierung ist das so gelöst, daß existentiell quantifizierte Variablen den Schachtelungsindex null und allquantifizierte Variablen den gerade aktuellen Schachtelungsindex erhalten. Bei der Unifikation solcher Terme können also keine allquantifizierten Variablen an diese existentiellen Variablen gebunden werden, und umgekehrt kann so ein Term nur an eine andere existentielle Variable gebunden werden, die *nach* der Abarbeitung des letzten Allquantors erzeugt wurde. Dieses Vorgehen sichert zumindest zu, daß die in Lemma 2.4 auf Seite 20 formulierte Relation erhalten bleibt.

Die Probleme mit der Datenbasis lassen sich noch relativ leicht verschmerzen, da man durch hypothetische Implikationen ein viel effizienteres und mächtigeres Instrument in die Hand bekommt. Gravierender sind diese Einschränkungen bei dem `findall`–Prädikat, das alle möglichen Lösungen für eine Anfrage in einer Liste aufsammelt. Dieses vielseitig verwendbare Prädikat macht intern von der Datenbank Gebrauch, die aufgesammelten Lösungen unterliegen den oben geschilderten Einschränkungen.

Eine mögliche Lösung dieses Problems besteht darin, das Prädikat auf der Maschinenebene zu realisieren. Das müßte eigentlich recht einfach sein, da man durch Implikationspunkte schon einen guten Ausgangspunkt besitzt, um Informationen zu speichern und später wiederzugewinnen.

6.2.4 Modulkonzept

Ein weiteres Problem besteht in der Interaktion des Modulkonzeptes mit hypothetischen Implikationen. Das Modulkonzept soll die Implementierung der Prädikate verbergen. Andererseits will man aber auch dynamische Klauseln zu Prädikaten definieren, die aus einem anderen Modul importiert werden. Man denke dabei zum Beispiel an das Prädikat, das die Ordnungsrelation bei dem generischen Sortieren in Abschnitt 5.3.1 beschreibt.

Wenn man die Abarbeitung einer hypothetischen Implikation betrachtet, so werden die hinzugefügten Klauseln für die Konklusion der hypothetischen Implikation *importiert*. Man kann dies als Grundlage für ein erweitertes Modulkonzept sehen. Zusätzlich bieten die geteilten Variablen einer hypothetischen Implikation die Möglichkeiten für Module mit lokalen Variablen. Ein Modulkonzept, das auf dieser Idee aufsetzt, ist dynamisch: Erst zur Laufzeit werden die

Module importiert. Damit braucht man für eine effiziente Realisierung so eines Konzeptes insbesondere eine effiziente Verwaltung dynamischer Klauseln, wie sie schon in Abschnitt 6.2.1.2 diskutiert wurde.

Ob ein dynamisches Modulkonzept oder eine Erweiterung des statischen Modulkonzeptes, wie es jetzt angeboten wird, sinnvoll ist, und wie eine konkrete Realisierung aussehen kann, müßte jedoch genauer untersucht werden.

6.3 Ausblick

In diesem Abschnitt wird vorgestellt, welche weiteren Aspekte diese Arbeit eröffnet. Einerseits betrifft dieses mögliche Erweiterungen des hier vorgestellten Systems, andererseits kommt auch die Grundlage dieser Arbeit, die WAM, zur Sprache.

6.3.1 Integration anderer Erweiterungen

In dieser Arbeit hat sich gezeigt, daß es möglich ist, intuitionistische logische Programme effizient auszuführen. In den betrachteten Beispielen wurde auch deutlich, daß es vielfältige Einsatzmöglichkeiten der neuen Sprachkonstrukte gibt. Doch kann durch die Integration anderer Erweiterungen des logischen Programmierens die Ausdrucksmächtigkeit noch weiter gesteigert werden. Die Integration zweier Erweiterungen, die sich direkt anbietet, wird hier diskutiert.

6.3.1.1 Ordnungssortiertes logisches Programmieren

In diesem System ist es möglich, universelle Konzepte sowohl durch die Angabe von Klauseln zu formulieren, als auch durch hypothetische Implikationen abzuleiten. In der Regel sollen solche Konzepte jedoch nicht tatsächlich für *alle* möglichen Individuen formuliert, beziehungsweise abgeleitet werden. Vielmehr ist man daran interessiert, nur bestimmte Klassen von Individuen zu betrachten. Prototypisch für diesen Sachverhalt ist das Beispiel 1.2 auf Seite 2: Durch die Prädikate *behälter* und *keim* werden die zu betrachtenden Individuen eingeschränkt.

Abhilfe in dieser Situation kann das ordnungssortierte logische Programmieren schaffen. Einerseits kann durch die Angabe von Sorten die Formulierung prägnanter und übersichtlicher gestaltet werden. Andererseits wird dadurch auch die Ableitung von universellen Konzepten effizienter: Der Suchraum bei der Ableitung wird eingeschränkt, und die Menge der Klauseln, die die hypothetische Information spezifiziert, nimmt ab.

Aus diesen Gründen bietet es sich an, das ordnungssortierte logische Programmieren mit dem intuitionistischen logischen Programmieren zu kombinieren. Die Ausgangssituation für die Integration von Sorten in das hier entworfene System ist ideal: Die *orthogonale* Erweiterung der WAM zur I–WAM macht es möglich, die zusätzlich benötigte Funktionalität in gleicher Weise hinzuzufügen, wie es auch für das LOPSTER System [Wei89] durchgeführt wurde. Im Gegensatz zum KA–PROLOG System ist die Darstellung der Datenstrukturen des KAIN–PROLOG Systems rein abstrakt gehalten, was die Erweiterbarkeit des Systems sehr erleichtert.

6.3.1.2 Logisches Programmieren mit Randbedingungen

In dem Beispielprogramm für das n–Damen Problem in Abschnitt 5.3.2 wurden die Damen durch das Hinzufügen einer dynamischen Klausel mit geteilten Variablen auf dem Brett plaziert, und erst später bei der Abarbeitung die eigentlichen Koordinaten der Plazierung berechnet. Dieses Vorgehen ist typisch: Man nimmt unvollständige hypothetische Information an und will diese Information bei der weiteren Ableitung vervollständigen. In dem erwähnten Beispiel geschah diese Vervollständigung allein durch das Generieren und Testen von verschiedenen Vervollständigungen.

Eine elegantere, mächtigere und auch effizientere Methode wären hierfür *Randbedingungen*[1]: Bei der Annahme der hypothetischen Information könnte von vornerein angegeben werden, welchen Randbedingungen die Belegungen der freien Variablen gehorchen müssen. Man kann sich leicht vorstellen, daß die Kombination von Randbedingungen und hypothetischer Information äußert vielfältige Einsatzmöglichkeiten bietet. Das ist eine erste Idee, es kann jedoch keine Aussage darüber gemacht werden, welcher Aufwand die Kombination dieser Ansätze mit sich bringt.

6.3.2 Komplexität der WAM

Bei der Erweiterung der WAM zur I–WAM tritt sehr klar die relativ hohe Komplexität der Instruktionen zu Tage. Betrachtet man zum Beispiel die Instruktionen, die Variablen erzeugen und dann irgendwo ablegen, so gibt es eine Fülle von Instruktionen, die sich teilweise in ihrer Funktionalität nur unwesentlich unterscheiden. Die Punkte, worin sich diese unterscheiden, sind die folgenden: Wie wird die Variable erzeugt, und wo wird sie abgelegt. Es wäre also sinnvoll, die Instruktionen weiter aufzuspalten, und somit insgesamt ihre Anzahl und Komplexität zu verringern. Einerseits würde dieses Vorgehen dazu führen, die Realisierung der Maschine zu vereinfachen. Andererseits ergeben sich für den Übersetzer auch mehr Möglichkeiten zur Optimierung.

[1]Logisches Programmieren mit Randbedingungen als Übersetzung von „constraint logic programming“

Anhang A

Instruktionen und Operationen der I–WAM

In diesem Anhang ist eine komplette Beschreibung aller Konzepte, Datenstrukturen und Algorithmen enthalten, die in die I–WAM Eingang finden.

Konzeptionell lehnen sich die Ausführungen hier an die Darstellung der KAP–Maschine [Nei86] an. Teilweise wird die Beschreibung von einigen Instruktionen, die in [Nei86] definiert werden, hier ausgelassen: Diese Instruktionen können dann unverändert von dort übernommen werden; sie sind für das Verständnis und eine vollständige Übersicht über die hier beschriebene Maschine auch nicht nötig.

Generell wird hier nur die Funktionalität der Maschine beschrieben, zum Einstieg und zur Erklärung der Erweiterungen ist das Kapitel 3 zum Entwurf der Maschine heranzuziehen.

Bei den angegebenen Programmstücken wird eine Adressarithmetik wie in der Programmiersprache **C** [KR88] angenommen. Dabei bezeichnet $\uparrow$ die Inhaltsoperation und $V\downarrow$ die Adresse der Variable V. Ist p ein Zeiger auf ein Objekt, so ist $p+i$ $(p-i)$ ein Zeiger, der auf ein Objekt zeigt, das $+i$ $(-i)$ Objekte entfernt im Speicher liegt. $i{+}{+}$ $(i{-}{-})$ inkrementiert (dekrementiert) den Wert von i um eins *nach* der Benutzung von i; $++i$ $(--i)$ inkrementiert (dekrementiert) den Wert von i um eins *vor* der Benutzung von i.

Sind Typen von Zeigern nicht von Interesse, so wird dieser Typ einfach mit $\uparrow\cdots$ bezeichnet: Es geht hier nicht um eine strikte Typisierung, sondern die Typen sollen nur die Verständlichkeit der Ausführungen erhöhen.

A.1 Datenrepräsentation

Der grundlegende Datentyp der Maschine ist der *Term*. Die Darstellung von Termen ist im wesentlichen bis auf Erweiterungen zur Behandlung von Variablen und Referenzen unverändert.

Bislang waren freie Variablen und Referenzen in den Maschinendarstellungen gleichartige Objekte. Das ist jetzt nicht mehr so. Es muß zwischen existentiell und universell quantifizierten Variablen unterschieden werden, da erstere an andere Datenobjekte gebunden werden können, letztere jedoch eine spezielle Form von Konstanten sind.

Zusätzlich ist noch die existentiell quantifizierte Variable konzeptionell von der Referenz zur Realisierung von Bindungen zwischen Variablen zu unterscheiden, da diese Variablen nicht mehr

nur eine Selbstreferenz zur Identifikation enthalten, sondern auch den Schachtelungsindex des Quantorenpräfixes zur korrekten Durchführung der Unifikation.

Die Typvereinbarung für Terme sieht folgendermaßen aus:

Tag = (Free, All, Ref, Const, Struct, List, Functor);	Das sind alle oben angesprochenen Arten von Datenobjekten.
Obj = **record** **case** *Tag* : Tag **of** Free: *Addr* : ↑Obj; *Nest* : **integer**;	Existentiell quantifizierte Variablen enthalten zur Realisierung der Bindung durch Überschreiben die Adresse *Addr*, an der sie erzeugt wurden. Zusätzlich ist auch der Schachtelungsindex enthalten.
All: *IdNr* : **integer**; *Nest* : **integer**;	Allquantifizierte Variablen enthalten eine Nummer *IdNr*, die der eindeutigen Identifikation dient und genau wie die existentiell quantifizierten Variablen den Schachtelungsindex.
Ref, Struct, List: *Addr* : ↑Obj; Const: *Val* : ConstType; Functor: *Val* : FunctorType; **esac**; **end**;	Referenzen enthalten die Adresse *Addr* des referenzierten Objektes, Strukturobjekte und Listenobjekte wie üblich Adressen auf den Funktor, respektive auf den Kopf der Liste. An den Funktor schließen sich der Stelligkeit des Funktors gemäß die Datenobjekte der Argumente an; auf den Kopf der Liste folgt das den Rest derselben enthaltende Objekt. Die interne Repräsentation von Konstanten und Funktoren ist für die weitere Darstellung hier nicht von Interesse.

Datenobjekte des Typs Obj werden durch *Tupel* bezeichnet: Zum Beispiel wird eine existentiell quantifizierte Variable mit den Komponenten *Addr* = *AnyAddr* und *Nest* = *AnyNest* durch ⟨Free, *AnyAddr*, *AnyNest*⟩ beschrieben.

Für das Erzeugen allquantifizierter Variablen gibt es eine spezialisierte Operation, die bei jeder Verwendung eine „neue“ Variable dieses Typs generiert:

function *GenAll*(*Nest* : **integer**) : Obj;	
return ⟨All, *NewIdNr*++, *Nest*⟩;	Liefert eine allquantifizierte Variable zurück, die als Schachtelungsindex den Wert von *Nest* und als Identifikationsnummer von *NewIdNr* enthält. Der Wert von *NewIdNr* wird anschließend inkrementiert, so daß jeder Aufruf dieser Funktion eine allquantifizierte Variable mit unterschiedlicher Identifikationsnummer erzeugt.

Zur weiteren Beschreibung sind noch zwei weitere Funktionen nötig, die der direkten Behandlung von Termen dienen.

function *GetArity*(*Functor* : FunctorType) : **integer**;

...

Liefert die Stelligkeit eines Funktorobjektes.

Das Ablaufen von Referenzketten erledigt die folgende Operation.

function *DeRef*(*RefObj* : Obj) : Obj;

```
while RefObj.Tag = Ref do
  RefObj ← RefObj.Addr↑;
od;
return RefObj;
```

Läuft die bei dem übergebenen Objekt *RefObj* beginnende Referenzkette ab und liefert den dereferenzierten Wert zurück.

A.2 Register

Der Registersatz der I–WAM setzt sich aus Datenregistern und aus Spezialregistern, die auf verschiedene Belange der Maschine zugeschnitten sind, zusammen. Die Datenregister dienen einerseits zur Parameterübergabe bei der Abwicklung von Prozeduraufrufen; sie werden dann mit `A`[i] bezeichnet. Andererseits werden sie auch zur Aufbewahrung von temporären Variablen verwendet; hier werden sie mit `X`[n] bezeichnet.

Die dedizierten Register der I–WAM sind in Tabelle A.1 zusammengestellt; unter dem Eintrag **Wahlpunkt** ist vermerkt, ob das Register in einem Wahlpunkt gespeichert wird. Die Bedeutung der einzelnen Register ist wie folgt:

Umgebungszeiger Dieser zeigt auf die gerade aktuelle Prozedurumgebung. Bezüglich dieses Registers erfolgt auch der Zugriff auf eine permanente Variable `Y`[n].

Fortsetzungspunkt Beinhaltet die Adresse, an der bei einem Rücksprung aus einer Prozedur die Abarbeitung fortgesetzt wird. Diese Adresse liefert auch den Zugriffspfad auf die Anzahl der permanenten Variablen der aktuellen Prozedurumgebung.

Instruktionszeiger Zeigt auf die nächste auszuführende Instruktion; dadurch wird auch auf die Operanden einer Instruktion zugegriffen.

Wahlpunktzeiger Gibt die Position des zuletzt angelegten Wahlpunktes auf dem Kontrollstapel an.

Haldenrücksetzpegel Enthält den Wert des Haldenpegels, der im zuletzt angelegten Wahlpunkt gespeichert wurde.

Strukturzeiger Zeigt auf das zur Unifikation im Lesemodus bei der Parameterübergabe gerade benötigte Objekt auf der Halde.

Spurstapelpegel Zeigt auf den ersten freien Platz, wo eine beim Rücksetzen wiederherzustellende Variable gespeichert werden kann.

Tabelle A.1: Dedizierte Register der I–WAM

Register	Bezeichnung	Wahlpunkt
E	Umgebungszeiger	+
CP	Fortsetzungspunkt	+
P	Instruktionszeiger	−
B	Wahlpunktzeiger	+
HB	Haldenrücksetzpegel	−
S	Strukturzeiger	−
TR	Spurstapelpegel	+
H	Haldenpegel	+
IP	Implikationspunktzeiger	+
EIP	Implikationspunktumgebungszeiger	−
NL	Schachtelungsindex	+
$PROC$	Zeiger auf Prozedurbeschreibungsblock	+
$WRITING$	Modus zur Parameterunifikation	−
WNL	Schreibschachtelungsindex	−

Haldenpegel Zeigt auf die erste freie Speicherzelle der Halde.

Implikationspunktzeiger Hierdurch wird der zuletzt angelegte Implikationspunkt referenziert.

Implikationspunktumgebungszeiger Enthält den Umgebungszeiger der Prozedur, die eine gerade bearbeitete dynamische Klausel aktiviert hat. Hierdurch ist ein Zugriff auf die geteilten Variablen `SY`[m] möglich.

Schachtelungsindex Gibt die aktuelle Anzahl von Allquantoren an, in deren Bindungsbereich sich die gerade bearbeitete Instruktion befindet.

Zeiger auf Prozedurbeschreibungsblock Gibt den Zugriffspfad für verschiedene Informationen zu der gerade bearbeiteten Prozedur.

Modus zur Parameterunifikation Gibt an, ob sich die Maschine bei der Unifikation von Parametern im Schreibmodus oder im Lesemodus befindet.

Schreibschachtelungsindex Gibt den Schachtelungsindex an, der bei der Parameterunifikation im Schreibmodus zur Erzeugung neuer existentiell quantifizierter Variablen herangezogen wird.

A.3 Arbeitsbereiche

Neben den Registern der Maschine gibt es noch drei andere Bereiche, in denen sich Daten befinden: Die Halde, der Kontrollstapel und der Spurstapel. Die Anordnung dieser drei Ar-

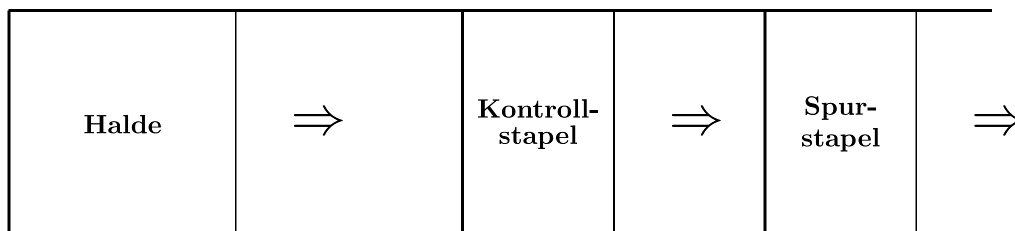


Abbildung A.1: Anordnung der Arbeitsbereiche

beitsbereiche ist in Abbildung A.1 dargestellt.

Die Pfeile in der Abbildung geben an, in welche Richtung die Datenbereiche wachsen. Die Anordnung des Kontrollstapels auf höheren Speicheradressen als die der Halde ist zwingend notwendig. Der Grund hierfür ist, genauso wie bei der normalen WAM, daß diese Anordnung zusammen mit Einhaltung der Tatsache, das Bindungen von Variablen immer in Richtung niedriger Adressen vorgenommen werden, sicherstellt, daß keinesfalls Objekte durch Überschreiben zerstört werden, solange noch Referenzen auf sie existieren.

A.3.1 Die Halde

Die Halde dient wie sonst auch zur Ablage von Objekten, die bei der Unifikation von Strukturen und Listen aufgebaut werden.

A.3.2 Der Kontrollstapel

Der Kontrollstapel enthält jetzt drei verschieden Typen von Daten: Umgebungen, Wahlpunkte und Implikationspunkte. Nachfolgend wird dargelegt, welche Daten diese jeweils enthalten, und welche Operationen hierauf benötigt werden.

Um die Darstellung zu erleichtern, wird vorausgesetzt, daß die drei Register E, B und IP beim Start der Maschine mit dem Wert **nil** initialisiert sind.

A.3.2.1 Umgebungen

Umgebungen sind in der Form einer linearen, verzeigerten Liste auf dem Kontrollstapel angeordnet. Auf die zuletzt angelegte Umgebung zeigt das Register E. Zur Speicherung der Verzeigerung von Umgebungen enthält jede Umgebung auch den Wert des Registers E vor dem Anlegen dieser Umgebung. Schematisch ist dies auch in Abbildung A.2 zu sehen. Zusätzlich wird bei dem Anlegen einer Prozedurumgebung auch der Wert des Fortsetzungspunktregisters CP gespeichert.

Als Zugriffsoperation auf den Inhalt einer Prozedurumgebung wird einfach die Notation $E\uparrow.E$ benutzt: Hierdurch wird also zum Beispiel der in der aktuellen Umgebung gespeicherte Wert des Umgebungszeigerregisters beschrieben. Generell sind alle Komponenten dieser Datenobjekte auf dem Kontrollstapel, sofern sie zur Ablage von Registerwerten dienen, auch mit dem Namen des jeweiligen Registers bezeichnet.

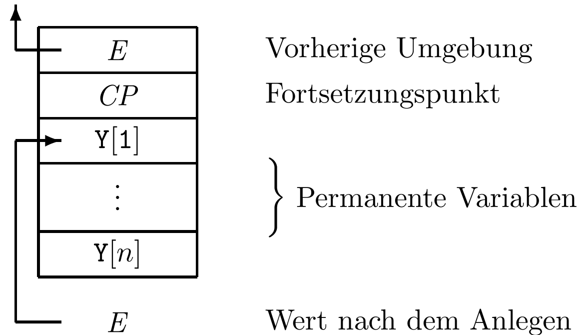


Abbildung A.2: Eine Prozedurumgebung auf dem Kontrollstapel

Die einzige Abweichung von dieser Konvention ist die Bezeichnung der permanenten Variablen einer Prozedurumgebung: Anstatt auf die n–te permanente Variable in der aktuellen Prozedurumgebung mittels $(E + n - 1)\uparrow$ zuzugreifen, wird diese einfach durch $\mathtt{Y}[n]$ bezeichnet. Analog ist die Bezeichnung der n–ten permanenten Variablen in einer Umgebung, die durch den Implikationspunktumgebungszeiger *EIP* referenziert wird: $\mathtt{SY}[m]$[1].

A.3.2.2 Wahlpunkte

Wahlpunkte nehmen alle Daten auf, die nötig sind, um beim Rücksetzen den Zustand der Maschine zu dem Zeitpunkt, an dem der Wahlpunkt angelegt wurde, wieder herzustellen. Schematisch ist ein Wahlpunkt in Abbildung A.3 dargestellt.

Genau wie bei den Prozedurumgebungen bilden die Wahlpunkte eine lineare Liste: Der Vorgänger läßt sich jeweils aus dem im Wahlpunkt gespeicherten Wert von B entnehmen.

Die Bedeutung der Daten in einem Wahlpunkt ist im wesentlichen klar: Sie dienen zur Restauration des Maschinenzustandes beim Rücksetzen. Dieses geht durch das Wiederherstellen der gespeicherten Registerwerte einschließlich der Argumentregister. Der Bezeichnungsweise für Zugriffe auf die Prozedurumgebung folgend, enthält das Feld $B\uparrow.\mathit{DynAlt}$ einen Zeiger auf den Implikationspunkt, der die Information enthält, die zur Ausführung des nächsten dynamischen Codestückes notwendig ist. Analog dazu enthält $B\uparrow.\mathit{StatAlt}$ die Adresse der Instruktion, an die bei der Ausführung der nächsten statischen Alternative gesprungen werden muß.

Das Anlegen eines Wahlpunktes wird durch die folgende Operation beschrieben:

procedure *MkChoicePt*(*Arity* : **integer**; *StatAlt*, *DynAlt* : ↑ · · ·);

$\mathit{Top} \leftarrow \max\{E + \mathit{EnvSize}(\mathit{CP}), B, \mathit{IP}\}$; **for** *ArgCt* ← 1 **to** *Arity* **do** $\mathit{Top}{+}{+}\uparrow \leftarrow \mathtt{A}[\mathit{ArgCt}]$; **od**;	Nach dem Berechnen der ersten freien Speicherzelle auf dem Kontrollstapel werden die *Arity* Argumente sukzessive dort abgelegt.

[1]Das `S` vor dem `Y` steht für geteilt, englisch shared.

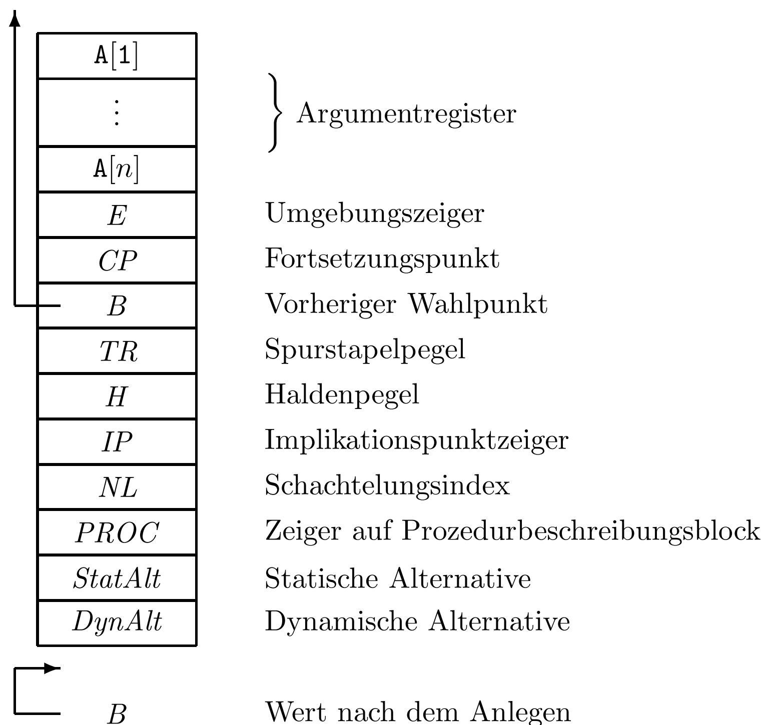


Abbildung A.3: Ein Wahlpunkt auf dem Kontrollstapel

$Top{+}{+}\uparrow \leftarrow E;$
$\vdots$
$Top{+}{+}\uparrow \leftarrow PROC;$
$Top{+}{+}\uparrow \leftarrow StatAlt;$
$Top{+}{+}\uparrow \leftarrow DynAlt;$
$HB \leftarrow H;$
$B \leftarrow Top;$

Darauf folgend werden die anderen Maschinenregister, *StatAlt* und *DynAlt* abgelegt. Schließlich wird noch der Haldenrücksetzpegel *HB* gesetzt und *B* aktualisiert.

Eine Hilfsoperation zum Wiederauffinden der gespeicherten Argumentregister ist:

function $LastArg(B : \uparrow \cdots) : \uparrow \cdots;$
$\cdots;$

Liefert die Adresse der Speicherzelle, die auf die Zelle folgt, in dem das letzte Argumentregister abgelegt wird. Nach Abbildung A.3 ist das die Adresse der Zelle, in der *E* abgelegt wird.

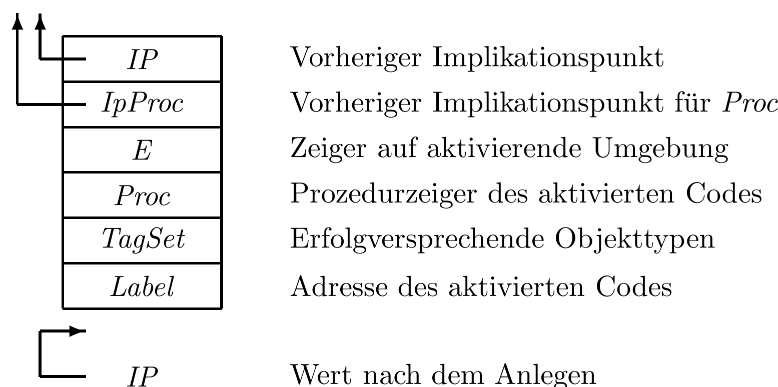


Abbildung A.4: Ein Implikationspunkt auf dem Kontrollstapel

A.3.2.3 Implikationspunkte

Implikationspunkte dienen zur Verwaltung der Information über gerade aktivierte dynamische Codestücke. Schematisch ist ein Implikationspunkt in Abbildung A.4 dargestellt.

Genau wie die bei den Umgebungen und Wahlpunkten, bilden auch die Implikationspunkte durch die Verkettung mittels der Speicherung des Implikationspunktzeigers *IP* eine lineare Liste. Zusätzlich ist diese lineare Liste noch partitioniert in die Implikationspunkte für jeweils eine Prozedur.

Ist *Proc* ein Zeiger auf einen Prozedurbeschreibungsblock, so enthält *Proc*↑.*IP* den letzten Implikationspunkt für diese Prozedur. Der Verkettung durch die in den Implikationspunkten gespeicherten Werte von *IpProc* folgend, kann sukzessive auf alle Implikationspunkte für eine Prozedur zugegriffen werden.

Weiterhin gibt *IP*↑.*Proc* den Beschreibungsblock der jeweiligen Prozedur an, *IP*↑.*Label* den Start des zugehörigen Codestückes, *IP*↑.*E* die Umgebung der Prozedur, die die Klausel aktiviert hat, und schließlich *IP*↑.*TagSet* eine Menge von Objekttypen, in der der Objekttyp des ersten Argumentes bei einem Prozeduraufruf enthalten sein muß, wenn die Abarbeitung des dynamischen Codestückes erfolgreich sein soll. Gibt es kein erstes Argument, so enthält diese Menge alle Objekttypen.

Das Suchen nach einem Implikationspunkt für eine Prozedur mit entsprechender Menge von Objekttypen wird folgendermaßen realisiert:

```
function SearchImpPt(Start : ↑ · · · ; Tag : Tag) : ↑ · · ·;
  while Start ≠ nil do
    if Tag ∈ Start↑.TagSet then
      return Start;
    fi;
    Start ← Start↑.IpProc;
  od;
  return nil;
```

Die bei *Start* beginnende Liste der Implikationspunkte wird durchsucht. Der Zeiger auf den ersten Implikationspunkt, der in der Tag–Menge den übergebenen Wert von *Tag* enthält, wird zurückgeliefert.

Bei Umgebungen und Wahlpunkten kann die jeweilige Liste einfach durch Manipulation von E beziehungsweise B geändert werden. Da in den Prozedurbeschreibungsblöcken auch Information über den Zustand der Liste der Implikationspunkte gespeichert ist, muß diese Information bei Änderung dieser Liste mitverwaltet werden. Eine Operation, die das leistet, ist:

```
procedure LinkImpPts(NewIpt : ↑ ···);
  if IP > NewIpt then
    while IP > NewIpt do
      Proc ← IP↑.Proc;
      Proc↑.IP ← Proc↑.IP↑.IpProc;
      IP ← IP↑.IP;
    od;
  else
    SaveIpt ← NewIpt;
    while NewIpt > IP do
      Proc ← NewIpt↑.Proc;
      if NewIpt > Proc↑.IP then
        Proc↑.IP ← NewIpt;
      fi;
      NewIpt ← NewIpt↑.IP;
    od;
    IP ← SaveIpt;
  fi;
```

Ist der Wert des Implikationspunktzeigers *IP* größer als der angestrebte Wert *NewIpt*, so müssen Implikationspunkte gelöscht werden.
Solange noch zu löschende Implikationspunkte vorhanden sind, wird in dem Prozedurbeschreibungsblock des gerade betrachteten Implikationspunktes der Anfang der Liste der Implikationspunkte für diese Prozedur um einen Implikationspunkt zurückgesetzt.
Andernfalls werden Implikationspunkte wieder sichtbar gemacht.
Solange noch nicht alle wiederherzustellenden Implikationspunkte betrachtet sind, wird in den Prozedurbeschreibungsblock der letzte der Implikationspunkte für diese Prozedur eingetragen.
Anschließend wird der Implikationspunktzeiger *IP* auf den angestrebten Wert gesetzt.

A.3.3 Der Spurstapel

Der Spurstapel dient zum Speichern der existentiell quantifizierten Variablen, deren Bindungen an Objekte bei dem Rücksetzen der Maschine wieder gelöst werden müssen. Die Operation, die überprüft, ob eine Speicherung notwendig ist, und die das eigentliche Speichern vornimmt, ist die folgende:

```
procedure Trail(Var : Obj);
  if (Var.Addr < HB)∨
     (Var.Addr > H)∧
     (Var.Addr < E) then
    TR++↑ ← Var;
  fi;
```

Liegt die Speicherzelle der Variablen *Var* in einem Bereich auf der Halde oder dem Kontrollstapel, der beim Rücksetzen der Maschine nicht sowieso unsichtbar gemacht wird, wird die übergebene Variable an der Spitze des Spurstapels gespeichert.

Neben dem Speichern müssen beim Rücksetzen der Maschine auch Bindungen von Variablen

wieder rückgängig gemacht werden. Dieses leistet:

```
procedure UndoBindings(NewTr : ↑···);
  while TR > NewTr do
    TR−−;
    TR↑.Addr↑ ← TR↑;
  od;
```

Alle Variablen auf dem Spurstapel, die über dem angestrebten Pegel *NewTr* liegen, werden an die Speicherzelle, in der sie erzeugt wurden, zurückgeschrieben.

A.4 Maschinenoperationen

In diesem Abschnitt werden neben der Rücksetzoperation noch ein paar andere Operationen vorgestellt, die die weitere Beschreibung der Maschine vereinfachen. Zunächst eine Operation, die die Behandlung von existentiell quantifizierten Variablen erleichtert:

```
function AssignVal(Obj : Obj) : Obj;
  if Obj.Tag = Free then
    return ⟨Ref, Obj.Addr⟩;
  else
    return Obj;
  fi;
```

Macht aus *Obj* einen kopierbaren Wert: Ist *Obj* eine existentiell quantifizierte Variable, so wird eine Referenz auf diese Variable zurückgegeben, sonst das übergebene Objekt selbst.

Zwei weitere Funktionen, die die Darstellung bestimmter Sachverhalte zu erleichtern, kommen jetzt:

```
function EnvSize(CP : ↑···) : integer;
  ···;
```

Diese Funktion berechnet aus dem Wert des Fortsetzungspunktregisters die Anzahl der permanenten Variablen der Umgebung der Prozedur, in die *CP* zeigt.

```
function NextInst(P : ↑···) : ↑···;
  ···;
```

Liefert zu einem Zeiger in den Codebereich der Maschine die Adresse der nächsten Instruktion.

Eine der zentralen Operationen der I–WAM ist das Rücksetzen der Maschine in einen Zustand, der im letzten Wahlpunkt vermerkt wurde:

```
procedure DoFail();
  while B↑.DynAlt = B↑.StatAlt = nil do
    B ← B↑.B;
  od;
```

Solange kein Wahlpunkt gefunden wurde, der entweder eine statische oder eine dynamische Alternative besitzt, bei der die Abarbeitung fortgesetzt werden kann, wird der nächste Wahlpunkt zur Wiederherstellung des Maschinenzustandes herangezogen.

Code	Kommentar
$E \leftarrow B\uparrow.E$; $CP \leftarrow B\uparrow.CP$; $UndoBindings(B\uparrow.TR)$; $H \leftarrow HB$; $LinkImpPts(B\uparrow.IP)$; $NL \leftarrow B\uparrow.NL$;	Sukzessive werden die Werte der Register E, CP, TR, H, IP und NL wieder hergestellt. Sowohl die Bindungen werden gelöst als auch die Implikationspunkte aktualisiert.
$Start \leftarrow \max\{E + EnvSize(CP), B.B, IP\}$; **for** $ArgCt \leftarrow 1$ **to** $LastArg(B) - Start$ **do** $\mathtt{A}[ArgCt] \leftarrow (Start + ArgCt)\uparrow$; **od**;	Nachdem in *Start* die Adresse des ersten gespeicherten Argumentregisters berechnet wurde, werden alle gespeicherten Werte wieder in die Argumentregister übertragen.

if $B\uparrow.DynAlt \neq$ **nil then**
  $P \leftarrow B\uparrow.DynAlt\uparrow.Label$;
  $EIP \leftarrow B\uparrow.DynAlt\uparrow.E$;
  $B\uparrow.DynAlt \leftarrow SearchIpt(B\uparrow.DynAlt\uparrow.IpProc, DeRef(\mathtt{A}[1]).Tag)$;
	Ist noch eine dynamische Alternative vorhanden, so wird der Instruktionszeiger P auf die erste Instruktion des Codestückes und der Implikationspunktumgebungszeiger auf den im Implikationspunkt gespeicherten Wert gesetzt. Anschließend wird die nächste dynamische Alternative, die zum Objekttyp des ersten Argumentregisters paßt, gesucht und in den Wahlpunkt eingetragen.

else $P \leftarrow B\uparrow.StatAlt$; **fi**;	Gibt es keine dynamische Alternative, wird die Ausführung bei der im Wahlpunkt vermerkten statischen Alternative fortgesetzt.

A.5 Operationen zur Unifikation

Nach dem Bereitstellen aller Operationen, die zum Umgang mit Termen nötig sind, können die Algorithmen, die die Unifikation realisieren, vorgestellt werden. Zur Vereinfachung ist nur der Fall berücksichtigt, daß die *korrekte* Unifikation mit Occur–Check erfolgt.

procedure $Bind(Var, Term : \mathrm{Obj})$;

Code	Kommentar
$Trail(Var)$; $Var.Addr\uparrow \leftarrow Term$;	Die existentiell quantifizierte Variable *Var* wird gesichert und mit *Term* überschrieben.

procedure $Unify(TermA, TermB : \mathrm{Obj})$;

Code	Kommentar
$TermA \leftarrow DeRef(TermA)$; $TermB \leftarrow DeRef(TermB)$;	Durch Dereferenzieren werden die zu unifizierenden Datenobjekte erhalten.
if $TermA \neq TermB$ **then** **case** $\langle TermA.Tag, TermB.Tag\rangle$ **of**	Nur wenn die beiden Datenobjekte nicht gleich sind, muß je nach den Typen der beiden Objekte unterschieden werden.

```
⟨Free, Free⟩:
  if TermA.Addr > TermB.Addr then
    Trail(TermA);
    TermA.Addr↑ ← ⟨Ref, TermB.Addr⟩;
    if TermA.Nest < TermB.Nest then
      Trail(TermB);
      TermB.Addr↑.Nest ← TermA.Nest;
    fi;
  else
    Trail(TermB);
    TermB.Addr↑ ← ⟨Ref, TermA.Addr⟩;
    if TermA.Nest > TermB.Nest then
      Trail(TermA);
      TermA.Addr↑.Nest ← TermB.Nest;
    fi;
  fi;
```

Sind beide Objekte existentiell quantifizierte Variablen, so wird die Variable, die in der Speicherzelle mit der höheren Adresse steht, mit einer Referenz auf die andere Variable überschrieben. Vor dem Überschreiben wird die auf den Spurstapel gesichert.
Besitzt die nicht überschriebene Variable einen geringeren Schachtelungsindex, so wird auch diese Variable auf dem Spurstapel gesichert, und der geringere Wert der Schachtelungstiefe eingetragen.

```
⟨Free, All⟩:
  if TermA.Nest < TermB.Nest then
    DoFail();
  else
    Bind(TermA, TermB);
  fi;
⟨All, Free⟩:
  ···;
```

Ist ein Datenobjekt eine existentiell quantifizierte Variable, die an eine allquantifizierte Variable zu binden ist, so wird die Bindung nur dann ausgeführt, wenn der Schachtelungsindex der freien Variablen nicht kleiner als der Schachtelungsindex der allquantifizierten Variablen ist.
Andernfalls wird Rücksetzen ausgelöst.

```
⟨Free, Struct⟩:
  Arity ← GetArity(TermB.Addr↑);
  Occurs(TermA, TermB.Addr + 1,
            Arity);
  Bind(TermA, TermB);
⟨Struct, Free⟩:
  ···;
```

Nach dem Durchführen des Occur–Check für alle Argumente des Strukturobjektes wird die freie Variable an dieses Objekt gebunden.

```
⟨Free, List⟩:
  Occurs(TermA, TermB.Addr, 2);
  Bind(TermA, TermB);
⟨List, Free⟩:
  ···;
```

Analog zu dem oben geschilderten Fall, spezialisiert für den Listenfunktor.

```
⟨Free, Const⟩:
  Bind(TermA, TermB);
⟨Const, Free⟩:
  ···;
```

Eine existentiell quantifizierte Variable wird einfach an eine Konstante gebunden.

```
  ⟨Struct, Struct⟩:
    ArgsA ← TermA.Addr;
    ArgsB ← TermB.Addr;
    if ArgsA↑ ≠ ArgsB↑ then
      DoFail();
    else
      for ArgCt ← 1 to GetArity(TermA) do
        Unify((++ArgsA)↑, (++ArgsB)↑);
      od;
    fi;
```

Die Unifikation zweier Strukturobjekte verlangt die Gleichheit der Funktoren und die Unifizierbarkeit der Argumente.

```
  ⟨List, List⟩:
    Unify(TermA.Addr↑,
          TermB.Addr↑);
    Unify((TermA.Addr + 1)↑,
          (TermB.Addr + 1)↑);
```

Bei zwei Listenobjekten werden die beiden Argumente unifiziert.

```
  otherwise
    DoFail();
  esac;
fi;
```

Alle anderen Fälle führen zum Rücksetzen. Diese Fälle sind entweder nicht unifizierbare Objekttypen (z.B. Listen und Konstanten) oder gleichartige Objekte mit unterschiedlichen Werten (z.B. zwei unterschiedliche allquantifizierte Variablen).

Den normalen Occur–Check, der das Vorkommen einer existentiell quantifizierten Variablen in einem Term überprüft, die Schachtelungsindices weiterer existentiell quantifizierter Variablen der vorzunehmenden Bindung anpaßt und die vorkommenden allquantifizierten Variablen einschränkt, realisiert die nächste Operation.

procedure *Occurs*(*Var* : Obj; *Args* : ↑Obj; *Arity* : **integer**);

```
for ArgCt ← 1 to Arity do
  Arg ← DeRef(Args++↑);
  case Arg.Tag of
```

Für jedes Argument wird zuerst der eigentliche Wert durch Dereferenzieren festgestellt.

```
   Free:
    if Var = Arg then
      DoFail();
    else
      if Var.Nest < Arg.Nest then
        Trail(Arg);
        Arg.Addr↑.Nest ← Var.Nest;
      fi;
    fi;
```

Ist die Variable *Var* gleich dem Argument *Arg*, so wird Rücksetzen ausgelöst. Ist der Schachtelungsindex der Variablen *Var*, an die der untersuchte Term gebunden werden soll, kleiner als der Schachtelungsindex der in dem Term auftretenden Variablen *Arg*, so wird dieser Schachtelungsindex der vorzunehmenden Bindung angepaßt.

```
   All:
    if Var.Nest < Arg.Nest then
      DoFail();
    fi;
```

Enthält der Term eine allquantifizierte Variable, deren Schachtelungsindex größer als der der existentiell quantifizierten Variablen *Var* ist, wird Rücksetzen ausgelöst.

```
    Struct:
      Occurs(Var, Arg.Addr + 1,
              GetArity(Arg.Addr↑));
```

Der Occur–Check wird bei den Argumenten eines Strukturobjektes fortgesetzt.

```
    List:
      Occurs(Var, Arg.Addr, 2);
```

Hier gilt das oben gesagte, spezialisiert für den Fall des Listenfunktors.

```
    Const:
      ;
  esac;
od;
```

Konstanten bedürfen als unkritische Objekte beim Occur–Check keiner weiteren Behandlung.

Ein Spezialfall des Occur–Check für die Überprüfung von Termen bei der Unifikation von Kopfargumenten im Schreibmodus ist die Operation *WriteOccurs*. Es werden die Schachtelungsindices von existentiellen und universellen Variablen geprüft und der Aufbau von zyklischen Termen verhindert.

```
procedure WriteOccurs(Loc : ↑Obj; Nest : integer; Term : Obj);
  Term ← DeRef(Term);
  case Term.Tag of
```

Der eigentliche Wert des Terms wird durch Dereferenzieren festgestellt.

```
   Free:
     if Term.Nest > Nest then
       Trail(Term);
       Term.Nest ← Nest;
     fi;
```

Es wird geprüft, ob die Variable einen zu hohen Schachtelungsindex besitzt. Ist dem so, wird die Variable auf den Spurstapel gesichert und der Schachtelungsindex angepaßt.

```
   All:
     if Term.Nest > Nest then
       DoFail();
     fi;
```

Es wird geprüft, ob der Schachtelungsindex der allquantifizierten Variable nicht zu hoch ist.

```
   Struct:
     Arity ← GetArity(Term.Addr↑);
     if Term.Addr + Arity > Loc then
       DoFail();
```

Zeigt die Adresse *Loc*, an der der Term abgelegt werden soll, in den abzulegenden Term, so würde bei Ablage ein zyklischer Term aufgebaut: Es wird Rücksetzen ausgelöst.

```
     else
       for Args ← 1 to Arity do
         WriteOccurs(Loc, (Term.Addr + Args)↑);
       od;
     fi;
```

Andernfalls werden die Argumente des Strukturobjektes untersucht.

```
   List:
     if Term.Addr + 2 > Loc then
       DoFail();
     else
       WriteOccurs(Loc, Term.Addr↑);
       WriteOccurs(Loc, (Term.Addr + 1)↑);
     fi;
```

Wiederum stellt dieser Fall für den Listenfunktor nur eine Spezialisierung des Falles für Strukturobjekte dar.

```
  Const:                                  Hier muß nichts überprüft werden.
    ;
esac;
```

A.6 get–Instruktionen

Der Satz der `get`–Instruktionen, die den Einstieg in die Unifikation der Terme realisieren, die an eine Prozedur übergeben wurden, ist gemäß den neuen Typen von Variablen der I–WAM erweitert.

`get_variable V[n], A[i]`

```
V[n] ← A[i];
```

Der Inhalt des Argumentregisters $\texttt{A}[i]$ wird in $\texttt{V}[n]$ kopiert.

`get_value V[n], A[i]`

```
Unify(V[n], A[i]);
```

$\texttt{V}[n]$ und $\texttt{A}[i]$ werden miteinander unifiziert.

`get_constant C, A[i]`

```
Tmp ← DeRef(A[i]);
if Tmp.Tag ≠ Free then
  Bind(Tmp, C);
else
  if Tmp ≠ C then
    DoFail();
  fi;
fi;
```

Der Wert von $\texttt{A}[i]$ wird dereferenziert und die Konstante `C` an das dereferenzierte Objekt gebunden, falls dieses eine freie Variable ist. Ist dieses weder eine freie Variable noch die Konstante `C`, so wird Rücksetzen ausgelöst.

`get_structure F, A[i]`

```
Tmp ← DeRef(A[i]);
case Tmp.Tag of
  Free:
    WRITING ← true;
    WNL ← Tmp.Nest;
    Bind(Tmp, ⟨Struct, H⟩);
    H++↑ ← F;
```

Ist der dereferenzierte Wert von $\texttt{A}[i]$ eine freie Variable, so wird in den Schreibmodus geschaltet. Der Schreibschachtelungsindex wird mit dem Schachtelungsindex dieser freien Variablen initialisiert, ein Strukturobjekt, das auf den aktuellen Haldenpegel H zeigt, an diese Variable gebunden und auf der Halde der übergebene Funktor `F` abgelegt.

```
  Struct:
    S ← Tmp.Addr;
    if S++↑ = F then
      WRITING ← false;
    else
      DoFail();
    fi;
  otherwise
    DoFail();
esac;
```

Ist dieser Wert ein Strukturobjekt, das auch auf den Funktor F zeigt, wird der Strukturzeiger S initialisiert und in den Lesemodus geschaltet.
Ist der dereferenzierte Wert kein Strukturobjekt oder die Funktoren stimmen nicht überein, so wird Rücksetzen ausgelöst.

`get_list A[i]`

```
Tmp ← DeRef(A[i]);
case Tmp.Tag of
  Free:
    WRITING ← true;
    WNL ← Tmp.Nest;
    Bind(Tmp, ⟨List, H⟩);

  List:
    S ← Tmp.Addr;
    WRITING ← false;
  otherwise
    DoFail();
esac;
```

Ist der dereferenzierte Wert von A[i] eine freie Variable, so wird in den Schreibmodus geschaltet. Der Schreibschachtelungsindex wird mit dem Schachtelungsindex der freien Variable initialisiert und ein Listenobjekt, das auf den aktuellen Haldenpegel H zeigt, an diese Variable gebunden.
Ist dieser Wert ein Listenobjekt, so wird der Strukturzeiger initialisiert und in den Lesemodus geschaltet. Ist der dereferenzierte Wert kein Listenobjekt, wird Rücksetzen ausgelöst.

`get_shared_variable V[n], SY[m]`

```
V[n] ← AssignVal(SY[m]);
```

Der *Wert* der geteilten permanenten Variablen SY[m] wird in die Variable V[n] kopiert.

`get_shared_value V[n], SY[m]`

```
Unify(V[n],SY[m]);
```

V[n] und SY[m] werden unifiziert.

A.7 `unify`–Instruktionen

Auch die `unify`–Instruktionen, die die Unifikation verschachtelter Terme bei der Parameterübergabe realisieren, sind bei dieser Maschine gegenüber denen einer normalen WAM modifiziert.

`unify_variable V[n]`

```
if WRITING then
  V[n] ← ⟨Ref, H⟩;
  H++↑ ← ⟨Free, H, WNL⟩;
else
  V[n] ← AssignVal(S++↑);
fi;
```

Ist der Schreibmodus eingeschaltet, wird auf der Halde eine freie Variable mit dem Schachtelungsindex *WNL* erzeugt. In V[n] wird eine Referenz auf diese Variable abgelegt.
Sonst wird in der Variable V[n] das durch *AssignVal* kopierbar gemachte Objekt, auf das der Strukturzeiger S zeigt, abgelegt und der Strukturzeiger um ein Objekt weiterbewegt.

`unify_void`

```
if WRITING then
  H++↑ ← ⟨Free, H, WNL⟩;
else
  S++;
fi;
```

Im Schreibmodus wird auf der Halde eine existentiell quantifizierte Variable mit dem Schachtelungsindex *WNL* erzeugt; im Lesemodus wird der Strukturzeiger S um ein Objekt weiterbewegt.

`unify_value V[n]`

```
if WRITING then
  WriteOccurs(H, WNL, V[n]);
  H++↑ ← V[n];
else
  Unify(S++↑, V[n]);
fi;
```

Im Schreibmodus wird überprüft, daß durch den Term in V[n] auf der Halde kein zyklischer Term entsteht. Nach erfolgreicher Überprüfung wird der Term an die Spitze der Halde kopiert.
Im Lesemodus wird das vom Strukturzeiger S referenzierte Objekt mit V[n] unifiziert.

`unify_local_value V[n]`

```
if WRITING then
  Tmp ← DeRef(V[n]);
  if Tmp.Tag = Free then
    if Tmp.Addr > H then
      Trail(Tmp);
      H↑ ← ⟨Free, H, min{WNL, Tmp.Nest}⟩;
      Tmp.Addr↑ ← ⟨Ref, H⟩;
      H++;
```

Im Schreibmodus wird V[n] dereferenziert. Ist der Wert von V[n] eine freie Variable, die sich auf dem Kontrollstapel befindet, wird diese Variable auf den Spurstapel gesichert. Dann wird eine neue freie Variable auf der Halde erzeugt, die einen eventuell an den Schreibschachtelungsindex *WNL* angepaßten Schachtelungsindex enthält. Die bisherige Variable wird durch eine Referenz auf diese neu erzeugte Variable ersetzt und der Haldenpegel um eins erhöht.

```
    else
      if Tmp.Nest > WNL then
        Trail(Tmp);
        Tmp.Addr↑.Nest ← WNL;
      fi;
      H++↑ ← ⟨Ref, Tmp.Addr⟩;
    fi;
  else
    WriteOccurs(H, Tmp);
    H++↑ ← Tmp;
  fi
else
  Unify(S++↑, V[n]);
fi;
```

Ist der Wert jedoch eine freie Variable auf der Halde, so wird der Schachtelungsindex an den Schreibschachtelungsindex *WNL* angepaßt, und dann eine Referenz auf diese Variable an der Haldenspitze abgelegt. Anschließend wird noch der Haldenpegel um eins fortbewegt.

Ist der Wert keine freie Variable, so wird dieser Wert mit der Operation *WriteOccurs* überprüft, an die Spitze der Halde kopiert, und der Haldenpegel um eins erhöht.

Im Lesemodus wird das vom Strukturzeiger *S* referenzierte Objekt mit V[n] unifiziert.

`unify_constant C`

```
if WRITING then
  H++↑ ← C;
else
  Tmp ← DeRef(S++↑);
  if Tmp.Tag = Free then
    Bind(Tmp, C);
  else
    if Tmp ≠ C then
      DoFail();
    fi;
  fi;
fi;
```

Im Schreibmodus wird die Konstante C an die Haldenspitze kopiert.

Im Lesemodus wird das von *S* bezeichnete Objekt dereferenziert. Ist der Wert eine freie Variable, wird diese an die Konstante C gebunden. Ist dieser keine freie Variable, jedoch gleich der Konstanten, so wird fortgefahren, andernfalls Rücksetzen ausgelöst. In jedem Fall wird der Strukturzeiger um ein Objekt fortgeschaltet.

A.8 put–Instruktionen

Hier gibt es neben den neuen Instruktionen zur Realisierung der Übergabe der beiden Variablentypen auch solche, die der Erzeugung von geteilten Variablen dienen.

`put_variable X[n], A[i]`

```
X[n] ← ⟨Ref, H⟩;
A[i] ← ⟨Ref, H⟩;
H++↑ ← ⟨Free, H, NL⟩;
```

Erzeugt eine neues Variablenobjekt auf der Halde. Sowohl in das temporäre Register X[n] als auch in das Argumentregister A[i] wird eine Referenz auf dieses Variablenobjekt geschrieben.

`put_variable Y[n], A[i]`

$\texttt{Y}[n] \leftarrow \langle \text{Free}, \texttt{Y}[n]{\downarrow}, NL \rangle;$
$\texttt{A}[i] \leftarrow \langle \text{Ref}, \texttt{Y}[n]{\downarrow} \rangle;$

Erzeugt ein neues Variablenobjekt in der permanenten Variable Y[n] auf dem Kontrollstapel und erzeugt in dem Argumentregister A[i] eine Referenz auf dieses.

`put_variable_offset X[n], A[i], O`

$\texttt{X}[n] \leftarrow \langle \text{Ref}, H \rangle;$
$\texttt{A}[i] \leftarrow \langle \text{Ref}, H \rangle;$
$H{+}{+}{\uparrow} \leftarrow \langle \text{Free}, H, NL - \texttt{O} \rangle;$

Erzeugt eine neues Variablenobjekt auf der Halde, dessen Schachtelungsindex mit der Differenz des aktuellen Schachtelungsindex *NL* und dem angegebenen Offset O initialisiert wird. Sowohl in X[n] als auch in das Argumentregister A[i] wird eine Referenz auf dieses Variablenobjekt geschrieben.

`put_variable_offset Y[n], A[i], O`

$\texttt{Y}[n] \leftarrow \langle \text{Free}, \texttt{Y}[n]{\downarrow}, NL - \texttt{O} \rangle;$
$\texttt{A}[i] \leftarrow \langle \text{Ref}, \texttt{Y}[n]{\downarrow} \rangle;$

Erzeugt ein neues Variablenobjekt, dessen Schachtelungsindex mit der Differenz des aktuellen Schachtelungsindexes *NL* und dem angegebenen Offset O initialisiert wird, in der permanenten Variable Y[n] auf dem Kontrollstapel und legt in dem Argumentregister A[i] eine Referenz auf dieses Objekt ab.

`put_variable_all V[n], A[i]`

$\texttt{V}[n] \leftarrow \mathit{GenAll}(NL);$
$\texttt{A}[i] \leftarrow \texttt{V}[n];$

Erzeugt eine neue allquantifizierte Variable mit eindeutiger Identifikationsnummer in V[n] und kopiert diese in das Argumentregister A[i].

`put_variable_all_offset V[n], A[i], O`

$\texttt{V}[n] \leftarrow \mathit{GenAll}(NL - \texttt{O});$
$\texttt{A}[i] \leftarrow \texttt{V}[n];$

Erzeugt ein neue allquantifizierte Variable, die einen negativen Offset von O zum aktuellen Wert des Registers *NL* besitzt, in V[n]. Anschließend wird diese auch in das Argumentregister A[i] kopiert.

`put_void A[i]`

$\texttt{A}[i] \leftarrow \langle \text{Ref}, H \rangle;$
$H{+}{+}{\uparrow} \leftarrow \langle \text{Free}, H, NL \rangle;$

Legt im Argumentregister A[i] eine Referenz auf eine an der Haldenspitze neu erzeugte existentielle Variable an.

`put_void_offset A[i], O`

$\mathtt{A}[i] \leftarrow \langle \text{Ref}, H \rangle$;
$H{+}{+}\!\uparrow \leftarrow \langle \text{Free}, H, NL - \mathtt{O} \rangle$;

Legt im Argumentregister A[i] eine Referenz auf eine an der Haldenspitze neu erzeugte existentielle Variable an, die als Schachtelungsindex den aktuellen Wert von *NL*, vermindert um den Offset O, enthält.

`put_void_all A[i]`

$\mathtt{A}[i] \leftarrow \mathit{GenAll}(NL)$;

Legt im Argumentregister A[i] eine neue universelle Variable an.

`put_void_all_offset A[i], O`

$\mathtt{A}[i] \leftarrow \mathit{GenAll}(NL - \mathtt{O})$;

Legt im Argumentregister A[i] eine neue universelle Variable an, die als Schachtelungsindex die Differenz von dem aktuellen Wert des Registers *NL* und dem Offset O enthält.

`put_value V[n], A[i]`

$\mathtt{A}[i] \leftarrow \mathtt{V}[n]$;

Kopiert V[n] in das Argumentregister A[i].

`put_unsafe_value Y[n], A[i]`

```
Tmp ← DeRef(Y[n]);
if Tmp.Tag = Free else
  if Tmp.Addr < E then
    A[i] ← ⟨Ref, Tmp.Addr⟩;
  else
    H↑ ← ⟨Free, H, Tmp.varNest⟩;
    Tmp.Addr↑ ← ⟨Ref, H⟩;
    A[i] ← ⟨Ref, H⟩;
    H++;
  fi;
else
  A[i] ← Tmp;
fi;
```

Der Wert der permanenten Variablen Y[n] wird dereferenziert.
Ist dieser Wert eine freie Variable, die unterhalb der aktuellen Umgebung liegt, so ist diese *sicher*, und in das Argumentregister A[i] wird eine Referenz auf diese Variable kopiert. Liegt sie jedoch in der aktuellen Umgebung auf dem Kontrollstapel, so wird eine neue Variable mit demselben Schachtelungsindex auf der Halde erzeugt, die alte Variable durch eine Referenz auf die Neue überschrieben und eine Referenz auf diese neue Variable in das Argumentregister A[i] kopiert.
Ist der dereferenzierte Wert von Y[n] keine freie Variable, so wird dieser Wert direkt in das Argumentregister A[i] kopiert.

`put_structure F, A`$[i]$

`A`$[i] \leftarrow \langle \text{Struct}, H \rangle$;
$H{+}{+}\uparrow \leftarrow$ `F`;

In dem Argumentregister `A`$[i]$ wird ein Strukturobjekt erzeugt, das auf die aktuelle Haldenspitze zeigt. Auf der Halde wird der Funktor `F` abgelegt und der Haldenpegel um eins erhöht.

`put_list A`$[i]$

`A`$[i] \leftarrow \langle \text{List}, H \rangle$;

In das Argumentregister `A`$[i]$ wird ein Listenobjekt kopiert, das auf die Haldenspitze zeigt.

`put_constant C, A`$[i]$

`A`$[i] \leftarrow$ `C`;

Die Konstante `C` wird in das Argumentregister `A`$[i]$ kopiert.

`put_shared Y`$[n]$

`Y`$[n] \leftarrow \langle \text{Free}, \mathtt{Y}[n]\downarrow, \mathit{NL} \rangle$;

Erzeugt auf dem Kontrollstapel eine neue existentiell quantifizierte Variable.

`put_shared_all Y`$[n]$

`Y`$[n] \leftarrow \mathit{GenAll}(\mathit{NL})$;

Erzeugt auf dem Kontrollstapel eine neue allquantifizierte Variable.

`put_unsafe_shared Y`$[n]$

`Y`$[n] \leftarrow \langle \text{Ref}, H \rangle$;
$H{+}{+}\uparrow \leftarrow \langle \text{Free}, \mathtt{Y}[n]\downarrow, \mathit{NL} \rangle$;

Erzeugt auf der Halde eine neue existentiell quantifizierte Variable und legt auf dem Kontrollstapel eine Referenz auf diese ab.

Die nächsten drei Instruktionen besitzen die gleiche Funktionalität wie die drei oben angegebenen, sie ziehen beim Erzeugen neuer Variablen jedoch noch den Offset `O` zur Ablage des Schachtelungsindex heran: Eine Beschreibung ihrer Funktionalität erübrigt sich also.

`put_shared_offset Y`$[n]$`, O`

`Y`$[n] \leftarrow \langle \text{Free}, \mathtt{Y}[n]\downarrow, \mathit{NL} - \mathtt{O} \rangle$;

`put_shared_all_offset Y`$[n]$`, O`

`Y`$[n] \leftarrow \mathit{GenAll}(\mathit{NL} - \mathtt{O})$;

`put_unsafe_shared_offset Y[n], O`

$\texttt{Y}[n] \leftarrow \langle \text{Ref}, H \rangle;$
$H{+}{+}\uparrow \leftarrow \langle \text{Free}, H, NL - \texttt{O} \rangle;$

A.9 set–Instruktionen

Verschachtelte Terme im Rumpf einer Klausel zum Bereitstellen der Argumente eines Prozeduraufrufes werden durch `set`–Instruktionen erzeugt.

`set_variable V[n]`

$\texttt{V}[n] \leftarrow \langle \text{Ref}, H \rangle;$
$H{+}{+}\uparrow \leftarrow \langle \text{Free}, H, NL \rangle;$

Auf der Halde wird eine neue freie Variable erzeugt, die als Schachtelungsindex den Wert von *NL* erhält, und in V[n] eine Referenz auf diese Variable abgelegt.

`set_variable_all V[n]`

$\texttt{V}[n] \leftarrow \mathit{GenAll}(NL);$
$H{+}{+}\uparrow \leftarrow \texttt{V}[n];$

Es wird ein neue allquantifizierte Variable erzeugt, die sowohl in V[n] als auch an der Haldenspitze abgelegt wird.

`set_variable_offset V[n], O`

$\texttt{V}[n] \leftarrow \langle \text{Ref}, H \rangle;$
$H{+}{+}\uparrow \leftarrow \langle \text{Free}, H, NL - \texttt{O} \rangle;$

An der Haldenspitze wird eine existentiell quantifizierte Variable erzeugt, die den aktuellen Schachtelungsindex vermindert um den Offset O enthält. In V[n] wird eine Referenz auf diese Variable erzeugt.

`set_variable_all_offset V[n], O`

$\texttt{V}[n] \leftarrow \mathit{GenAll}(NL - \texttt{O});$
$H{+}{+}\uparrow \leftarrow \texttt{V}[n];$

Es wird ein neue allquantifizierte Variable erzeugt, die sowohl in V[n] als auch an der Haldenspitze abgelegt wird. Diese enthält als Schachtelungsindex den aktuellen Wert von *NL* vermindert um den Offset O.

`set_void`

$H{+}{+}\uparrow \leftarrow \langle \text{Free}, H, NL \rangle;$

Auf der Halde wird ein neues Variablenobjekt mit dem Schachtelungsindex *NL* erzeugt.

`set_void_all`
$H{+}{+}\uparrow \leftarrow \mathit{GenAll}(\mathit{NL});$

Es wird ein neue allquantifizierte Variable erzeugt, die an der Haldenspitze abgelegt wird.

`set_void_offset O`
$H{+}{+}\uparrow \leftarrow \langle \text{Free}, H, \mathit{NL} - \mathtt{O} \rangle;$

An der Haldenspitze wird eine existentiell quantifizierte Variable erzeugt, die den aktuellen Schachtelungsindex vermindert um den Offset `O` enthält.

`set_void_all_offset O`
$H{+}{+}\uparrow \leftarrow \mathit{GenAll}(\mathit{NL} - \mathtt{O});$

Es wird ein neue allquantifizierte Variable erzeugt, die als Schachtelungsindex den aktuellen Wert von *NL* vermindert um den Offset `O` enthält.

`set_value V`$[n]$
$H{+}{+}\uparrow \leftarrow \mathtt{V}[n];$

Der Term aus `V`$[n]$ wird an die Spitze der Halde kopiert und der Haldenpegel um eins erhöht.

`set_local_value V`$[n]$

```
Tmp ← DeRef(V[n]);
if Tmp.Tag = Free then
  if Tmp.Addr > H then
    Trail(Tmp);
    H↑ ← ⟨Free, H, Tmp.Nest⟩;
    Tmp.Addr↑ ← ⟨Ref, H⟩;
    H++;
```

`V`$[n]$ wird dereferenziert. Ist der Wert eine freie Variable, die sich auf dem Kontrollstapel befindet, so wird diese Variable auf den Spurstapel gesichert. Dann wird eine neue freie Variable mit dem gleichen Schachtelungsindex auf der Halde erzeugt. Die bisherige Variable wird durch eine Referenz auf diese neu erzeugte Variable ersetzt und der Haldenpegel um eins erhöht.

```
  else
    H++↑ ← ⟨Ref, Tmp.Addr⟩;
  fi;
```

Ist der Wert eine freie Variable auf der Halde, so wird eine Referenz auf diese Variable an der Haldenspitze abgelegt.

```
else
  H++↑ ← Tmp;
fi
```

Der Wert wird an die Spitze der Halde kopiert, wenn er keine freie Variable ist, und der Haldenpegel um eins erhöht.

`set_constant C`
$H{+}{+}\uparrow \leftarrow \mathtt{C};$

Die Konstante `C` wird an die Haldenspitze kopiert und der Haldenpegel um eins erhöht.

A.10 Kontrollinstruktionen

Neben den Kontrollinstruktionen der normalen WAM gibt es noch weitere Instruktionen, die die Verwaltung von Allquantoren und dynamischen Klauseln übernehmen.

`call N, L`

$CP \leftarrow \texttt{N}\downarrow$;
$P \leftarrow \texttt{L}$;

Die Rückkehradresse wird in CP gespeichert. Gleichzeitig liefert dann CP auch den Zugriffspfad auf die Anzahl der permanenten Variablen der Prozedurumgebung. Die Ausführung wird bei `L` fortgesetzt.

`execute L`

$P \leftarrow \texttt{L}$;

Die Ausführung wird bei `L` fortgesetzt.

`proceed`

$P \leftarrow \mathit{NextInst}(CP)$;

Die Abarbeitung wird beim Fortsetzungspunkt weitergeführt.

`allocate`

$\mathit{Top} \leftarrow \max\{E + \mathit{EnvSize}(CP), B, IP\}$;
$\mathit{Top}{+}{+}\uparrow \leftarrow E$;
$\mathit{Top}{+}{+}\uparrow \leftarrow CP$;
$E \leftarrow \mathit{Top}$;

Eine neue Prozedurumgebung wird angelegt. Dazu wird an der Spitze des Kontrollstapels der Fortsetzungspunkt und der Zeiger auf die vorherige Prozedurumgebung gespeichert. Anschließend wird E auf die neue Prozedurumgebung gesetzt.

`deallocate`

$CP \leftarrow (--E)\uparrow$;
$E \leftarrow (--E)\uparrow$;

Die letzte Prozedurumgebung wird entfernt, dadurch wird die vorherige wieder sichtbar und der Fortsetzungspunkt restauriert.

`quantify`

$NL{+}{+}$;

Erhöht den Schachtelungsindex um eins.

`dequantify`

$NL{-}{-}$;

Erniedrigt den Schachtelungsindex um eins.

`activate N, Len, List`

$Top \leftarrow \max\{E + \texttt{N}, B, IP\}$;
for $ClauseCt \leftarrow 1$ **to** `Len` **do**
  $\langle Proc, TagSet, Label\rangle \leftarrow \texttt{List}[ClauseCt]$;
  $Top{+}{+}\uparrow \leftarrow IP$;
  $Top{+}{+}\uparrow \leftarrow Proc\uparrow.IP$;
  $Top{+}{+}\uparrow \leftarrow E$;
  $Top{+}{+}\uparrow \leftarrow Proc$;
  $Top{+}{+}\uparrow \leftarrow TagSet$;
  $Top{+}{+}\uparrow \leftarrow Label$;
  $IP \leftarrow Top$;
  $Proc\uparrow.IP \leftarrow Top$;
od;

Die Adresse der ersten freien Speicherzelle auf dem Kontrollstapel wird berechnet. Dann werden sukzessive für alle `Len` zu aktivierenden dynamischen Klauseln Implikationspunkte angelegt. Die nötigen Informationen dazu werden aus `List` entnommen.
Die Implikationspunkte werden durch das Ablegen von *IP* und für jede einzelne Prozedur, zu denen die aktivierten Klauseln gehören, durch Aktualisierung des entsprechenden Wertes $Proc\uparrow.IP$ im Prozedurbeschreibungsblock verkettet.

`deactivate Len`

for $ClauseCt \leftarrow 1$ **to** `Len` **do**
  $IP\uparrow.Proc \leftarrow IP\uparrow.Proc\uparrow.IP$;
  $IP \leftarrow IP\uparrow.IP$;
od;

Sowohl in den Prozedurbeschreibungsblöken als auch im Implikationspunktregister werden die letzten `Len` Implikationspunkte ausgekettet.

A.11 Indexinstruktionen

Die Indexinstruktionen, die einzelne Klauseln zu Prozeduren verbinden, sind die letzten, die noch beschrieben werden müssen.

`try_dynamic N`

$Tag \leftarrow DeRef(\texttt{A[1]}).Tag$;
$DynAlt \leftarrow SearchImpPt(PROC\uparrow.IP, Tag)$;
$NextDynAlt \leftarrow SearchIpt(DynAlt\uparrow.IpProc, Tag)$;
if $DynAlt \neq$ **nil then**
  $MkChoicePt(\texttt{N}, NextInst(P), NextDynAlt)$;
  $P \leftarrow DynAlt\uparrow.Label$;
  $EIP \leftarrow DynAlt\uparrow.E$;
else
  $MkChoicePt(\texttt{N}, \textbf{nil}, \textbf{nil})$;
fi;

Die ersten beiden Implikationspunkte werden in *DynAlt*, respektive *NextDynAlt* für die aktuelle Prozedur *PROC*, die zum Objekttyp des ersten Argumentes passen, berechnet.
Gibt es eine dynamische Alternative, so wird ein Wahlpunkt angelegt, der die folgende Instruktion als statische und den nächsten Implikationspunkt als dynamische Alternative enthält. Das Register *EIP* als Zugriffspfad für die geteilten Variablen wird initialisiert und die Ausführung bei dieser dynamischen Alternative fortgesetzt.
Gibt es keine, so wird ein Wahlpunkt angelegt, der weder eine statische noch eine dynamische Alternative enthält, und die Abarbeitung bei der nächsten Instruktion fortgesetzt.

`trust_dynamic N`

$Tag \leftarrow DeRef(\mathtt{A}[1].Tag)$;
$DynAlt \leftarrow SearchImpPt(PROC\uparrow.IP, Tag)$;
if $DynAlt =$ **nil then**
  $DoFail()$;
fi;
$NextDynAlt \leftarrow SearchIpt(DynAlt\uparrow.IpProc, Tag)$;
if $NextDynAlt \neq$ **nil then**
  $MkChoicePt(\mathtt{N}, \mathbf{nil}, NextDynAlt)$;
fi;
$P \leftarrow DynAlt\uparrow.Label$;
$EIP \leftarrow DynAlt\uparrow.E$;

Wenn es keine dynamische Alternative *DynAlt* für die aktuelle Prozedur *PROC* gibt, so wird Rücksetzen ausgelöst. Nur wenn feststeht, daß es noch eine weitere dynamische Alternative *NextDynAlt* gibt, wird auch wirklich ein Wahlpunkt angelegt. Dieser enthält keine statische Alternative, jedoch den zweiten Implikationspunkt *NextDynAlt* als dynamische Alternative.
Das Register *EIP* wird als Zugriffspfad für die geteilten Variablen initialisiert und die Programmausführung bei der dynamischen Alternative fortgesetzt.

`try N, L`

$B \leftarrow MkChoicePt(\mathtt{N}, NextInst(P), \mathbf{nil})$;
$P \leftarrow \mathtt{L}$;

Ein neuer Wahlpunkt wird auf dem Kontrollstapel eingerichtet. Er enthält kein dynamische Alternative. Als statische Alternative wird die Adresse der nachfolgenden Instruktion in dem Wahlpunkt abgelegt.
Dann findet ein Sprung nach `L` statt.

`retry L`

$B\uparrow.StatAlt \leftarrow NextInst(P)$;
$P \leftarrow \mathtt{L}$;

In den aktuellen Wahlpunkt wird als statische Alternative die nächste Instruktion eingetragen. Anschließend findet ein Sprung nach `L` statt.

`trust L`

$HB \leftarrow B\uparrow.H$;
$B \leftarrow B\uparrow.B$;
$P \leftarrow \mathtt{L}$;

Der Haldenrücksetzpegel *HB* wird auf den im Wahlpunkt gespeicherten Wert zurückgesetzt und der vorhergehende Wahlpunkt wird durch Rücksetzen des Wahlpunktzeigers *B* sichtbar gemacht. Anschließend findet ein Sprung nach `L` statt.

`switch_on_type LV, LS, LL, LC`

case *DeRef*(`A[1]`).*Tag* **of**
  Free, All: $P \leftarrow$ `LV`;
  Struct: $P \leftarrow$ `LS`;
  List: $P \leftarrow$ `LL`;
  Const: $P \leftarrow$ `LC`;
esac;
if $P =$ **nil then**
  DoFail();
fi;

Setzt je nach Typ des durch Dereferenzieren erhaltenen Wertes des ersten Argumentes die Abarbeitung fort.
Ist der ausgewählte Punkt zum Fortsetzen der Ausführung nicht vorhanden, wird Rücksetzen ausgelöst.

Die eigentliche Instruktion, die je nach Typ des Argumentes den Kontrollfluß steuert, ist die nächste, die vorhergehende ist aus Gründen der Abwärtskompatibilität vorhanden.

`switch_on_ext_type LE, LA, LS, LL, LC`

case *DeRef*(`A[1]`).*Tag* **of**
  Free: $P \leftarrow$ `LE`;
  All: $P \leftarrow$ `LA`;
  Struct: $P \leftarrow$ `LS`;
  List: $P \leftarrow$ `LL`;
  Const: $P \leftarrow$ `LC`;
esac;
if $P =$ **nil then**
  DoFail();
fi;

Setzt je nach Typ des durch Dereferenzieren erhaltenen Wertes des ersten Argumentes die Programmausführung fort, wobei diese Instruktion auch den Typ der allquantifizierten Variablen berücksichtigt.
Ist der ausgewählte Punkt zum Fortsetzen der Ausführung nicht vorhanden, wird Rücksetzen ausgelöst.

Anhang B

Sprachdefinition und Bedienungsanleitung des KAIN–PROLOG Systems

Dieser Anhang beschreibt die Implementierung des KAIN–PROLOG Systems aus Benutzersicht. Im wesentlichen sind nur solche Aspekte beschrieben, die sich im Vergleich zum KA–PROLOG System geändert haben. Die unveränderten Aspekte lassen sich der Sprachdefinition von KA–PROLOG [BBL+91] und der Benutzeranleitung von KA–PROLOG [BBB+91] entnehmen.

B.1 Modulstruktur

Genau wie in KA–PROLOG sind Programme in KAIN–PROLOG aus Modulen zusammengesetzt. Ein Modul besteht aus einem *Deklarationsteil* und einem *Implementierungsteil*. Der Aufbau eines Moduls läßt sich mit der folgenden Grammatik in EBNF–Form [WG85, subsection 5.1.4] beschreiben.

Modul ::= `module` *Modulname* `.`
Deklaration*
`body.`
Implementierung
`end_module.`

Terminalsymbole der Grammatik sind in der Form `Terminalsymbol`, Nichtterminalsymbole in der Form *Nichtterminalsymbol* gekennzeichnet. Die Sätze der Grammatik sind Terme nach der üblichen Termgrammatik, diese ist genauso definiert wie für KA–PROLOG [BBL+91]. Für alle weiteren Grammatiken, die in diesem Anhang angegeben werden, gilt das hier gesagte.

B.1.1 Deklaration

Die Deklarationen eines Moduls bestehen zum einen aus Operatordeklarationen, die wie üblich die Aufgabe haben, die Lesbarkeit von Termen zu erhöhen. Zum anderen gibt es Sichtbarkeits-

Tabelle B.1: Operatordefinitionen zur Übersetzungszeit

Operator	Präzedenz	Typ	Bedeutung
`:-`	1200	`xfx`	$\leftarrow$
`-:`	1150	`xfy`	$\rightarrow$
`;`	1100	`xfy`	$\vee$
`->`	1050	`xfy`	if–then–else
`all`	950	`xfy`	$\forall$
`exists`	950	`xfy`	$\exists$
`:`	950	`xfy`	Trennzeichen für $\forall$, $\exists$
`not`	900	`fy`	Nichtableitbarkeit

deklarationen, durch sie wird bestimmt, welche Prozeduren in welchen Modulen sichtbar sind. Damit ergibt sich die folgende Grammatik:

$$\begin{array}{rcl} \textit{Deklaration} & ::= & \textit{Operatordeklaration} \\ & | & \textit{Sichtbarkeitsdeklaration} \end{array}$$

B.1.1.1 Operatordeklaration

Die Form der Operatordeklarationen ist im Vergleich zum KA–PROLOG System ungeändert. Eine Änderung ergibt sich nur daraus, daß es andere vorgegebene Operatordeklarationen zur Übersetzungszeit gibt. Die Operatordeklarationen, die sowohl zur Übersetzungs- als auch zur Laufzeit vorgegeben sind, gelten nach wie vor. Die Operatoren und deren Definition, die nur zur Übersetzungszeit vorgegeben sind, sind Tabelle B.1 zu entnehmen.

B.1.1.2 Sichtbarkeitsdeklaration

Die Möglichkeiten zur Sichtbarkeitsdeklaration sind gleich geblieben, geändert haben sich jedoch teilweise die Bedeutung der Deklarationen. Nach wie vor müssen diese Deklarationen der folgenden Grammatik entsprechend aufgebaut sein:

$$\begin{array}{rcl} \textit{Sichtbarkeitsdeklaration} & ::= & \texttt{private}\ \textit{Funktorliste}\ \texttt{.} \\ & | & \texttt{export}\ \textit{Funktorliste}\ \texttt{.} \\ & | & \texttt{from}\ \textit{Modulname}\ \texttt{import}\ \textit{Funktorliste}\ \texttt{.} \\ \textit{Funktorliste} & ::= & (\textit{Funktorname}\ \texttt{/}\ \textit{Stelligkeit})\ ||\ \texttt{,} \end{array}$$

Funktorname bezeichnet einen Namen nach der Konvention von KA–PROLOG, *Stelligkeit* eine ganze positive Zahl. Im folgenden wird die Bedeutung dieser Deklarationen erklärt.

`private` *Funktorname*`/`*Stelligkeit*`.` Die durch den Funktornamen und die Stelligkeit beschriebene Prozedur kann nur in diesem Modul benutzt werden.

`export` *Funktorname*`/`*Stelligkeit*`.` Durch diese Deklaration wird die Prozedur mit dem entsprechenden Funktornamen und der entsprechenden Stelligkeit nach *außen* sichtbar. Andere Module können diese Prozedur dann mit einer `import`–Deklaration für das jeweilige Modul sichtbar machen.

`from` *Modulname* `import` *Funktorname*`/`*Stelligkeit*`.` Diese Deklaration macht die durch den Funktornamen und die Stelligkeit bezeichnete Prozedur benutzbar, wenn in dem Modul *Modulname* diese Prozedur mittels `export` deklariert ist.

In den obigen Erklärungen wurde jeweils nur die Benutzbarkeit beschrieben. Wie der Zusammenhang zwischen der Deklaration und Implementierung von Prozeduren in einem Modul ist, wird in dem nächsten Abschnitt beschrieben. In analoger Art und Weise können wie in KA–PROLOG auch Datenbankschlüssel deklariert werden. Genaueres zur Behandlung der internen Datenbank findet sich in Abschnitt B.2.1.

Genauso wie es vorgegebene Operatordeklarationen gibt, gibt es auch vorgegebene Sichtbarkeitsdeklarationen. In jedem Modul ist die Prozedur `goal/0` als private Prozedur deklariert. Zusätzlich sind implizit alle Prozeduren des Standardmoduls `public` als importiert deklariert. Siehe dazu auch Abschnitt B.2.1.

B.1.2 Implementierung

Dieser Abschnitt beschäftigt sich mit dem Implementierungsteil eines KAIN–PROLOG Moduls.

B.1.2.1 Klauseln

Der Implementierungsteil eines KAIN–PROLOG Moduls besteht aus Klauseln:

$$\textit{Implementierung} ::= (\textit{Klausel}\,\texttt{.})^*$$

Entscheidend ist natürlich, wie Klauseln aufgebaut sind. Die folgende Grammatik gibt den Aufbau an, sie benutzt dabei die Operatoren aus der Tabelle B.1.

Klausel	::=	`all` *Variablen* `:` *Klausel*
	|	`all` *Klausel*
	|	*Atom* `:-` *Rumpf*
	|	*Atom*
Rumpf	::=	*Rumpf* `,` *Rumpf*
	|	`all` *Variablen* `:` *Rumpf*
	|	`exists` *Variablen* `:` *Rumpf*
	|	(*Klausel* || `,`) `-:` *Rumpf*
	|	*Atom*
	|	!
	|	*Kontrollstruktur*
Variablen	::=	*Variable* || `,`

```
Kontrollstruktur ::= Rumpf ; Rumpf
                   | Rumpf -> Rumpf ; Rumpf
                   | not Rumpf
                   | forall( Rumpf )
                   | backtrack( Rumpf )
                   | no_backtrack( Rumpf )
                   | true
                   | fail
                   | repeat
                   | abort
```

Die Nichtterminalsymbole *Atom* und *Variable* beschreiben einen zusammengesetzten Term nach der üblichen Termgrammatik, beziehungsweise einen Term, der nur aus einem Variablenbezeichner besteht. Ein Rumpf kann auch aus einer Kontrollstruktur bestehen. Dies sind genau die gleichen Kontrollstrukturen, wie sie auch in KA–PROLOG vorgesehen sind.

Nun bleibt noch zu klären, wie der Zusammenhang zwischen der Definition von Prozeduren in dem Implementierungsteil eines Moduls und der Deklaration ist. Je nach ist folgendes möglich:

`private` *Funktorname*`/`*Stelligkeit*`.` Für die durch den Funktornamen und die Stelligkeit beschriebene Prozedur können sowohl statische als auch dynamische Klauseln definiert werden. Dynamische Klauseln sind dabei solche, die verschachtelt in einer anderen Klausel auftreten.

`export` *Funktorname*`/`*Stelligkeit*`.` Hier gilt das oben gesagte.

`from` *Modulname* `import` *Funktorname*`/`*Stelligkeit*`.` Es ist erlaubt, für die durch den Funktornamen und die Stelligkeit bezeichnete Prozedur, dynamische Klauseln zu definieren. Es ist jedoch genau wie in KA–PROLOG *nicht* erlaubt, statische Klauseln zu implementieren.

B.1.2.2 Abarbeitung

In diesem Abschnitt wird erklärt, wie die Abarbeitung von Klauseln und Anfragen im KAIN–PROLOG System vor sich geht. Grundlegend ist die Abwicklung eines Prozeduraufrufes, dieser Mechanismus wird zuerst erklärt.

Prozeduraufruf Bei einem Prozeduraufruf wird die erste Klausel der aufgerufenen Prozedur ausgewählt. Kann der Kopf der ausgewählten Klausel mit dem Prozeduraufruf unifiziert werden, wird mit der Abarbeitung des Rumpfes der ausgewählten Klausel fortgesetzt, dies natürlich nur für den Fall, daß die ausgewählte Klausel kein Atom ist. Ist der Kopf der ausgewählten Klausel nicht mit dem Prozeduraufruf unifizierbar, oder schlägt die Abarbeitung der ausgewählten Klausel fehl, so wird in den Zustand der Berechnung vor Auswahl der Klausel zurückgesetzt, und die nächste mögliche Klausel der Prozedur ausgewählt. Die Reihenfolge der ausgewählten Klauseln ist einmal statisch durch die lexikalische Reihenfolge ihres Auftretens in dem Implementierungsteil des Moduls bestimmt. Zum anderen können auch *dynamische Klauseln* vorhanden sein, die

dann zuerst berücksichtigt werden. Wie dynamische Klauseln der Abarbeitung zur Verfügung gestellt werden, wird später erklärt.

$Rumpf_1$, $Rumpf_2$ — **Konjunktion** Die Konjunktion zweier Rümpfe wird wie üblich behandelt: Zuerst wird $Rumpf_1$ abgearbeitet, dann $Rumpf_2$.

`all` *Variablen* `:` *Rumpf* — **Allquantifizierung** Um die Allquantifizierung abzuarbeiten, wird wie folgt vorgegangen: Für jede Variable aus der Liste *Variablen* wird eine neue interne Konstante erzeugt. Alle Auftreten der Variablen in *Rumpf* werden durch diese Konstanten ersetzt. Die erzeugten Konstanten dürfen bei der weiteren Abarbeitung von *Rumpf* jedoch nicht an Variablen gebunden werden, die bisher schon aufgetreten sind.

`exists` *Variablen* `:` *Rumpf* — **Existentielle Quantifizierung** Durch die Abarbeitung eines Existenzquantors werden für alle Variablen aus *Variablen* neue Variablen erzeugt. Die bisherigen Variablen werden durch diese neu erzeugten Variablen in *Rumpf* ersetzt, sie dürfen im weiteren nur an Terme gebunden werden, die keine Konstanten enthalten, die bei der weiteren Abarbeitung von *Rumpf* aus allquantifizierten Variablen hervorgehen.

(*Klausel* || `,`) `-:` *Rumpf* — **Hypothetische Implikation** Eine hypothetische Implikation wird dadurch abgearbeitet, daß die Klauseln der Implikation den bisher vorhandenen Klauseln hinzugefügt werden. Dann wird mit der Abarbeitung von *Rumpf* fortgefahren. Nach Abarbeitung des Rumpfes werden die hinzugefügten Klauseln wieder entfernt. Bei einem Prozeduraufruf werden die zuletzt hinzugefügten Klauseln bei der weiter oben geschilderten Auswahl zuerst berücksichtigt. Dabei werden die Klauseln einer hypothetischen Implikation von links nach rechts betrachtet.

`!` — **Cut, systemdefinierte Kontrollstrukturen** Die Abarbeitung des Cut und der systemdefinierten Kontrollstrukturen erfolgt in der gleichen Art und Weise wie bei KA–PROLOG.

B.2 Standardmodule

In diesem Abschnitt werden vordefinierte Module für das KAIN–PROLOG System beschrieben.

B.2.1 Das Modul `public`

Das Modul `public` enthält verschiedene Standardprädikate. Die angebotenen Prädikate und deren Funktionalität stimmt im wesentlichen mit KA–PROLOG überein. Nur veränderte und neue Prädikate werden in diesem Abschnitt vorgestellt.

Metalogik Die meisten Prozeduren, die die Metalogik betreffen, sind unverändert geblieben. Die zwei verschiedenen Arten von Variablen, die bei der Abarbeitung von Programmen in KAIN–PROLOG auftreten können, schlagen sich jetzt auch in veränderten Prädikaten für die Untersuchung von Termen darauf, ob sie aus Variablen bestehen, nieder. Das bisher schon

vorhandene Prädikat `var(`*Term*`)` ist jetzt erfüllt, wenn *Term* eine existentiell quantifizierte Variable ist. Neu sind die beiden Prädikate `var_any(`*Term*`)` und `var_all(`*Term*`)`. `var_any(`*Term*`)` ist erfüllt, wenn *Term* eine existentiell oder universell quantifizierte Variable ist. Dagegen ist `var_all(`*Term*`)` nur dann erfüllt, wenn *Term* eine universell quantifizierte Variable ist. Zu diesen drei Prädikaten gibt es jeweils eine negierte Form, die durch Voranstellen von `non` erhalten werden kann; diese ist genau dann erfüllt, wenn die nicht negierte Form fehlschlägt.

Ein- und Ausgabe von Termen Zwischen den Prädikaten, die Terme einlesen, und denen, die Terme ausgeben, besteht ein Unterschied in der Behandlung von Variablen. Bei der Eingabe eines Termes durch das Prädikat `read(`*Term*`)` können nach wie vor nur existentiell quantifizierte Variablen auftreten. Diese Variablen verhalten sich bei der Unifikation so, als wären sie zum Zeitpunkt des Einlesens explizit existentiell quantifiziert worden.

Bei der Ausgabe von Termen gibt es jetzt die folgende Besonderheit: In einem auszugebenden Term können auch allquantifizierte Variablen vorkommen. Diese werden so ausgegeben, daß sie sich als allquantifiziert erkennen lassen: Sie fangen mit einem `A` an, darauf folgt eine Identifikationsnummer. Bei der Wiedereingabe eines ausgegebenen Termes werden diese Variablen jedoch als existentiell quantifiziert behandelt.

Interne Datenbank Die Prädikate, die den Umgang mit der internen Datenbank realisieren, sind im wesentlich identisch zu den entsprechenden Prädikaten von KA–PROLOG. Ein Unterschied in der Semantik ergibt sich durch die Behandlung von existentiell und universell quantifizierten Variablen. Terme werden nach wie vor mit `recorda(`*Schlüssel*`,`*Term*`)` am Anfang, beziehungsweise mit `recordz(`*Schlüssel*`,`*Term*`)` am Ende der Datenbank unter dem Schlüssel *Schlüssel* eingetragen.

Mit `recorded(`*Schlüssel*`,`*MusterTerm*`,`*Liste*`)` werden Terme aus der Datenbank extrahiert, dabei sind die Variablen in den Termen der Liste *Liste* nicht nur umbenannt, sondern noch weiter verändert. Sie behalten die Art ihrer Quantifizierung bei, es ändern sich jedoch ihre Schachtelungsindices, siehe dazu Kapitel 2. Die Schachtelungsindices von existentiell quantifizierten Variablen werden auf null gesetzt. Damit verhalten sich diese Variablen bei der Unifikation so, als wären sie vor jedem Auftreten eines Allquantors bei der Abarbeitung erzeugt worden. Allquantifizierte Variablen erhalten den gerade aktuellen Schachtelungsindex, verhalten sich also bei der Unifikation so, als wären sie bei der Abarbeitung des letzten Allquantors erzeugt worden.

Die Prädikate, die das Einlesen und Abspeichern von ganzen Datenbanken steuern, sind den gleichen Einschränkungen wie die Ein- und Ausgabe von Termen unterworfen.

B.2.2 Weitere Module

Für das KAIN–PROLOG System gibt es neben dem Modul `public` noch andere vordefinierte Module. Dies sind die Module `list`, `solutions`, `sorting` und `utilities`. Bis auf das Modul `solutions` sind diese ungeändert geblieben.

Das Modul `solutions` Das Prädikat `findall(`*Variable*`, `*Anfrage*`, `*Lösungen*`)` wird von diesem Modul angeboten. Es sammelt alle Instantiierungen der existentiell quantifizierten Variable

Variable in *Lösungen* auf, die sich bei der Abarbeitung von *Anfrage* ergeben. Da die Implementierung dieses Moduls auf die interne Datenbasis zurückgreift, gilt für die in *Lösungen* enthaltenen Variablen dasselbe wie für Variablen, die in einem Term auftreten, der aus der internen Datenbank extrahiert wurde.

B.3 Bedienung

Dieser Abschnitt beschreibt die Bedienung des KAIN–PROLOG Systems. Neben der Erläuterung der verschiedenen Prädikate, die zur Steuerung des Systems und des Debuggers dienen, wird die Bedienung anhand einer Beispielsitzung erläutert.

B.3.1 Toplevel

Nach dem Starten des Systems durch die Eingabe von `kain` erscheint auf dem Bildschirm der Prompt des *Toplevels*: `:-)`. Damit ist das System bereit, Anfragen entgegenzunehmen oder vordefinierte Prädikate, die das Verhalten des Toplevel steuern, auszuführen.

B.3.1.1 Setzen des aktuellen Moduls

Bevor an ein Modul eine Anfrage gestellt werden kann, muß es zum *aktuellen* Modul erklärt werden. Dies geschieht durch die Eingabe von

`set_current_module(`*Modulname*`).`

oder kürzer von

`scm(`*Modulname*`).`

Der Name *Modulname* muß einerseits mit dem Namen in der Deklaration des Moduls übereinstimmen, andererseits muß sich das Modul in der Datei *Modulname*`.kai` befinden.

B.3.1.2 Sichtbare Prädikate

Bei der Sichtbarkeit von Prädikaten sind zwei verschiedene Aspekte zu berücksichtigen. Zum einen können Prädikate normal benutzt werden, zum anderen können dynamische Klauseln in hypothetischen Implikationen für Prädikate definiert werden.

Für die Benutzung stehen die Prädikate des Moduls `public`, die Toplevel–Prädikate und alle im aktuellen Modul deklarierten Prädikate — egal ob privat, exportiert oder importiert — zur Verfügung. Dagegen können nur für als exportiert deklarierte Prädikate des aktuellen Moduls dynamische Klauseln in einer Anfrage angegeben werden.

Durch das Eingeben von

`private(`*Name*`/`*Stelligkeit*`).`

kann ein neues Prädikat mit dem Namen *Name* und der angegebenen Stelligkeit im aktuellen Modul deklariert werden. Dieses Prädikat darf benutzt werden, und es dürfen hierfür dynamische Klauseln in Anfragen definiert werden.

B.3.1.3 Anfragen an das aktuelle Modul

Die Anfragen, die an das aktuelle Modul gestellt werden, dürfen genau wie Rümpfe von Klauseln aufgebaut sein, siehe dazu die Definition von Rümpfen in Abschnitt B.1.2.1.

In üblicher Art und Weise werden nach Abarbeitung der Anfrage die Bindungen für die *freien*[1] Variablen ausgegeben. Durch die Eingabe von `;` kann eine weitere Lösung angezeigt werden.

B.3.1.4 Toplevel–Prädikate

Nachfolgend sind alle Toplevel–Prädikate in alphabetischer Reihenfolge aufgelistet. Nicht aufgeführt sind Prädikate, die den Debugger steuern. Diese werden im nächsten Abschnitt behandelt.

`cd(`*Verzeichnis*`).` Es wird in das Betriebssystem–Verzeichnis *Verzeichnis* navigiert.

`current_module.` Gibt den Namen des aktuellen Moduls aus; synonym dazu ist das Prädikat `cm`.

`init_db(`*Modulname*`).` Die Datenbank des Moduls *Modulname* wird initialisiert.

`init_dbs.` Die Datenbanken aller geladenen Module werden initialisiert.

`occ(off).` Schaltet den Occur–Check aus.

`occ(on).` Schaltet den Occur–Check an.

`ocm.` Gibt aus, ob der Occur–Check an- oder ausgeschaltet ist.

`private(`*Name*`/`*Stelligkeit*`).` Deklariert das durch *Name* und *Stelligkeit* beschriebene Prädikat im aktuellen Modul.

`pwd.` Gibt das aktuelle Verzeichnis des Betriebssystems aus.

`quiet(on).` Unterdrückt im weiteren die Ausgabe von Warnungen.

`quiet(off).` Warnungen werden ausgegeben.

`quit.` Die Ausführung des KAIN–PROLOG Systems wird beendet. Die gleiche Semantik haben die Prädikate: `bye`, `end`, `end_of_file`, `exit` und `halt`. Zusätzlich kann auch durch gleichzeitiges Drücken der Tasten [CONTROL] und [D] die Ausführung beendet werden.

`read_dbs(`*Dateiname*`).` Liest die Datei *Dateiname* ein, und fügt die darin enthaltenen Terme den Datenbasen der Module zu. Die Datei muß durch `write_dbs/1` oder `write_db/0` erzeugt worden sein.

`set_current_module(`*Module*`).` Das Modul *Modulname* wird das aktuelle Modul, geladen wird es aus der Datei *Modulname*`.kai`. Synonym dazu ist: `scm(`*Modulname*`)`.

`verbose(on).` Schaltet die Ausgabe von ausführlichen Meldungen zur Übersetzung ein.

`verbose(off).` Schaltet die Ausgabe von ausführlichen Meldungen zur Übersetzung aus.

`write_dbs(`*Dateiname*`).` Die Inhalte aller Datenbanken werden in die Datei *Dateiname* geschrieben.

[1] Man beachte, daß hier *frei* mehr als bei KA–PROLOG bedeutet: Explizit quantifizierte Variablen werden *nicht* betrachtet.

B.3.2 Debugger

Der Debugger des KAIN–PROLOG Systems dient dazu, die Abarbeitung von Anfragen zu verfolgen. Da sich bei der Abarbeitung nicht mehr nur die Anfrage, sondern auch die zur Verfügung stehenden Klauseln ändern können, gibt der Debugger mehr Informationen aus.

Die Ausgabe des Debuggers erfolgt nach einem „Vier–Tor“ Modell, siehe dazu auch Abschnitt 4.5. Prozeduraufrufe erden dabei dabei in der gleichen Art und Weise wie bei KA–PROLOG ausgegeben. Bei der Abarbeitung einer hypothetischen Implikation zeigt der Debugger die folgenden Tore an:

ADD Eine dynamische Klausel der hypothetischen Implikation wird dem Programm hinzugefügt.

REMOVE Nach erfolgreicher Abarbeitung wird die dynamische Klausel entfernt.

READD Durch Rücksetzen wird die dynamische Klausel erneut hinzugefügt.

KILL Nach endgültigem Fehlschlagen der Konklusion der hypothetischen Implikation wird die dynamische Klausel zerstört.

Bei jedem dieser vier Tore wird ausgegeben, welches Kopfprädikat die dynamische Klausel hat, und zu welchem Modul die Klausel gehört.

Die Abarbeitung von Quantoren wird nur implizit bei der Ausgabe der Argumentterme eines Prozeduraufrufes sichtbar. Wie in Abschnitt B.2.1 beschrieben, beginnen allquantifizierte Variablen bei der Ausgabe mit einem `A`. An alle Variablen wird zusätzlich $_n$ angehängt, wobei n den Schachtelungsindex der Variablen angibt, siehe dazu auch Kapitel 2.

Die Behandlung von Spy–Punkten ist gleich geblieben, siehe dazu die Benutzeranleitung von KA–PROLOG [BBB+91].

B.3.2.1 Toplevel–Prädikate für den Debugger

Hier sind alle Toplevel–Prädikate aufgelistet, mit denen man den Debugger steuern kann.

`all_spy_points.` Gibt alle Prädikate aus, auf die ein Spy–Punkt gesetzt ist.

`debug(machine).` Schaltet die Ablaufverfolgung der Maschineninstruktionen ein, beziehungsweise aus.

`debug(off).` Schaltet den Debugger ein.

`debug(on).` Schaltet den Debugger aus.

`nospy(`*Name*`/`*Stelligkeit*`).` Löscht einen eventuellen Spy–Punkt, der auf das spezifizierte Prädikat gesetzt ist.

`spy(`*Name*`/`*Stelligkeit*`).` Setzt auf das spezifizierte Prädikat einen Spy–Punkt. Wird bei der Abarbeitung einer Anfrage dieses Prädikat angesprungen, so wird der Debugger eingeschaltet.

`spy_points.` Gibt alle sichtbaren Prädikate aus, auf die ein Spy–Punkt gesetzt ist.

B.3.2.2 Debugger–Kommandos

Ist der Debugger aktiv, so wartet er nach dem Anzeigen eines ersten Prozeduraufrufs auf eine Tastatureingabe vom Benutzer. Man kann dann die folgenden Kommandos vom Debugger ausführen lassen, wobei die Eingabe aus den jeweils unterstrichenen Buchstaben besteht, und mit einem Drücken der RETURN–Taste abzuschließen ist.

`creep` Setzt die Abarbeitung fort. Das Drücken der RETURN–Taste hat den gleichen Effekt.

`leap` Schaltet den Debugger für den Rest der Abarbeitung aus; Spy–Punkte werden jedoch angezeigt.

`quasi-skip` Schaltet den Debugger für die Abarbeitung der gerade betrachteten Teilanfrage aus. Bei einem Spy–Punkte wird er jedoch wieder eingeschaltet.

`skip` Schaltet den Debugger für die Abarbeitung der gerade betrachteten Teilanfrage aus.

`machine-trace` Schaltet die Ablaufverfolgung der KAIN–Maschine ein, beziehungsweise aus.

`fail` Löst Rücksetzen aus.

`abort` Bricht die Abarbeitung der Anfrage ab.

`nodebug` Der Debugger wird für den Rest der Anfrage ausgeschaltet.

`show redos` Alle Redo–Tore werden angezeigt.

`hide redos` Es werden nur Redo–Tore angezeigt, die wirklich bearbeitet werden müssen.

`?` Alle sichtbaren Spy–Punkte werden aufgelistet.

`??` Alle Spy–Punkte werden aufgelistet.

B.3.3 Beispielsitzung

In diesem Abschnitt werden wir eine kurze Sitzung an dem KAIN System verfolgen. Um das System zu starten, tippt man

```
kain
```

ein. Nach kurzer Wartezeit erscheint der Systemprompt:

```
:-)
```

Jetzt können, wie in Abschnitt B.3.1 erläutert, an das System Anfragen gestellt werden. Bei der Verfolgung von Ein- und Ausgaben des Systems bedienen wir uns der folgenden Notation: Eingaben, die der Benutzer eintippen muß, sind unterstrichen, Ausgaben des Systems nicht. Das Drücken der Taste RETURN ist auch gekennzeichnet.

Zuerst wollen wir ein Beispiel verfolgen, wie neue Prädikate in einer Sitzung deklariert und benutzt werden können. Dazu tippt man das folgende ein:

```
:-) private(p/1).  RETURN
:-) p(a), p(b) -:  p(X).  RETURN
X = a;  RETURN
```

```
module kaefer.

export  steril/1,   erhitzt/1,
        in/2,       tot/1.

body.

steril(X) :-
    all Y : (in(Y, X) -:  tot(Y)).

tot(X) :-
    in(X, Y), erhitzt(Y).

end_module.
```

Abbildung B.1: Das Modul `kaefer`

```
X = b;  RETURN
no
:-)
```

Nach der Deklaration des Prädikates `p/1` wurde eine hypothetische Implikation als Anfrage gestellt. Die hypothetische Implikation wurde abgearbeitet, danach der Prozeduraufruf `p(X)`. Dieser unifiziert mit der dynamischen Klausel `p(a)`, man erhält die Antwortsubstitution `X = a`. Durch das Eingeben von `;` RETURN wird noch die Lösung `X = b` generiert. Eine weitere Lösung gibt es nicht.

Jetzt wird noch einmal das Beispiel 1.2, Seite 2 aufgegriffen. Man braucht dazu das in Abbildung B.1 dargestellte Modul in der Datei `kaefer.kai`.

Zunächst wird das Modul `kaefer` sichtbar gemacht, und der Debugger eingeschaltet. Das geschieht durch Eintippen von:

```
:-) scm(kaefer).  RETURN
:-) debug(on).  RETURN
```

Nun wird die Anfrage gestellt, ob erhitzte Behälter steril sind.

```
:-) all Z : (erhitzt(Z) -:  steril(Z)).  RETURN
(    1)    1  Call:  do_query(G1_0) ?
```

Durch das Ausgeben von `do_query` zeigt der Debugger an, daß er aktiv ist. Um fortzufahren, drückt man einfach RETURN . Wenn an den gekennzeichneten Stellen RETURN gedrückt wird, erscheint nach und nach die folgende Ausgabe:

```
(    2)    2  Add:    kaefer$erhitzt/1
(    3)    3  Call:   steril(A0_1) ?  RETURN
(    3)    3  Redo:   steril(A0_1)
(    4)    4  Add:    kaefer$in/2
(    5)    5  Call:   tot(A1_2) ?  RETURN
(    5)    5  Redo:   tot(A1_2)
```

```
(    6)   6  Call:    in(A1_2, L128_2) ?  RETURN
(    6)   6  Exit:    in(A1_2, A0_1)
(    7)   6  Call:    erhitzt(A0_1) ?  RETURN
(    7)   6  Exit:    erhitzt(A0_1)
(    5)   5  Exit:    tot(A1_2)
(    4)   4  Remove:  kaefer$in/2
(    3)   3  Exit:  steril(A0_1)
(    2)   2  Remove:  kaefer$erhitzt/1
(    1)   1  Exit:    do_query(G1_0)
yes.
```

Die Ausgaben sind im wesentlichen selbsterläuternd. Man beachte, daß für Prädikate, die dynamische Klauseln haben können, bei der Abarbeitung einer statischen Klausel sowohl ein `Call` als auch ein `Redo` angezeigt wird: Der Debugger informiert darüber, daß es keine dynamische Klausel gibt, sondern daß die Abarbeitung bei den statischen Klauseln fortgesetzt wird.

Um die Sitzung zu beenden, tippt man

```
:-) quit.  RETURN
```

ein.

B.4 Installation

In diesem Abschnitt ist die Installation des KAIN–PROLOG Systems auf einem Rechner, der unter dem Betriebssystem UNIX läuft, beschrieben. Sie gestaltet sich denkbar einfach. Zuerst navigiert man in das Betriebssystemverzeichnis `kain/`. In diesem Verzeichnis sind alle Unterverzeichnisse und Dateien enthalten, die für eine Installation des Systems nötig sind. Zusätzlich sind auch noch Verzeichnisse mit Beispielmodulen und der Dokumentation vorhanden.

In dem Verzeichnis `kain/` sind mehrere verschiedene Unterverzeichnisse für verschiedene Rechnertypen enthalten. Je nachdem, auf welchem Rechnertyp die Installation stattfinden soll, navigiert man in das entsprechende Unterverzeichnis. Momentan werden die folgenden Rechnertypen unterstützt: Sun SPARCstation, Sun3 und VAX. Die entsprechenden Verzeichnisse lauten `sparc/`, `sun/` und `vax/`.

Um das KAIN–PROLOG System zu installieren, gibt man

`inst-kain` *Programmverzeichnis Startverzeichnis*

ein. Dabei bezeichnet *Programmverzeichnis* das Betriebssystemverzeichnis, in dem das ausführbare Programm `kain` abgelegt werden soll. Bei dem Start des Programms `kain` werden noch verschiedene Dateien geladen, die das System benötigt: *Startverzeichnis* gibt an, wo diese Dateien bei der Installation abgelegt werden sollen. Nach Ausführung dieses Befehls ist das System komplett installiert.

Nachträglich kann das Verhalten des Systems durch das Setzen von Umgebungsvariablen beeinflußt werden; dies sind:

`kainpath` Wenn das System ein Modul mit dem Namen *Modulname* lädt, so wird die Datei für das Modul anhand dieser Variablen bestimmt. Enthält die Variable eine Liste von Betriebssystemverzeichnissen, so wird in diesen Verzeichnissen nach der Datei *Modulname*`.kai` gesucht.

`kainstart` Enthält diese Variable einen Dateinamen, so wird diese Datei bei dem Start des Systems anstatt der Datei `kain-startup` geladen. Diese Datei enthält Module, die der Initialisierung des Systems dienen.

`kaintop` Wenn anstatt des normalen ein anderes Toplevel geladen werden soll, so muß diese Variable auf die entsprechende Datei gesetzt werden. Diese Datei muß eine Shell–Datei [VA] sein, die Erweiterung des Dateinamens muß „`kit`“ sein.

Weitere Information zu diesen Umgebungsvariablen ist der konkreten Implementierung und den Technischen Noten zum KA–PROLOG System [VA] zu entnehmen.

Anhang C

Beispielprogramme

C.1 Sortieren

```
module sort.

export
    generic_sort/2,          % Quicksort, benutzt...
    generic_sort_unique/2,   % -- " --, Duplikate entfernen, benutzt...
    relation/2,              %         ... diese Relation.
    sort_term_up/2,          % Aufsteigend nach Termkomplexität sortieren.
    sort_term_up_unique/2,   %  -- " --, Duplikate entfernen.
    sort_term_down/2,        % Absteigend nach Termkomplexität sortieren.
    sort_term_down_unique/2. %  -- " --, Duplikate entfernen.

private
    partition/4.

from list import
    append/3.

body.

% generic_sort(-Liste, ?SortierteListe)
%  Sortiert "Liste" nach Quicksort-Algorithmus.
generic_sort([], []).

generic_sort([X | Xs], SXs) :-
    partition(Xs, X, Ls, Gs),
    generic_sort(Ls, SLs),
    generic_sort(Gs, SGs),
    append(SLs, [X | SGs], SXs).
```

```
% partition(-Liste, -Pivotelement, ?KleinerListe, ?GrößerListe)
%  Partitioniert "Liste" nach "Pivotelement" in die Liste der kleineren
%  (oder gleichen) "KleinerListe" und der größeren Elemente "GrößerListe",
%  bestimmt durch relation/2.
partition([], X, [], []).

partition([L | Xs], X, [L | Ls], Gs) :-
    relation(L, X), !,
    partition(Xs, X, Ls, Gs).

partition([G | Xs], X, Ls, [G | Gs]) :-
    partition(Xs, X, Ls, Gs).


% generic_sort_unique(-Liste, ?SortierteListeOhneDuplikate)
%  Sortiert "Liste" in "SortierteListeOhneDuplikate", entfernt Duplikate.
generic_sort_unique(Xs, SXs) :-
    all (
        partition([E | Xs], X, Ls, Gs) :-
            E == X, !,
            partition(Xs, X, Ls, Gs)
    )
    -:  generic_sort(Xs, SXs).


% sort_term_up(-Liste, ?SortierteListe)
%  Sortiert "Liste" aufsteigend nach Termkomplexität in "SortierteListe".
sort_term_up(Xs, SXs) :-
    all (
        relation(X, Y) :-
            X @=< Y
    )
    -:  generic_sort(Xs, SXs).


% sort_term_down(-Liste, ?SortierteListe)
%  Sortiert "Liste" absteigend nach Termkomplexität in "SortierteListe".
sort_term_down(Xs, SXs) :-
    all (
        relation(X, Y) :-
            X @>= Y
    )
    -:  generic_sort(Xs, SXs).
```

```
% sort_term_up_unique(-Liste, ?SortierteListe)
%  Sortiert "Liste" aufsteigend nach Termkomplexität in "SortierteListe",
%  entfernt dabei Duplikate.
sort_term_up_unique(Xs, SXs) :-
    all (
        relation(X, Y) :-
            X @=< Y
    )
    -:  generic_sort_unique(Xs, SXs).


% sort_term_down_unique(-Liste, ?SortierteListe)
%  Sortiert "Liste" absteigend nach Termkomplexität in "SortierteListe",
%  entfernt dabei Duplikate.
sort_term_down_unique(Xs, SXs) :-
    all (
        relation(X, Y) :-
            X @>= Y
    )
    -:  generic_sort_unique(Xs, SXs).


end_module.
```

C.2 Das n–Damen Problem

```
module queens.
% Das N-Damen Problem: Ineffiziente, aber deklarative Version.

export
    queens/2.   % Generiert eine Lösung.

private
    queen/3,    % Die plazierten Damen.
    place/2,    % Plaziert eine Dame.
    attack/0,   % Wann attackieren sich die plazierten Damen?
    position/2, % Wählt ein Platz auf dem Brett aus.
    between/3,  % Wählt eine ganze Zahl aus einem Bereich.
    absdiff/3.  % Berechnet die absolute Differenz zweier Zahlen

body.
```

```
% queens(-BrettGröße, ?Lösung)
%  Plaziert soviel Damen auf dem Brett wie die Kantenlänge
%  "BrettGröße". In "Lösung" stehen die plazierten Damen.
queens(N, QueenList) :-
    % Abkürzung für Brettpositionen
    all (Row, Col) : (
        position(Row, Col) :-
            between(1, N, Row),
            between(1, N, Col)
    )
    -:  place(N, QueenList),


% place(-NummerDerDame, ?TeilLösung)
%  Plaziert die "NummerDerDame"--te Dame auf dem Brett, gibt
%  Lösung in "TeilLösung" zurück.
place(0, []).

place(Queen, [q(Row, Col) | QueenList]) :-
    queen(Queen, Row, Col)            % Dame plazieren
    -:  position(Row, Col),           % Position auf dem Brett
        not attack,                   % Keine Bedrohungen
        NextQueen is Queen - 1,       % Nächste Dame...
        place(NextQueen, QueenList).  % ...plazieren


% attack
%  Ist wahr, falls sich zwei Damen auf dem Brett attackieren.
attack :-
    queen(QueenA, SameRow, _),  % In derselben Zeile sind...
    queen(QueenB, SameRow, _),
    QueenA \== QueenB.          % ...zwei Damen

attack :-
    queen(QueenA, _, SameCol),  % In derselben Spalte sind...
    queen(QueenB, _, SameCol),
    QueenA \== QueenB.          % ...zwei Damen

attack :-
    queen(QueenA, RowA, ColA),
    queen(QueenB, RowB, ColB),
    QueenA \== QueenB,          % Zwei Damen...
    absdiff(RowA, RowB, Delta),
    absdiff(ColA, ColB, Delta). % ...in derselben Diagonale
```

```
% absdiff(+X, +Y, ?Z)
%  "Z" wird zur absoluten Differenz von "X" und "Y" berechnet.
absdiff(X, Y, Z) :-
    X > Y,
    Z is X - Y.

absdiff(X, Y, Z) :-
    X =< Y,
    Z is Y - X.


% between(+Left, +Right, ?BetweenNumber)
%  Wählt "BetweenNumber" aus "Left"..."Right" aus.
between(Left, Right, Left) :-
    Left =< Right.

between(Left, Right, Selected) :-
    Left < Right,
    NewLeft is Left + 1,
    between(NewLeft, Right, Selected).


end_module.
```

C.3 Der Cocke–Younger–Kasami–Algorithmus

```
module cyk.
% Implementierung des Cocke-Younger-Kasami-Algorithmus.

export
    nonterminals/1, % Liste der Nichtterminalsymbole
    production/2,   % Produktionen in Chomsky-Normalform:
                    %   production(NT, [LeftNT, RightNT]), bzw.
                    %   production(NT, [TermSym])
    startsym/1,     % Das Startsymbol der Grammatik
    cyk/1.          % Test auf Ableitbarkeit eines Wortes

private
    base/1,         % Start der Berechnung der Basisfälle
    do_base/3,      % Berechnung der Basisfälle
    step/1,         % Start der weiteren Berechnung
    do_step/2,      % Einen Berechnungschritt vorbereiten...
```

```
    derive_step/4,  % ... und ausführen
    prefix/3,       % Präfix einer Liste bestimmen
    split/3,        % Liste in zwei echte Teillisten aufspalten
    word/2,         % Zum Merken des bearbeiteten Wortes
    table/2.        % Zum Merken der Ableitbarkeit von Teilwörtern

from list import
    length/2.       % Zur Bestimmung der Länge einer Liste

body.

% cyk(-Wort)
%  Ist erfolgreich wenn "Wort" in der spezifizierten Grammatik
%  herleitbar ist
cyk(Word) :-
    length(Word, WordLen),
    (   word(Word, WordLen)
        -:  base(Word)
    ).


% base(-Wort)
%  Berechnet den Basisfall für alle Terminalsymbole aus "Wort".
base([]) :-
    step(2).

base([TermSym | Word]) :-
    nonterminals(NonTermSyms),
    do_base(NonTermSyms, TermSym, Word).


% do_base(-Nichtterminalsymbole, -Terminalsymbol, -Wort)
%  Alle "Nichtterminalsymbole", die sich auf "Terminalsymbol" ableiten
%  lassen, werden in die Tabelle übertragen.
do_base([], TermSym, Word) :-
    base(Word).

do_base([NonTermSym | NonTermSymS], TermSym, Word) :-
    production(NonTermSym, [TermSym]), !,
    (   table(NonTermSym, [TermSym])
        -:  do_base(NonTermSymS, TermSym, Word)
    ).

do_base([ _ | NonTermSymS], TermSym, Word) :-
```

```
    do_base(NonTermSymS, TermSym, Word).


% step(-Länge)
%  Startet die Untersuchung der Teilwörter der Länge "Länge".
step(Len) :-
    % Ist das Ausgangswort schon untersucht?
    word(Word, WordLen),
    Len > WordLen, !,
    % Ja! Ist es aus dem Startsymbol herleitbar?
    startsym(StartSym),
    table(StartSym, Word).

step(Len) :-
    word(Word, _),
    do_step(Len, Word).


% do_step(-PräfixLänge, -RestWort)
%  Startet für alle Teilwörter von "RestWort" der Länge "PräfixLänge" den
%  eigentlichen Ableitungsschritt.
do_step(Len, Word) :-
    prefix(Len, Word, Prefix), !,
    nonterminals(NonTermSymS),
    % Berechne alle Tabelleneinträge für das Teilwort "Prefix"
    derive_step(NonTermSymS, Prefix, Word, Len).

do_step(Len, Word) :-
    % Alle Teilwörter sind betrachtet
    NewLen is Len + 1,
    step(NewLen).


% derive_step(-Nichtterminalsymbole, -Präfix, -Wort, -Länge)
%  Berechnet die eigentlichen Tabelleneinträge für alle "Nichtterminal-
%  symbole", die sich auf "Präfix" ableiten lassen. Danach wird das
%  nächste Teilwort der Länge "Länge" von "Wort" untersucht.
derive_step([], _, [TermSym | Word], Len) :-
    % Und das nächste Teilwort untersuchen...
    do_step(Len, Word).

derive_step([NonTermSym | NonTermSymS], Prefix, Word, Len) :-
    % Dieser Tabelleneintrag ist schon berechnet!
    table(NonTermSym, Prefix), !,
```

```
    derive_step(NonTermSymS, Prefix, Word, Len).

derive_step([NonTermSym | NonTermSymS], Prefix, Word, Len) :-
    % Zuerst eine Produktion auswählen.
    production(NonTermSym, [LeftNonTermSym, RightNonTermSym]),
    % Linkes und rechtes Teilwort des Präfix...
    split(LeftPrefix, RightPrefix, Prefix),
    % ...müssen aus den Nichtterminalsymbolen ableitbar sein.
    table(LeftNonTermSym, LeftPrefix),
    table(RightNonTermSym, RightPrefix), !,
    (   table(NonTermSym, Prefix)
        -:  derive_step(NonTermSymS, Prefix, Word, Len)
    ).

derive_step([ _ | NonTermSymS], Prefix, Word, Len) :-
    % Betrachte den Rest der Nichtterminalsymbole
    derive_step(NonTermSymS, Prefix, Word, Len).


% prefix(-Länge, -Wort, ?Präfix)
%  Berechnet den Präfix "Präfix" des Wortes "Wort" der Länge "Länge".
prefix(0, _, []).

prefix(Len, [Sym | Word], [Sym | Prefix]) :-
    Len > 0,
    NewLen is Len - 1,
    prefix(NewLen, Word, Prefix).


% split(?LinkesTeilwort, ?RechtesTeilwort, ?Wort)
%  Spaltet das Wort "Wort" in die echten Teilwörter "LinkesTeilwort"
%  und "RechtesTeilwort" auf.
split([LeftSym], [RightSym | Right], [LeftSym, RightSym | Right]).

split([Sym | Left], Right, [Sym | Word]) :-
    split(Left, Right, Word).


end_module.
```

Literaturverzeichnis

[AK90] H. Aït-Kaci. The WAM: A (Real) Tutorial. Report 5, Digital Equipment Corporation, Paris Research Laboratory, January 1990.

[ASU88] A. V. Aho, R. Sethi und J. D. Ullman. *Compilerbau.* Addison–Wesley, 1988.

[BBB+91] M. Baur, A. Bockmayr, C. Brzoska, E. Heinz, P. Lukowicz, H. H. Rath, C. Scharnhorst und B. Schiele. Benutzeranleitung für die KA–Prolog–Sprachfamilie. Interner Bericht 20/91, Fakultät für Informatik, Universität Karlsruhe (TH), 1991.

[BBL+91] A. Bockmayr, C. Brzoska, N. Lindenberg, B. Neidecker, H. H. Rath, C. Scharnhorst, B. Schiele und I. Varšek. KA–Prolog: Sprachdefinition. Interner Bericht 19/91, Fakultät für Informatik, Universität Karlsruhe (TH), 1991.

[BM90] A. J. Bonner and L. T. McCarty. Adding Negation–as–failure to Intuitionistic Logic Programming. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the North American Conference*, pages 681–703, Austin, Texas, USA, 1990. MIT–Press.

[BMV89] A. J. Bonner, L. T. McCarty, and K. Vadaparty. Expressing Database Queries with Intuitionistic Logic. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, pages 831–850, Cleveland, Ohio, USA, 1989. MIT–Press.

[Bue86] K. A. Buettner. Fast Decompilation of Compiled Prolog Clauses. In E. Shapiro, editor, *Logic Programming: Proceedings of the Third International Conference on Logic Programming*, London, UK, 1986. Springer–Verlag.

[CB83] J. Corbin and M. Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In R. E. A. Mason, editor, *Information Processing '83*, pages 909–914. Elsevier Science Publishers, 1983.

[Fit69] M. C. Fitting. *Intuitionistic Logic, Model Theory and Forcing.* North–Holland, 1969.

[Gab85] D. M. Gabbay. N–Prolog: an Extension of Prolog with Hypothetical Implication. II. Logical Foundations, and Negation as Failure. *Journal of Logic Programming*, 4:251–283, 1985.

[GR84] D. M. Gabbay and U. Reyle. N–Prolog: an Extension of Prolog with Hypothetical Implication. I. *Journal of Logic Programming*, 4:319–355, 1984.

[HU90] J. E. Hopcroft und J. E. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison–Wesley, zweite Auflage, 1990.

[JN91] B. Jayaraman and G. Nadathur. Implementation Techniques for Scoping Constructs in Logic Programming. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eight International Conference*, pages 871–886, Paris, France, 1991. MIT–Press.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice–Hall International, second edition, 1988.

[Kri65] S. Kripke. Semantical Analysis of Intuitionistic Logic. I. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North–Holland, 1965.

[Kur86] P. Kursawe. How to Invent a Prolog Machine. In E. Shapiro, editor, *Logic Programming: Proceedings of the Third International Conference on Logic Programming*, London, UK, 1986. Springer–Verlag.

[Lin87] N. Lindenberg. KA–Prolog: Der Compiler. Arbeitsbericht, Institut für Informatik I, Universität Karlsruhe (TH), Mai 1987.

[McC87] J. McCarthy. Generality in Artificial Intelligence. *Communications of the ACM*, 30(12):1029–1035, Dec 1987.

[McC88a] L. T. McCarty. Clausal Intuitionistic Logic. I. Fixed–Point Semantics. *Journal of Logic Programming*, 5:1–31, 1988.

[McC88b] L. T. McCarty. Clausal Intuitionistic Logic. II. Tableau Proof Procedures. *Journal of Logic Programming*, 6:93–132, 1988.

[Mil89a] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

[Mil89b] D. Miller. Lexical Scoping as Universal Quantification. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 268–283, Lisbon, Portugal, 1989. MIT–Press.

[Mil90] D. Miller. Abstractions in Logic Programming. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[MJT91] L. Matyska, A. Jergová, and D. Toman. Register Allocation in WAM. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eight International Conference*, pages 142–156, Paris, France, 1991. MIT–Press.

[Nei86] B. Neidecker. KA–Prolog: Maschinenmodell 0.4. Interner Bericht 19/86, Institut für Informatik I, Universität Karlsruhe (TH), 1986.

[Nei87] B. Neidecker. KA–Prolog: Entwurf eines compilierenden Entwicklungssystems. Interner Bericht 29/87, Institut für Informatik I, Universität Karlsruhe (TH), 1987.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In K. Bowen and R. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seattle, Washington, USA, 1988. MIT–Press.

[Rob65] J. A. Robinson. A Machine oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):107–124, Jan 1965.

[RP89] P. Ružička and I. Prìvara. An Almost Linear Robinson Unification Algorithm. *Acta Informatica*, 27:61–71, 1989.

[Sch91a] C. Scharnhorst. Entwurf eines übersetzenden Narrowing–Systems basierend auf der abstrakten Warren–Maschine (WAM). Diplomarbeit, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe (TH), Mai 1991.

[Sch91b] C. Schulte. Ansätze zum Intuitionistischen Logischen Programmieren. Studienarbeit, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe (TH), Februar 1991.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT–Press, 1986.

[VA] Verschiedene Autoren. KA–PROLOG: Technical Notes. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe (TH).

[vD86] D. van Dalen. Intuitionistic Logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 3, pages 225–339. D. Reidel Publishing Company, 1986.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artifical Intelligence Center, SRI International, October 1983.

[Wei89] T. Weinstein. LOPSTER, Entwurf und Realisierung einer Laufzeitumgebung für mehrsortiges Prolog. Diplomarbeit, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe (TH), März 1989.

[WG85] W. M. Waite and G. Goos. *Compiler Construction.* Springer–Verlag, 1985.

[Wir83] N. Wirth. *Algorithmen und Datenstrukturen.* Teubner–Verlag, dritte überarbeitete Auflage, 1983.