Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-91-08

Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN

Thomas Krause

März 1991

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN

Thomas Krause

DFKI-D-91-08

Diplomarbeit

Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN

Thomas Krause

März 1991

Betreuung:

Prof. Dr. Michael M. Richter
Dr. Harold Boley

Universität Kaiserslautern

Fachbreich Informatik

Danksagung

Für die freundliche Atmosphäre im ARC-TEC Projekt unter der Gesamtleitung von Prof. Dr. M. M. Richter bin ich allen wissenschaftlichen Mitarbeitern und Studenten zu Dank verpflichtet. Besonders möchte ich Dr. Harold Boley danken, der diese Arbeit betreute und mit RELFUN eine der Grundlagen für die Arbeit geschaffen hat. Eine besondere Anerkennung verdient ebenfalls meine Verlobte Iris Segebade, die es in der Schlußphase meiner Diplomarbeit verstanden hat, die alltäglichen Sorgen von mir fernzuhalten und einen großen Anteil am Gelingen dieser Arbeit trägt.

Übersicht

Einleitung

PROLOG und LISP sind ohne Frage die wohl etabliertesten Programmiersprachen in der Künstlichen Intelligenz. Das allgemeine Interesse an ihnen hat im Hinblick auf leistungsstarke Sprachimplementationen große Fortschritte gebracht. Für PROLOG ist in dieser Hinsicht mit David H. D. Warren und der Definition eines *Abstract Instruction Set for PROLOG* (-> WAM *Warren Abstract Machine*) [War83] ein Durchbruch gelungen. Praktisch alle heute verfügbaren PROLOG-Systeme compilieren die PROLOG-Klauseln hinunter nach *WAM-Instruktionen,* welche von einem *WAM-Emulator* ausgeführt werden. Die Emulatoren laufen auf *general purpose Prozessoren* und sind teilweise in C geschrieben (z. B. PROLOG XT [Bue89], NU-PROLOG [Tho88], KAP-PROLOG [Nei87] oder QUINTUS-PROLOG [Qui85]), teilweise existieren aber auch sehr maschinennahe Realisierungen wie beispielsweise bei dem MIPS-Compiler von Andrew Taylor [Tay90]. Es existieren auch bereits Spezialentwicklungen auf der Hardware- und Mikroprogramm-Ebene wie etwa die am ECRC[1] entwickelte KCM (*Knowledge Crunching Machine*) [Ben90].

In dieser Arbeit beschäftige ich mich mit RELFUN [Bol86]. RELFUN ist eine den relationalen Stil von PROLOG und den funktionalen Stil von LISP integrierende Programmiersprache. Für RELFUN wird das experimentelle LISP-basierte RFM[2]-System [Bol91] (bestehend aus Interpreter, Compiler und Emulator[3]) am DFKI entwickelt. Ein LISP-basierter Emulator hat gegenüber maschinennäheren Implementationen den Vorteil, daß er zum einen überschaubar und gut portierbar und zum anderen leicht modifizierbar ist. Für die Exploration neuer Ideen ist dies ein wichtiger Faktor, während die Effizienz (des Emulators) am Anfang der Entwicklung keine sehr große Rolle spielt, da sie durch eine Reimplementierung in einer "tieferen" Sprache nachträglich erreicht werden kann.

Das Hauptanliegen meiner Arbeit ist es, "intelligente" Compilationsmethoden für eine "sehr hohe" Programmiersprache zu untersuchen und einen Beitrag zur Realisierung dieser Methoden zu leisten. Die Problemstellung erfordert zum Teil eine Auseinandersetzung mit einer tiefgehenden Architekturbeschreibung (bis hin zur Registerebene der WAM) und ist deswegen größtenteils sehr technisch. Obwohl man den klassischen Compilerbau nicht zu der KI zählt, was wahrscheinlich ursächlich daran liegt, daß die imperativen Sprachen keine "KI-Sprachen" sind und der Compilerbau viel früher in der Informatik Gegenstand der Forschung war, darf sich die KI dieser Thematik nicht verschließen, wenn ihre Sprachen auch für den Einsatz in

---

[1] ECRC- European Computer-Industrie Research Center GmbH.

[2] relational/functional machine: Das RELFUN-Analogon der WAM für PROLOG.

[3] Es handelt sich um einen fast unveränderten (relationalen) WAM-Emulator. Die funktionalen Aspekte von RELFUN werden durch den Compiler realisiert, der die Instruktionen so generiert, daß die X-Register der WAM (normalerweise nur für die Aufrufargumente und Zwischenergebnisse genutzt) als Rückgabe-Register für die Funktionswerte mitbenutzt werden.

großen Systeme interessant werden sollen. Es reicht beispielsweise nicht aus, vertrauensvoll auf den Fortschritt der Hardwaretechnologie zu warten, denn die eher kurze Geschichte der elektronischen Rechner hat gezeigt, daß mit steigender Geschwindigkeit der Hardware die Komplexität der in Angriff genommenen Problemklassen den Geschwindigkeitsgewinn in aller Regel kompensieren.

Es ist deshalb naheliegend, für die Compilation von KI-Sprachen auch bewährte Techniken aus der KI einzusetzen. Die in dieser Arbeit untersuchten Ansätze lassen sich durch die zwei Schlüsselwörter *Normalform* und *wissensbasiert* zusammenfassen. Die Idee, eine RELFUN-Normalform zu bilden und als Grundlage für weitere Compilationsschritte zu verwenden, wird in Abschnitt 2 zu der Reduzierung von RELFUN-Klauseln auf den RELFUN-Kern ausgebaut. Es handelt sich hierbei um "horizontale" Transformationsschritte (oder auch Compilationsschritte), die unter anderem statische Vorauswertungen (*preevaluation*) auf Klausel-Eben durchführen. Es handelt sich hierbei nicht nur um syntaktische Transformationen, sondern auch um semantische Auswertungen. Abschnitt 3 ist durch wissensbasierte Aspekte einer intelligenten Compilation motiviert. Durch statische Analyse einer RELFUN-Datenbasis werden von einem *mode-Interpreter* Informationen über das Input-/Outputverhalten der Prozeduren berechnet und in einer *mode-Tabelle* zur Verfügung gestellt. Es werden in diesem Abschnitt zusätzlich einige weiterführende Transformationen vorgestellt, die auf den mode-Informationen aufbauen und über der Klausel-Ebene ganze Prozeduren der Datenbasis berücksichtigen.

Abschnitt 1 beschreibt den Rahmen der Arbeit im ARC-TEC Projekt am DFKI. Hier findet der Leser noch einmal detaillierter die Hauptideen und Motivationen für die Abschnitte 2 und 3 zusammengetragen.

Abschnitt 2 und Abschnitt 3 führen zu den Programmen *normalizer* und *mode-interpreter*. Sie wurden beide auf einem Macintosh II unter Allegro Common LISP [All87] entwickelt und sind in das RFM-System eingebunden, welches zur Zeit auf einer SUN 4/60 unter UNIX in einer KCL-Umgebung (Kyoto Common LISP [Kyo??]) läuft. Das Listing beider Programme ist im Anhang abgedruckt, während die Implementierungshinweise, welche über die Beschreibungen in den Listings hinausgehen, am Ende des jeweiligen Abschnitts zu finden sind.

1 Rahmen und Grundlage der Arbeit

Die Diplomarbeit ist am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI GmbH) im ARC-TEC Projekt unter der Gesamtleitung von Prof. Dr. M. M. Richter entstanden. Eine Einordnung der Arbeit in das Compilationsvorhaben von ARC-TEC (Akquisition, Repräsentation und Compilation von TEChnischem Wissen) wird im Abschnitt 1.1 gegeben. Die zugrundegelegte Programmiersprache RELFUN[4] wird im Abschnitt 1.2 in Form einer kurzen Einführung vorgestellt.

1.1 Das Forschungsvorhaben von ARC-TEC

ARC-TEC hat sich als Ziel die Entwicklung einer Expertensystem-Shell mit domänenspezifischen Aspekten aus dem Maschinenbau gesetzt, welche zur Erstellung eines technischen Expertensystem dienen soll. Die C-Gruppe hat hierbei die Aufgabe, Methoden für die deklarative Wissensrepräsentation und Inferenzen für diese Shell effizient zu implementieren. Es sind dies die Formalismen für Konzeptbeschreibungen (in TAXON), Constraints (in CONTAX (eng gekoppelt mit TAXON)) sowie bewertete Hornregeln und Vorwärtsregeln (in AFFIRM). AFFIRM enthält also ein Regelsystem, für welches sowohl Vorwärts- als auch Rückwärtsableitung gestattet ist. Die Rückwärtsableitung ist durch die Abarbeitung von Klauseln in RELFUN gegeben und die Vorwärtsableitung wird u. a. mittels einer ("horizontalen") Compilation der Vorwärtsregeln nach Hornregeln in RELFUN realisiert.

Der Begriff Compilation wird auf zweierlei Arten betrachtet. Zum einen, wie oben kennengelernt, als *horizontale Compilation*. Hierunter verstehen wir Programmtransformationen auf der Source-Ebene. Ein weiteres Beispiel für die horizontale Compilation ist etwa *Partielle Evaluation* bzw. die Normalisierung von RELFUN, die uns in Abschnitt 2 begegnen wird. Die *vertikale Compilation* ist die zweite Verwendung des Begriffs. In diesem Fall betrachten wir einen Transformationsprozeß von einer abstrakten (high level) Sprache (oder Darstellungsform) in eine maschinenabhängige (low level) Sprache. Beispielsweise erhalten wir die RFM-Instruktionen aus einem RELFUN-Programm aufgrund einer vertikalen Compilation. Die Abbildung der T-Box in TAXON auf RELFUN-Klauseln wäre ein weiteres Beispiel.

[4] relational/functional language

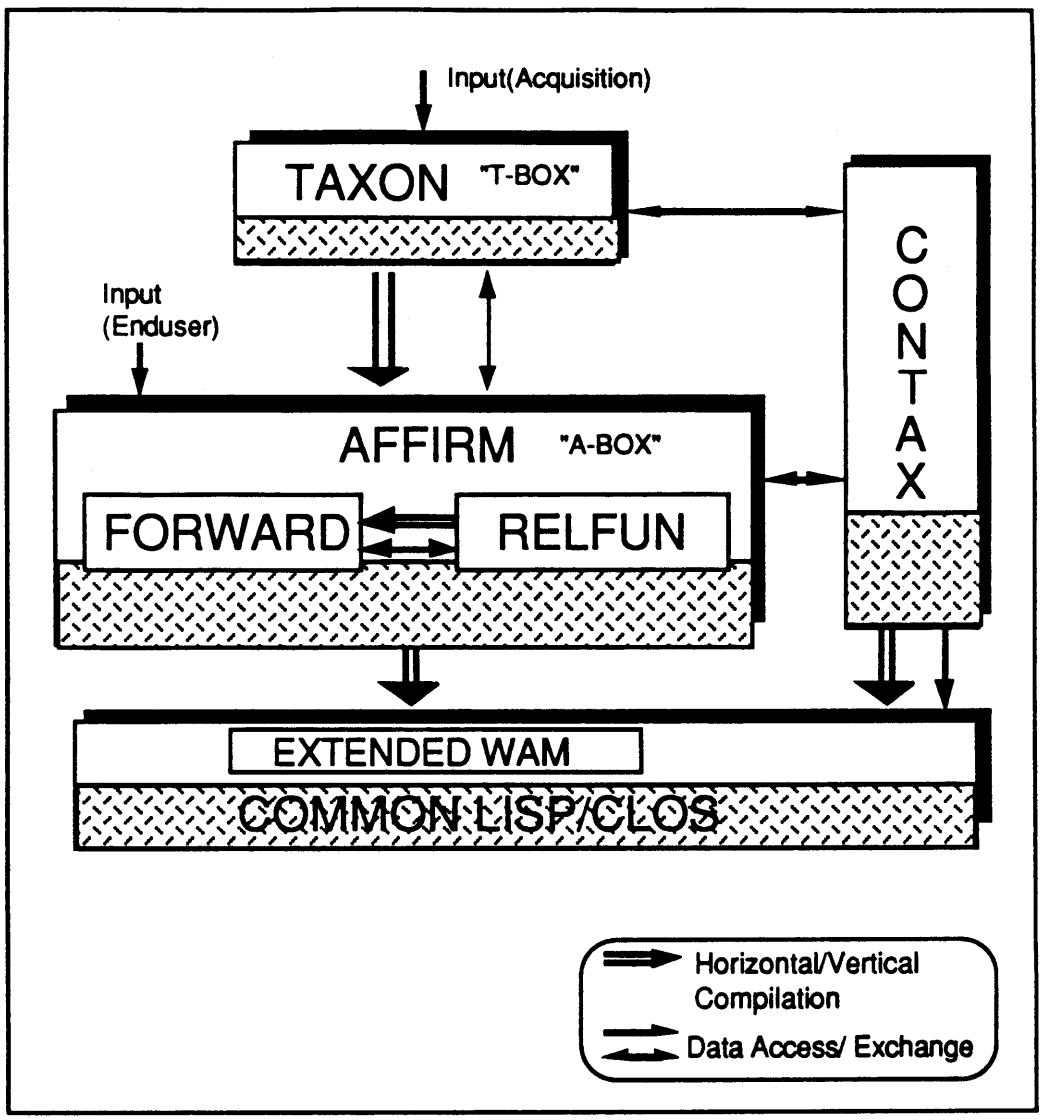


Bild 1.1 Architektur der vom C-Teil bereitzustellenden Formalismen

Diese Arbeit entstand innerhalb des RFM-Compilationslabor *Complab*, welches im C-Teil des ARC-TEC Projekts integriert ist. Das Complab hat sich zur Aufgabe gesetzt, Compilationsmethoden für relationale, funktionale und anderen Sprachen zu untersuchen, zu verbessern sowie neue Techniken zu entwickeln.

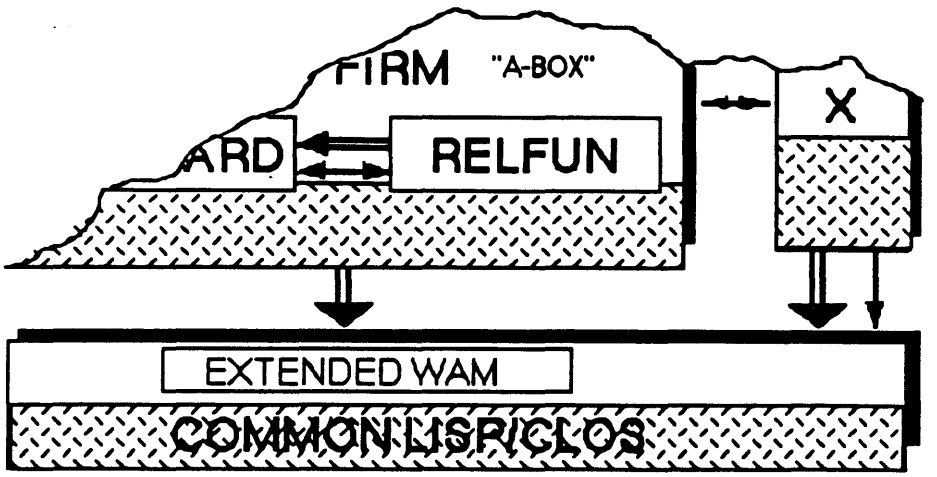


Bild 1.2 Einordnung von Complab in den C-Teil

Es wurde bereits ein Compiler entwickelt und implementiert, der RELFUN vertikal in RFM-Code compiliert. Hierbei wurde ein Compilationsansatz über die deklarative Zwischensprache *Classified Clauses* [Kra90] verwendet. RELFUN-Klauseln werden in dieser Zwischensprache mit zusätzlichen Informationen auf verschiedenen Ebenen angereichert. Es werden 5 Beschreibungsebenen[5] unterschieden, so daß wir Informationen über die Prozedur-, Klausel-, Chunk-, Literal- und Argumentebene in den Classified Clauses finden können. Die Idee der Zwischensprache macht die Philosophie des Complabs, möglichst viel Compilationsarbeit auf hoher Ebene zu tätigen und explizit zu machen, deutlich. Das Wissen über eine Klausel, das ein Codegenerator für optimale Codegenerierung benötigt und in einem konventionellen (Programm-) Compilersystem nur temporär während des Compilierens erzeugt wird, macht die Zwischensprache explizit und somit auch für spätere Verwendungen zugänglich. Der RFM-Codegenerator [Hei89] benutzt als Eingabe die Classified Clauses und erzeugt in nur einem Pass optimierte RFM-Instruktionen. Aufgrund der Philosophie des Complabs ist der Codegenerator von einer relativ einfachen Struktur und deswegen auch einfach auf andere Systeme zu adaptieren, was ein weiteres Plus für den in Complab gewählten Ansatz ist.

[5] Die Classified Clauses haben noch keine endgültige Fassung. Zum Beispiel wird derzeit an einer zusätzlichen db-Beschreibungsebene für komplette Datenbasen (oder Moduln) gearbeitet.

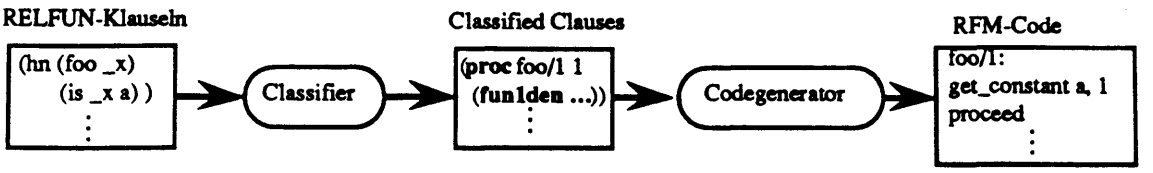


Bild 1.3 Das bisherige Compiler-System im Complab

Der Inhalt dieser Arbeit besteht nun darin, den oben beschriebenen Compilationsprozeß weiter zu verfeinern. Dies geschieht durch horizontale Compilationsschritte (im *normalizer*: Abschnitt 2) auf Source-Ebene, die eine (allgemeine) RELFUN-Klausel zum einen vereinfachen und zum anderen auf eine einheitliche (oder normalisierte) Form transformieren. Wir erhalten so eine Kern-Klausel. Der Name soll ausdrücken, daß die syntaktische Vielfalt von RELFUN-Ausdrucksweisen, auf eine kleine (aber von der Beschreibungsmächtigkeit vollständige) Anzahl von Möglichkeiten, nämlich den RELFUN-Kern, reduziert wird. Desweiteren wird in Abschnitt 3 das Ein- und Ausgabeverhalten von RELFUN-Variablen in einem Programmablauf untersucht. Dies führt uns zu der globalen Datenflußanalyse eines RELFUN-Programms, dessen Ergebnis die *modes* der einzelnen Prozeduren ist.

Für zukünftige Arbeiten könnten die modes auf vielfältige Art und Weise genutzt werden. Beispielsweise sind aufgrund der mode-Information entsprechende Erweiterungen der Classified Clauses um Indexierungsinformationen denkbar, die der Codegenerator für effiziente Indexierungsmethoden nutzen kann. Noch zu entwickelnde weitere horizontale Compilationsstools wie etwa ein *partieller Evaluator*, könnten die modes ebenfalls dazu benutzen, Ergebnisse von Ausdrücken statisch zu berechnen. Im Motivationsteil von Abschnitt 3 findet man weiterführende Beispiele für die Nutzung von mode-Informationen.

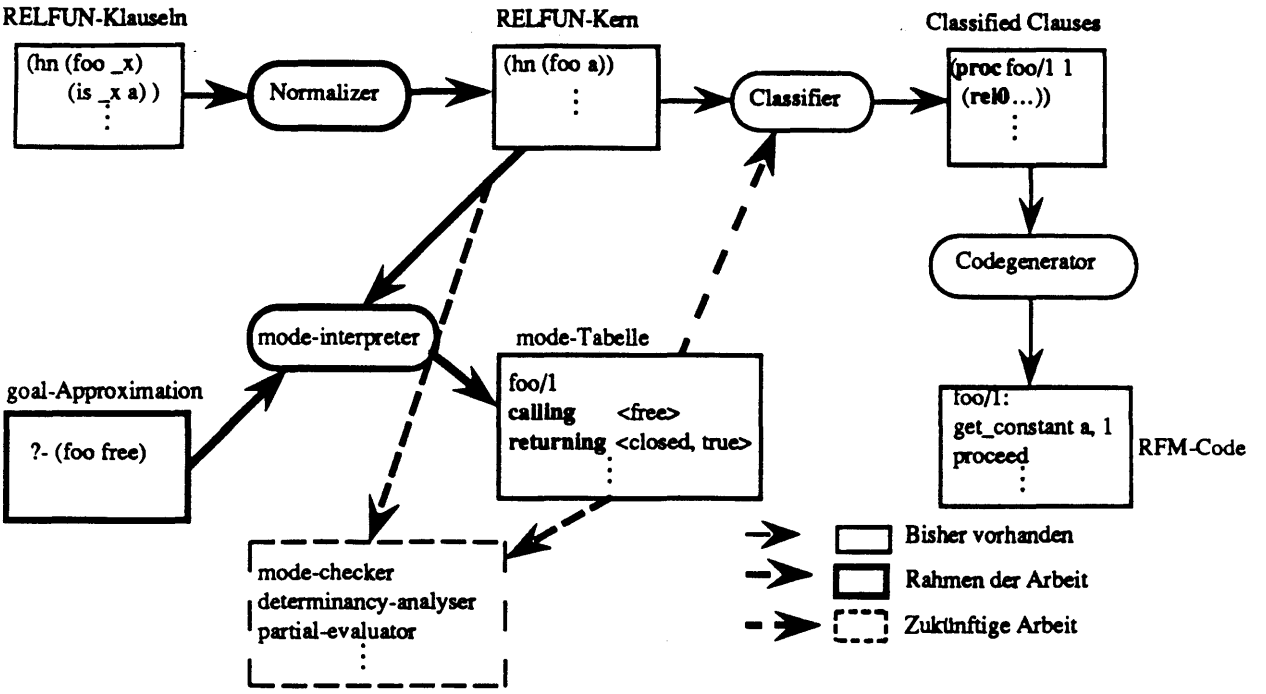


Bild 1.4 Das in Angriff genommene Compiler-System von Complab

1.2 Kurzeinführung in RELFUN

Die vorliegende Arbeit beschäftigt sich unter anderem mit horizontalen *Source-to-Source*-Transformationen innerhalb der Programmiersprache RELFUN. Dieser Abschnitt gibt eine kurze Einführung in RELFUN. Die Einführung beschränkt sich auf *first-order*-RELFUN und ist zum Teil sehr knapp gehalten. Für das Verständnis ist Grundwissen über Logik, sowie die Kenntnis der Programmiersprachen LISP und vor allem PROLOG nötig. Eine tiefergehende Beschreibung findet man in [Bol86] und [Bol90a]. Dort werden zusätzlich die *higher-order*-Möglichkeiten von RELFUN besprochen.

RELFUN ist eine deklarative Programmiersprache, die den funktionalen (applikativen) Stil von LISP und den logischen (relationalen) Stil von PROLOG vereinheitlicht, indem sie eine um Wertrückgabe verallgemeinerte *SLD-Resolution*[6] implementiert. Wenn in RELFUN von Funktionen gesprochen wird, ist ein weiterer Funktionsbegriff gemeint als er z. B. in LISP gebraucht wird. In LISP ist eine Funktion eine eindeutige Abbildung von festen Eingabeargumenten auf einen Ausgabewert. In RELFUN dagegen muß eine Funktion weder eindeutig (deterministisch) noch *ground*[7] sein, kann also auch mehrere Werte über *backtracking* aufzählen sowie *non-ground*-Werte liefern und Anfragevariablen binden. Eine Funktion in

[6]Linear resolution with Selection function for Definite clauses [Kow71].
[7]Es kommen in den Argumenten keine Variablen oder Terme mit Variablen vor.

RELFUN ist wie ein PROLOG-Prädikat zu sehen, welches zusätzlich Werte zurückgibt. RELFUN spricht deswegen von bewerteten Klauseln (*valued clauses*).

Ein RELFUN-Programm ist eine endliche Menge von Prozeduren (oder auch Prädikaten). Eine Prozedur besteht aus einer endlichen, geordneten Menge von Klauseln mit gleichem Prädikatnamen und gleicher Arität. Eine Klausel hat endlich viele Prämissen p_i (i= 1...n) und eine Konklusion *h*. Die Konklusion ist der Klauselkopf und die Prämissen der Klauselrumpf. Konklusion und Prämissen sind Literale bestehend aus einem Funktor und einer festen Anzahl $k \geq 0$ von Argumenten. In dieser Arbeit wird ein allgemeinerer Literalbegriff als üblich verwendet, so daß die Argumente von Literalen selbst wieder Literale (üblicherweise nur Terme) sind. Eine Klausel spiegelt die logische Implikation $h <- p_1,\ldots,p_n$ wider, wobei nicht nur der Wahrheitswert der Implikation interessiert, wie dies beispielsweise in PROLOG der Fall ist, sondern die Implikation einen beliebigen Wert liefern kann. Für $n = 0$ wird die Klausel wie üblich als Fakt bezeichnet. In der PROLOG-Terminologie geht man in RELFUN von Horn-Klauseln aus und hat eine wie in Bild 1.5 (a) angedeutete Syntax. Die entsprechende LISP-Syntax von RELFUN ist in Bild 1.5 (b) gezeigt.

(a) $h :- p_1,\ldots,p_n.$ $(n \geq 0)$ (b) $(hn\ h\ p_1,\ldots,p_n)$ $(n \geq 0)$

Bild 1.5 PROLOG-Syntax (a) und LISP-Syntax (b) einer RELFUN-Klausel

Man unterscheidet in RELFUN zwischen *hn-Klauseln* (hornish clauses) und *ft-Klauseln* (footed clauses). Eine hn-Klausel setzt sich syntaktisch aus dem Klausel-tag hn, dem Klauselkopf *head* und dem Klauselrumpf *body*[8] zusammen. Eine hn-Klausel liefert immer den Wert *true*, falls der Aufruf erfolgreich ist.

(hn *head body*)

(ft *head body foot*)

Bild 1.6 hn-Klausel und ft-Klausel in RELFUN

Eine ft-Klausel setzt sich aus dem Klausel-tag ft, dem Klauselkopf *head*, dem Klauselrumpf *body*, sowie dem Klauselfuß *foot* zusammen. Den Klauselfuß bezeichnen wir hier auch als foot-Literal. Das foot-Literal bestimmt den Wert der ft-Klausel, d. h. der Wert des foot-Literals ist zugleich der Wert der ft-Klausel. Es bestehen folgende Reduktionsmöglichkeiten zwischen hn- und ft-Klauseln [Bol90]. Eine hn-Klausel kann in eine äquivalente[9] ft-Klausel transformiert werden, indem an die hn-Klausel die foot-Konstante *true* angehängt wird, umgekehrt ist es möglich, ft-Klauseln, die als foot-Literal *true* besitzen, in hn-Klauseln zu transformieren.

[8] Der Bezeichner *body* steht für Null oder mehr Literale.
[9] "Äquivalent" bezüglich gelieferten Bindungen und Werten..

(hn *head body*) ⇔ (ft *head body* true)

Bild 1.7 Äquivalenz von hn-Klauseln und ft-Klauseln

Die Prämissen (Klauselrumpf und Klauselfuß) können *evaluativ* oder *denotativ* sein. Eine evaluative Prämisse entspricht einem Prozeduraufruf (*goal*) in PROLOG. Das goal kann scheitern oder erfolgreich sein. Ist das goal erfolgreich, gibt es seinen Wert zurück. Ein goal, das scheitert, liefert ein *unknown*-Signal[10]. Eine denotative Prämisse bezeichnet stets sich selbst (eventuell *instantiiert*[11]) als Wert und kann nicht scheitern. Denotative Prämissen sind Konstanten, Variablen, Strukturen und Listen; sie sind in Bild 1.8 exemplarisch aufgeführt. Man beachte, daß sich Strukturen von evaluativen Literalen syntaktisch dadurch unterscheiden, daß sie mit dem inst-Operator *backquote* ("`") beginnen. Der inst-Operator passiviert Literale; wir sprechen deswegen oft von passivierten Literalen anstatt von denotativen Literalen. LISP-Programmierer erkennen hier eine Verwandtschaft von backquote in RELFUN und dem backquote oder auch *quote* in COMMON-LISP.

Konstanten	a, b, alpha, beta, ... (Ausnahme *unknown*)
Variablen	_[12], _v, _variable, _1, _2, ...
Strukturen	`(f a), `(f (g _x _y)), ...
Listen	`(tup 1 2 3 4), `(tup 1 2 (tup 3 4))

Bild 1.8 Denotative Literale

Ist *t* ein denotatives Literal, so sind implizit alle Argumente von *t* denotativ. Zum Beispiel hat die Struktur `(f (g _x _y)) als Argument die Struktur `(g _x _y). Dagegen können die Argumente eines evaluativen Literals denotativ oder evaluativ sein. Die Argumente eines evaluativen Literals werden *call-by-value*-artig ausgewertet. Eine Sonderstellung hat das Kopfliteral, denn dessen Argumente sind implizit passiviert. Das bedeutet, daß im Normalfall kein inst-Operator im Kopfliteral vorkommt.

Da der RELFUN-Interpreter in LISP geschrieben ist, auf LISP-Funktionen durchgegriffen werden, wenn es sich bei ihren Aufrufen um *abgeflachte* evaluative Literale mit denotativen Argumenten handelt. Es sind hierbei nicht alle LISP-Durchgriffe gestattet, sondern nur eine "notwendige" Auswahl. Hierunter fallen z. B. die arithmetischen Funktionen und die arithmetischen Prädikate. Weiterhin sind noch einige *print*-Funktionen zugelassen. Welche Funktionen verfügbar sind, ist in den RELFUN-Systemvariablen *lisp-functions*, *lisp-predicates* und *lisp-extras* vermerkt. In der jetzigen Version haben sie folgende Belegungen:

---

[10]Vergleichbar mit dem *fail* in PROLOG.
[11]Die Variable wird durch ihren Bindungskettenendwert der Laufzeitumgebung ersetzt.
[12]PROLOGs "_" für anonyme Variablen wird in RELFUN auch häufig mit "id" notiert.

lisp-functions := (+ - * / 1+ 1- sqrt exp)
lisp-predicates := (< <= = > >= atom numberp)
lisp-extras := (load rf-print rf-princ rf-terpri rf-fresh-line rf-pprint)

Einige Beispiele sollen im weiteren helfen, die Syntax von RELFUN zu illustrieren. Um die Unterschiede zu PROLOG deutlich herauszuarbeiten, werden RELFUN-Beispiele den entsprechenden PROLOG-Programmen gegenübergestellt.

Beispiel 1: Verwandtschaftsverhältnis

PROLOG-Version:
```
ancestor(mary, john).
ancestor(john, fred).
ancestor(fred, rosa).
ancestor(X, Y): - ancestor(X, H), ancestor(H, Y).
```

RELFUN-Version:
```
(hn (ancestor mary john) )
(hn (ancestor john fred) )
(hn (ancestor fred rosa) )
(ft (ancestor _x _y) (ancestor _x _h) (ancestor _h _y) )
```

Mögliche RELFUN-Anfragen[13] wären nun:

```
rfi>(ancestor fred rosa)
  true
rfi> (ancestor mary _y)
  true
  _y = john
rfi> more
  true
  _y = fred
rfi> more
  true
  _y = rosa
rfi> more
  unknown
```

[13]Die Beispielanfragen stehen nach dem RELFUN-Interpreter-Prompt "rfi>".

Bemerkung: Die RELFUN-Faktenbasis wird immer in Form von hn-Klauseln realisiert, Regeln dagegen oft als ft-Klauseln. Das RELFUN-Programm im obigen Beispiel hat einen ausschließlich relationalen Charakter, obwohl eine ft-Klausel vorkommt. Es ist aber leicht zu sehen, daß das Programm äquivalent zu einem Programm ist, welches entsteht, wenn man den einzigen ft-tag durch einen hn-tag ersetzt.

Beispiel 2: Fakultät

PROLOG-Version:
```
fak(0, 1).
fak(N, Res) :- H1 = N - 1, fak(H1, HRes), Res = N * HRes.
```

RELFUN-Version:
```
(ft (fak 0) 1)
(ft (fak _n) (* _n (fak (- _n 1))) )
```

```
rfi> (fak 1)
 1
rfi> (fak 5)
 120
rfi> (fak _x)
 _x = 0
 1
rfi> more
 → error
```

Bemerkung: An diesem Beispiel kann man erkennen, daß der funktionale Charakter der Fakultätsfunktion sehr gut in RELFUN modelliert werden kann. Natürlich hätte man auch eine relationale Beschreibung wählen können, ähnlich wie die PROLOG-Version. Zusätzlich sieht man an diesem Beispiel, wie LISP-Funktionen (hier: -, *) verwendet werden. Vor dem eigentlichen LISP-Durchgriff der Multiplikations-Funktion (*) wird ihr 2. Argument, ein rekursiver fak-Aufruf, herausgezogen (Abflachung s. u.) und als RELFUN-Funktion berechnet. Der Fehler beim *more*-Nachfordern während des Aufrufes *(f _x)* tritt auf, da die LISP-Differenzen-Funktion (-) mit der freien Variablen _x aufgerufen wird. In RELFUN sind aber nur Aufrufe von LISP-Funktionen mit grundinstantiierten[14] Termen erlaubt.

[14]Die Instantiierung der Terme resultiert in Grundterme. (Es kommen keine Variablen mehr vor).

Bisher wurde noch nicht das is-Primitiv von RELFUN behandelt. Es ist als built-in in RELFUN implementiert und stellt eine Verallgemeinerung vom "=/2" in PROLOG dar. Es hat folgende Syntax:

(is *lhs-is rhs-is*)

Ähnlich wie beim Kopfliteral ist die linke Seite vom is-Primitiv implizit passiviert, während die rechte Seite evaluativ oder denotativ sein kann. Je nachdem sprechen wir vom *denotativen is-Literal* oder *evaluativen is-Literal.* Das denotative is-Literal dient dazu, die Unifikation explizit zu machen, während man die evaluative Form dazu benutzt, Funktionswerte zu vergleichen oder zwischenzuspeichern. Die Semantik ist, daß die rechte Seite des is-Primitivs *rhs-is* ausgewertet wird und mit der linken Seite *lhs-is* unifiziert. Sind die Seiten miteinander unifizierbar, werden die entsprechenden neuen Bindungen vorgenommen und das is-Primitiv gibt den Wert der rechten Seite *rhs-is* zurück. Beispiel 3 zeigt das Verhalten des is-Primitivs auf dem Interpreter-Top-Level. Die aktuelle Datenbasis ist weiterhin das Fakultätsprogramm aus Beispiel 2.

Beispiel 3: Is-Primitiv (Datenbasis wie in Beispiel 2)

```
rfi> (is 1 (fak 1))
  1
rfi> (is _x (fak 5))
  _x = 120
120
rfi> (is (fak 5)  (fak 5))
  unknown
rfi> (is (fak 5)  `(fak 5))
  (fak 5)
rfi> (is (fak _x)  `(fak 5))
_x = 5
(fak 5)
```

Bemerkung: Im ersten Moment erscheint es erstaunlich, daß der is-Ausdruck *(is (fak 5) (fak 5)* scheitert und zu *unknown* führt. Es sei aber daran erinnert, daß die linke Seite im is-Ausdruck passiviert ist und die rechte Seite ausgewertet wird. Rechts steht in unserem Fall ein evaluatives Literal, welches zum Wert 120 evaluiert. Die Auswertung von *(is (fak 5) 120)* scheitert schließlich und liefert unknown.

Bei der Abarbeitung von goals erzeugt der RELFUN-Interpreter zur Laufzeit is-Ausdrücke, um geschachtelte evaluative Literale abzuflachen. Der Benutzer kann mit dem Befehl *flatten* erreichen, daß ein Programm statisch abgeflacht wird. Dies hat den Vorteil, daß der Interpreter zur Laufzeit von dieser Aufgabe befreit ist und damit einen *speed-up* erfährt. Beispiel 4 zeigt noch einmal die Datenbasis aus Beispiel 2, wie sie nach dem flatten-Befehl entstanden ist. Zum Auflisten der Datenbasis wird der Befehl *l* benutzt.

Beispiel 4: Abgeflachtes Fakultätsprogramm

```
rfi> flatten
rfi> l
  (ft (fak 0) 1 )
  (ft (fak _n) (is _1 (- _n 1) )
               (is _2 (fak _1) )
               (* _n _2) )
```

Der Befehl *flatten* flacht nur evaluative Literale ab. Um auch die denotativen Literale abzuflachen, existiert der Befehl *flatter*. Auf weitere Besonderheiten von RELFUN und insbesondere des is-Primitivs wird in Abschnitt 2 eingegangen.

2 Horizontale Transformation in den RELFUN-Kern

In Bereichen, in denen man Problemstellungen in einem gewissen Formalismus darstellen möchte, hat man oft das Problem, daß eine unüberschaubare Vielfalt von inhaltlich gleichbedeutenden, aber formal-syntaktisch unterschiedlichen Darstellungen existiert. Die Darstellungen sind hierbei problemorientiert und für die Berechnungsmechanismus nicht notwendigerweise die geeignetsten. Man definiert sich deswegen "ergebnisorientierte" Normalformen, die den Berechnungsmechanismus übergeben werden. Die problemorientierte Darstellung wird hierbei auf algorithmische Weise in die Normalform transformiert. Die Transformation stellt oft schon einen wesentlichen Lösungsschritt dar. Als Beispiel aus der Mathematik sei hier nur die Jordansche Normalform für Matrizenprobleme und aus der Prädikatenlogik 1. Stufe die Pränexe Normalform für logische Formeln genannt [Ric88].

In diesem Abschnitt beschäftigen wir uns mit Normalisierung einer RELFUN-Datenbasis anhand horizontaler Transformationen [Bol90b]. Es handelt sich um Klausel-orientierte Transformationen, d. h. die Klauseln werden hier nicht im Kontext einer Menge von Klauseln (etwa Prozedur- oder Datenbasis-Kontext), sondern isoliert betrachtet. Im Vergleich hierzu ist die in Beispiel 1 aus Abschnitt 1.2 erwähnte Transformation der Klausel-tags eine Prozedur-orientierte Transformation, da alle Klauseln der Prozedur ancestor/2 beachtet werden müssen. Ein *Partieller Evaluator* [Kah83] [NGC88] arbeitet noch oberhalb der der Prozedurebene und ist ein Beispiel für Datenbasis-orientierte Transformationen. Hier interessieren nicht nur die Pädikatdefinierenden Klauseln, sondern auch die Aufrufe der Prädikate (bzw. Prozeduren) in einer beliebigen Klausel der Datenbasis.

Die in dieser Arbeit untersuchten und entwickelten Klausel-orientierten Transformationen bedeuten eine Reduktion von allgemeinen RELFUN-Klauseln in *Kernklauseln*. Eine Kernklausel benutzt nur noch einen Teil der von RELFUN angebotenen Features. Den benutzten Teil von RELFUN nennen wir *RELFUN-Kern*. Die Kernklauseln haben, wie oben schon angedeutet, unter Umständen ein völlig anderes Aussehen als die ursprünglichen Klauseln und man kann nicht mehr in allen Fällen von einem gut leserlichen Programm sprechen. Dies war aber auch zu erwarten, da das lesbare Programm der problemorientierten Darstellung dient, während die Kernklauseln eine effiziente Weiterbearbeitung unterstützen und dies gleich auf zweierlei Art und Weise. Zum einen werden bei den Transformationen echte Optimierungen vorgenommen wie etwa die Eliminierung von denotativen Rumpfliteralen oder die vorgezogene statische Unifikation (Abschnitt 2.2.2.1 + 2.2.2.3). Zum anderen können die nachgeschalteten Komponenten wie beispielsweise der *Classifier* aus [Kra90] und der *mode-Interpreter* (Abschnitt 3) wesentlich einfacher gehalten werden, da sie viele Spezialfälle aus RELFUN nicht beachten brauchen, sondern auf den Kernklauseln arbeiten.

2.1 Normalisierung des denotativen is-Literals

Dieser Abschnitt behandelt die Normalisierung von denotativen is-Literalen. Zur Erinnerung: Ein is-Literal mit einer denotativen rechten Seite bezeichnen wir als denotatives is-Literal. Denotative is-Literale spielen bei der vertikalen Compilation von RELFUN in die WAM-Ebene (oder auch RFM-Ebene) eine wichtige Rolle, da sie *inline-Prädikate* darstellen. Der Begriff *inline-Prädikat* und in diesem Zusammenhang auch der *chunk*-Begriff sind von Saumya K. Debray [Deb86] geprägte Standardbegriffe der "WAM-Gemeinde" und sollen wegen ihrer Wichtigkeit für die folgenden Ausführungen kurz vorgestellt werden.

inline-Prädikat Ein inline-Prädikat ist ein Literal, welches durch eine Sequenz von WAM-Instruktionen (*inline-Code*) ausgeführt wird, ohne daß zu einer Prozedur verzweigt wird.

Das charakteristische an einem inline-Prädikat ist, daß es keinen Prozeduraufruf bewirkt. Insbesondere werden also keine WAM-Register "unkontrolliert" verändert und können deswegen über mehrere inline-Prädikat hinweg betrachtet werden. Offensichtlich ist es sinnvoll, mehrere inline-Prädikat zusammenzufassen, da auch bei der Aneinanderreihung kein Prozeduraufruf erfolgt. Dies definiert uns dann einen *chunk*.

chunk Ein chunk ist eine Folge von null oder mehreren inline-Prädikaten, gefolgt von höchstens einem nicht inline-Prädikat.

In RELFUN gibt es zwei besondere chunks. Zum einen der Kopfchunk, denn das Kopfliteral wird diesem chunk zugerechnet und zum anderen der letzte chunk einer Klausel mit denotativen foot-Literal. Es ist auch möglich, daß diese beiden besonderen chunks identisch sind, wie es im (funktionalen) Fakultätsbeispiel aus Abschnitt 1.2 der Fall ist.

Innerhalb eines chunks können eine Vielzahl von Optimierungen vorgenommen werden, die sich auf die Allokation von WAM-Registern beziehen. In den Ausführungen von Hans-Günther Hein [Hei91] sind diese Optimierungen beschrieben. In den Untersuchungen hat sich herausgestellt, daß die große syntaktischer Vielfalt von denotativen is-Literalen einen hohen Aufwand bei den Fallunterscheidungen zur Folge hat, wenn man alle Registeroptimierungen ausnützen möchte.

Dieselbe Problematik stellt sich bei allen anderen Programmtransformationen oder der globalen Datenflußanalyse (Abschnitt 3) ein. Aus diesem Grund wird eine is-Normalform definiert, um sie in den später beschriebenen Klauseltransformationen (Abschnitt 2.2) zu nutzen. Hierfür

definieren wir zuerst *Termattribute* (Abschnitt 2.1.2), die die linken und rechten Seiten eines is-Literals näher beschreiben. Im weiteren wird eine totale Ordnung $>_{ta}$ auf den Termattributen definiert. Sie gestattet es uns, eine Ordnung auf den is-Literalen anzugeben und damit zu jedem is-Literal *is-t* ein kleinstes, semantisch äquivalentes is-Literal *is-t'* zu finden.

2.1.1 Statische Bindungsumgebung

Bezüglich einer Abarbeitungsreihenfolge der Literale in einer Klausel wird das Termattribut (s. u.) eines Terms bestimmt. Das Termattribut kennzeichnet unter anderem, ob der Term vorher schon einmal in einem Literal vorgekommen ist und weiterhin, in was für einem Literal das gewesen ist. Die Abarbeitungsreihenfolge der Literale definieren wir als Literalsequenz *lit-seq*, von der wir auf jeden Fall verlangen wollen, daß sie verträglich mit den chunk-Grenzen ist. Kommt also ein Literal l_1 in einem späteren chunk vor als l_2, so steht l_1 auch in der *lit_seq* nach l_2. Mit einem "späteren" chunk ist ein chunk gemeint, der weiter rechts in der Klausel liegt.[15]

Wird eine Klausel bezüglich einer Literalsequenz *lit-seq* für Transformationszwecke abgearbeitet, können statische Bindungen von Variablen erkannt und über die Klausel propagiert werden. Um die erkannten Bindungen zu verwalten, werden zwei Regelsysteme *simple-bindings* und *complex-bindings* benutzt. In den beiden Regelsystemen ist die augenblickliche statische Bindungsumgebung an einer Klauselposition p in einer Literalsequenz *lit-seq* zu einer Klausel *C* repräsentiert.

Definition 2.1 *Statische Bindungsumgebung*

Das Regelsystem *s-bindings* (*simple-bindings*) ist ein interreduziertes[16] System von Regeln der Art

var → *simple-term*,

wobei keine Regel aus *s-bindings* auf eine rechte Seite einer weiteren Regel aus *s-bindings* anwendbar ist. Die linken Seiten der Regeln sind ausschließlich Variablen. Die rechten Seiten der Regeln können Variablen und Konstanten (*simple-terms*) sein.

---

[15]Im Abschnitt 2.2 werden die Literale anhand von Regelsystemen abgearbeitet. Dies geschieht chunkweise und wird von einem Kontrollalgorithmus gesteuert (Abschnitt 2.3). Die Literalsequenz *lit_seq* soll gerade die verschiedenen Reihenfolgen modellieren.

[16]Unter einem interreduzierten Regelsystem $\mathcal{R}$ wollen wir nach [Ave89] ein Regelsystem verstehen, in dem die rechten Seiten r aller Regeln l->r bezüglich $\mathcal{R}$ und alle linken Seiten l bezüglich $\mathcal{R}\backslash\{l\text{->}r\}$ irreduzibel sind. Da wir hier ein sehr einfaches Regelsystem haben, in welchem links nur Variablen stehen, bedeutet dies: Alle linken Seiten sind verschiedene Variablen, die in den Termen auf der rechten Seite nicht vorkommen.

Das Regelsystem *c-bindings* (*complex-bindings*) ist ein interreduziertes System von Regeln der Art

var → *complex-term*,

wobei *complex-term* eine Struktur oder Liste darstellt. Weiterhin sind die rechten Seiten der Regeln mit keiner Regel aus *s-bindings* reduzierbar.

Das System *bindings* ist die Liste bestehend aus den Elementen *s-bindings* und *c-bindings*.

Der Grund weshalb einfache und komplexe Bindungen unterschieden werden, ist auf der WAM-Ebene zu finden. Betrachten wir dazu die Klausel in Bild 2.1 (a). Die denotativen is-Literale *(is _v a)* und *(is _y _w)* erzeugen die Bindungsregeln _v → *a* und _y → _w, die in das System *s-bindings* aufgenommen werden. Dies wird statisch, durch partielle Auswertung des is-Literals, festgestellt. Setzen wir eine Literalsequenz *lit-seq* = (0, 1, ..., 5) voraus, die Literale werden also von links nach rechts abgearbeitet, ist intuitiv klar, daß es sinnvoll ist, diese Bindungen durch die restliche Klausel zu propagieren und durch *rewriting* der weiteren Vorkommen von _v und _y mit *s-bindings* explizit zu machen. Man erkennt dies in Bild 2.1 an den Literalen in den Klauselpositionen p = 4 und 5, wo die Literale *(prem _v _y _w)* in *(prem a _w _w)* und *(prem _v _v)* in *(prem a a)* umgeschrieben worden sind.

(ft (head) (prem (f _v _y)) (is _y _w) (is _v a) (prem _v _y _w) (prem _v _v)) (a)

(ft (head) (prem (f _v _y)) (is _y _w) (is _v a) (prem a _w _w) (prem a a)) (b)

Bild 2.1 RELFUN-Klausel (a) und äquivalente Klausel (b) mit *simpler* Bindungspropagierung

In Bild 2.2 betrachten wir den Fall des denotativen is-Literals *(is _v `(f a))*, welches die komplexe Bindung _v → `(f a) bewirkt. Die explizite Propagierung der Bindung durch rewriting kann je nach der zur Verfügung stehenden WAM-Realisierung einen negativen Effekt in Bezug auf die Laufzeit und die Speichernutzung haben. In der Klausel 2.2 (b) wird nämlich die Struktur `(f a) dreimal auf dem Heap aufgebaut. In der ursprünglichen Klausel wäre die Struktur nur einmal im Heap erzeugt worden; (a) ist also wesentlich effizienter, was den Speicherbedarf betrifft. Außerdem sind zum Aufbau der Strukturen auf dem Heap mehrere WAM-Instruktionen nötig, die in der Klausel 2.2 (b) zweimal zuviel ausgeführt werden. Zusammen entsteht also zusätzlicher Speicherbedarf für die redundanten Instruktionen und eine längere Laufzeit durch redundante Abarbeitung der Instruktionen.

```
(ft (head) (prem _v)  (is _v `(f a)) (prem _v _v) )                    (a)

(ft (head) (prem _v)  (is _v `(f a)) (prem `(f a) `(f a)) )            (b)
```

Bild 2.2 RELFUN-Klausel (a) und äquivalente Klausel mit (b) ***komplexer*** Bindungspropagierung

In den Regelsystemen zur Klauselnormalisierung (Abschnitt 2.2) werden deswegen Klauseln der Bauart 2.2 (b) in Klauseln der Form 2.2 (a) transformiert. Die in Bild 2.1 beschriebene Transformation wird dort ebenfalls aufgegriffen und weiter verfeinert.

Zu der statischen Bindungsumgebung *bindings* und einem Term t interessiert uns die *Bindungs-Normalfom* t_N. Ist das System *bindings* fest, dann sind auch die Regelsysteme *s-bindings* und *c-bindings* gegeben und wir berechnen t_N folgendermaßen:

$$t_N := (t\downarrow_{s\text{-}bindings})\downarrow_{c\text{-}bindings}$$

Wobei mit der Schreibweise $t\downarrow_{\mathcal{R}}$ derjenige Term gemeint ist, der entsteht, wenn der Term t solange mit Regeln aus $\mathcal{R}$ reduziert wird, bis keine Regel aus $\mathcal{R}$ mehr anwendbar ist. Man beachte, daß nach der Definition von *s-bindings* und *c-bindings*, jeweils höchstens eine Regel aus den Regelsystemen anwendbar ist, falls es sich bei t um eine Variable handelt. Entsprechend kann die maximale Anzahl von Regelanwendungen anhand der Variablen im Term t berechnet werden. Daß zur Gewinnung der Term-Normalform zuerst mit *s-bindings* und dann mit *c-bindings* reduziert wird, begründet sich aus der Definition von *c-bindings*, denn hier wird verlangt, daß *c-bindings* bezügliche *s-bindings* irreduzibel ist.

2.1.2 Termattribute

In Abschnitt 1 wurden bereits die denotativen Terme (bzw. Literale) eingeführt. Zu ihnen gehören die Konstanten, Variablen, Strukturen und Listen. Hier interessieren wir uns für ihr Auftreten innerhalb eines is-Literals und in diesem Zusammenhang ist es zum Beispiel wichtig zu wissen, ob eine Variable bereits gebunden ist oder zum erstenmal in der Klausel vorkommt. Die nötigen Unterscheidungsmerkmale führen zu einer Menge $\mathcal{TS}$ von Termattributen *ta*. Das Attribut für einen denotativen Term ist abhängig von der Klauselposition und der Literalsequenz. Es ist z. B. möglich, daß die Variable _v im Literal mit der Position p ein unterschiedliches Attribut erhält, als an der Klauselposition p'.

Definition 2.2 *Termattribute*

Die Menge $\mathcal{TS}$ (Term attribute Set) ist gegeben durch:

$$\mathcal{TS} := \{\mathcal{TA}\text{-}Const,\ \mathcal{TA}\text{-}Struc,\ \mathcal{TA}\text{-}Tup,\ \mathcal{TA}\text{-}Var,\ \mathcal{TA}\text{-}FirstVar,\ \mathcal{TA}\text{-}FreeVar,\ \mathcal{TA}\text{-}Eval\}$$

Die Elemente aus $\mathcal{TS}$ werden als Termattribute *ta* bezeichnet. Über die Menge $\mathcal{TS}$ *wird* die **Termattributordnung** $>_{ta}$ definiert:

$$\mathcal{TA}\text{-}FirstVar >_{ta} \mathcal{TA}\text{-}FreeVar >_{ta} \mathcal{TA}\text{-}Var >_{ta} \mathcal{TA}\text{-}StrucVar >_{ta} \mathcal{TA}\text{-}Const >_{ta} \mathcal{TA}\text{-}tup >_{ta} \mathcal{TA}\text{-}Struc >_{ta} \mathcal{TA}\text{-}Eval$$

Im weiteren soll mit Hilfe des Systems $(\mathcal{TS}, >_{ta})$ is-Literale gerichtet werden, in der Art, daß die rechten nicht größer als linken Seiten sind. Dazu benötigen wir noch eine Abbildungsfunktion f_A von den Termen auf die Menge $\mathcal{TS}$, die jedem Term *t* ein eindeutiges Attribut *ta* zuordnet. Die Funktion f_A wird in Definition 2.3 informell beschrieben.

Definition 2.3 *Attributsfunktion* f_A

Sei die Literalsequenz *lit-seq* für eine Klausel *C* fest und das System *bindings* zur Klauselposition p gegeben. Dann gilt für einen Term *t* an der Klauselposition p:

$f_A(t) =$

$\mathcal{TA}\text{-}Const$	falls $t\downarrow_{s\text{-}bindings}$ eine RELFUN-Konstante[17] ist.
$\mathcal{TA}\text{-}Struc$	falls *t* eine RELFUN-Struktur ist.
$\mathcal{TA}\text{-}Tup$	falls *t* eine RELFUN-Liste ist.
$\mathcal{TA}\text{-}FirstVar$	falls *t* eine Variable in einem Literal mit der Position p ist, und es gibt kein Literal *lit* an der Klauselposition p', mit p' hat sein Vorkommen vor p bezüglich *lit-seq* und *t* kommt in *lit* vor.
$\mathcal{TA}\text{-}FreeVar$	falls *t* eine Variable ist, die nicht das Attribut $\mathcal{TA}\text{-}FirstVar$ hat und auf keiner linken Seite von *bindings* vorkommt.

[17] Die Frage, wie Konstanten, Strukturen, ... in RELFUN aussehen, wurde in Abschnitt 1.2 behandelt.

TA-StrucVar	falls *t* eine Variable ist und auf der linken Seite einer Regel aus *c-bindings* vorkommt.
TA-Var	falls *t* eine Variable ist und auf der linken Seite einer Regel aus *s-bindings* vorkommt und die rechte Seite der Regel keine Konstante ist.
TA-Eval	falls *t* evaluativ ist.

Offensichtlich ist die Funktion f_A total auf den Termen, denn falls ein Term *t* evaluativ ist, trifft nur der letzte Fall zu; der Term *t* hat das Attribut *TA-Eval*. Ist *t* denotative und keine Variable, so trifft genau einer der ersten drei Fallunterscheidungen zu. Falls *t* eine Variable ist, für die eine statische Bindung an eine Konstante festgestellt wurde, hat es das Attribut *TA-Const*. Kommt die Variable zum erstenmal vor, hat sie das Attribut *TA-FirstVar*, sowie das Attribut *TA-FreeVar*, wenn sie in vorherigen Vorkommen keine Bindungsregeln bewirkt hat, also z. B. in einem evaluativen Literal vorgekommen ist. Falls dies nicht der Fall ist, kann es sich nur noch um eine Bindung an einer Struktur (*TA-StrucVar*) oder einer Variablen handeln (*TA-Var*).

Mit der Funktion f_A können wir nun zu jedem is-Literal *is-t* eine eindeutige *is-Normalform* *is-t'* angeben, die semantisch äquivalent zu *is-t* ist und für die gilt: *is-t* $\geq_{is}$ *is-t'*, und es gibt kein *is-t''* mit *is-t'* $>_{is}$ *is-t''*. Hierfür wird die Ordnung $>_{is}$ auf den is-Literalen definiert. Zunächst führen wir noch die Funktion **mk-inst: TERM → TERM** ein, die die implizite Passivierung einer rechten Seite eines is-Literals explizit macht. Es gilt also zum Beispiel: mk-inst(_v) = _v und mk-inst((f a)) = `(f a). Entsprechend bezeichnen wir die Umkehrung der Funktion mit **un-inst: TERM → TERM.** Es gilt zum Beispiel: un-inst(_v) = _v und un-inst(`(f a)) = (f a).

Definition 2.4

Seien *C*, *lit-seq* und *bindings* gegeben.
Weiter sei (is *lhs-is rhs-is*) das Literal *is-t* und (is *lhs-is' rhs-is'*) das Literal *is-t'*.
Es gilt nun:

is-t $>_{is}$ *is-t'* gdw mk-inst(*lhs-is*) ≡ *rhs-is'* und *rhs-is* ≡ mk-inst(*lhs-is'*)

und f_A(mk-inst(*lhs-is'*)) $>_{ta}$ f_A(*lhs-is*)

Wir schreiben *is-t* $\geq_{is}$ *is-t'*, falls gilt: *is-t* $>_{is}$ *is-t'* oder *is-t* ≡ *is-t'* und bezeichnen ein bezüglich $>_{is}$ minimales is-Literal als **is-Normalform.**

Durch die is-Normalform wird erreicht, daß bei den normalisierten is-Literalen auf der linken Seite vornehmlich Variablen stehen und damit den instantiierenden Charakter des is-Literals ausdrücken. Weiterhin haben Variablen, die zum erstenmal vorkommen oder noch nicht statisch gebunden sind, ein größeres Gewicht und kommen deswegen immer auf der linken Seite der is-Normalform vor.

Ein is-Literal (is *lhs-is rhs-is*) ist nach Definition 2.4 in is-Normalform, falls f_A(mk-inst(*lhs-is*)) $\geq_{ta}$ f_A(*rhs-is*) gilt. Ist dies nicht der Fall, so ist die Normalform durch (is uninst(*rhs-is*) mk-inst(*lhs-is*)) gegeben. Ein evaluatives is-Literal ist immer eine is-Normalform, da die linke Seite des is-Literals das Termattribut *TA-Eval* hat und *TA-Eval* das kleinste Element bezüglich $>_{ta}$ ist. Die Termattribute werden also insbesondere zur Normalisierung von denotativen is-Literalen gebraucht.

Ein Beispiel soll diesen Abschnitt abschließen. Hierzu betrachten wir eine einfache ft-Klausel *C* und ermitteln schrittweise für die is-Literale in der Klausel deren Normalform (fettgedruckt), so daß die Klausel *C'* entsteht, in welcher nur noch is-Normalformen vorkommen. Die Abarbeitung der Klausel wird durch *lit-seq* vorgegeben. An Klauselpositionen p, an denen ein is-Literal vorkommt, wird ein "Transformationsschnappschuß" gezeigt. Es wird nicht näher darauf eingegangen, wie die jeweiligen Bindungsumgebungen entstehen und sie werden auch noch nicht propagiert. Es ist jedoch intuitiv klar, wie sie berechnet werden. Die Literalsequenz wird so gewählt, daß immer zuerst die denotativen Literale des chunks (inline-Prädikate) abgearbeitet werden.

C: (ft (head _v_1 _v_2) (is a _w_1) (is _v_1 `(f a)) (is _v_1 _v_2) ($prem_1$ _w_1)
($prem_2$ _w_1) (is _v_2 _w_1))
lit-seq = (1, 2, 3, 0, 4, 5, 6), *bindings* = (Ø, Ø)

p = 1: $is\text{-}t \equiv$ (is a $\_w_1$), *bindings* = ($\emptyset$, $\emptyset$)
f_A(a) = $\mathcal{TA}\text{-}Const <_{ta} \mathcal{TA}\text{-}FirstVar = f_A(\_w_1) \rightarrow is\text{-}t' \equiv$ **(is $\_w_1$ a)**

p = 2: $is\text{-}t \equiv$ (is $\_v_1$ `(f a)), *bindings* = ({$\_w_1$ ->a}, $\emptyset$)
$f_A(\_v_1) = \mathcal{TA}\text{-}FirstVar >_{ta} \mathcal{TA}\text{-}Struc = f_A$(`(f a)) $\rightarrow is\text{-}t' \equiv$ **(is $\_v_1$ `(f a))**

p = 3: $is\text{-}t \equiv$ (is $\_v_1$ $\_v_2$), *bindings* = ({$\_w_1$ ->a}, {$\_v_1$->`(f a)})
$f_A(\_v_1) = \mathcal{TA}\text{-}StrucVar <_{ta} \mathcal{TA}\text{-}FirstVar = f_A(\_v_2) \rightarrow is\text{-}t' \equiv$ **(is $\_v_2$ $\_v_1$)**

p = 5: $is\text{-}t \equiv$ (is $\_v_2$ $\_w_1$), *bindings* = ({$\_w_1$ ->a, $\_v_2$->$\_v_1$}, {$\_v_1$->`(f a)})
$f_A(\_v_1) = \mathcal{TA}\text{-}Var >_{ta} \mathcal{TA}\text{-}Const = f_A(\_w_1) \rightarrow is\text{-}t' \equiv$ **(is $\_v_2$ $\_w_1$)**

=> C: (ft (head $\_v_1$ $\_v_2$) (is $\_w_1$ a) (is $\_v_1$ `(f a)) (is $\_v_2$ $\_v_1$) ($prem_1$ $\_w_1$)
(is $\_v_2$ $\_w_1$) ($prem_2$ $\_w_1$))

2.2 Reduktion auf den RELFUN-Kern

Dieser Abschnitt definiert Regeln, die zusammen mit einem Kontrollalgorithmus (Abschnitt 2.3) eine beliebige RELFUN-Klausel in eine *Kern-Klausel*[18] transformieren. Die Regeln haben als Eingabe eine Klausel (und meistens eine zusätzliche Bindungsumgebung (siehe später)) und werden über Literale mit gewissen Eigenschaften *getriggert* und liefern wieder eine Klausel (gegebenfalls mit neuer Bindungsumgebung) zurück. Wir sprechen von Kern-Klauseln, da die Transformationen einige Sprachkonstrukte, die RELFUN normalerweise erlaubt, auf eingeschränkte und normalisierte Sprachkonstrukte zurückführen. Mit einer Kern-Klausel assoziieren wir zunächst eine *flache* Klausel, in der Hinsicht, daß keine geschachtelten Prozeduraufrufe in der Klausel existieren [Bol90a]. Weiterhin kommen passive Strukturen nur mit einer Schachtelungstiefe von 1 vor. Sie treten auch nur auf der rechten Seite eines is-Literals oder als Klauselfuß auf. Bis hierher definiert eine solche Klausel die *flatter-Normalform*. Die flatter-Normalform wird durch das Regelsystem *SFL* (*static flattening*) erzeugt. In Abschnitt 2.2.1 ist das Regelsystem *SFL* beschrieben.

Die flatter-Normalform wird durch weitere Regeln (Abschnitt 2.2.2 - 2.2.6) in eine Kern-Klausel transformiert. Die Eigenschaften einer Kern-Klausel beinhalten zusätzlich, daß denotative Literale nur als Klauselfuß vorkommen, eine Struktur höchstens einmal vorkommt

[18]Wir unterscheiden im weiteren zwar immer zwischen RELFUN- und Kern-Klauseln, doch sollte man sich klar machen, daß Kern-Klauseln auch RELFUN-Klauseln sind. Die Umkehrung gilt natürlich nicht.

(*structure sharing*), alle is-Literale in der is-Normalform sind, die statische Bindung einer Variablen an eine Konstante oder andere Variablen durch die Klausel propagiert wird, deterministische Funktionen[19] nur einmal berechnet werden und dies durch Vorauswertung möglichst schon zur Transformationszeit geschieht (*common subexpression extraction*) (*partial evaluation* und die Unifikation explizit gemacht wird, soweit es statisch möglich ist, wobei ein Scheitern der Unifikation die Generierung und Propagierung von *unknown* nach sich zieht.

Bild 2.3 zeigt eine RELFUN-Klausel C und die zugehörige Kern-Klausel C'. Hier haben die Regeln zum Abflachen von Klauseln sowie zur statischen Unifikation und der Propagierung der Bindungen Anwendung gefunden. Weitere Beispiele und Motivationen für die einzelnen Regeln sind in den entsprechenden Abschnitten enthalten.

C: (ft (h (f b) (f _x) _x) (is (g _x) `(g b)) (prem _x)) (a)

C': (ft (h _1 _1 b) (is _1 (f b)) (prem b)) (b)

Bild 2.3 RELFUN-Klausel (a) und zugehörige Kern-Klausel (b)

2.2.1 Die flatter-Normalform

In RELFUN sind beliebige Schachtelungstiefen für evaluative und denotative Literale erlaubt. Der RELFUN-Interpreter flacht evaluative Literale dynamisch ab, damit auch nicht-deterministische evaluative Argumente behandelt werden und via backtracking mehrere Werte liefern können. Dies kann innerhalb eines Programms häufig vorkommen, wenn der Programmierer die funktionalen Aspekte von RELFUN bei der Programmentwicklung bevorzugt. Das Fakultäts-Programm aus Abschnitt 1.2 ist dafür ein Beispiel. Seit dem Beginn von RELFUN gibt es deswegen auch einen statischen Abflacher als Interpreter-Vorverarbeitungskommando (flatten).

Denotative Literale werden zwar vom Interpreter nicht abgeflacht, aber das Compilersystem setzt voraus, daß alle denotativen Literale flach sind. Jeder WAM-Codegenerator muß irgendwann die Strukturen abflachen, um die nötigen *get_structure*- und *unify*-Instruktionen für die Strukturen zu erzeugen. Unser Ansatz, diesen Schritt schon auf Klauselebene zu vollziehen, hat den Vorteil, daß wir auf höhere Ebene daran anschließende Optimierungen vornehmen können. Beispiele dafür sind die optimierende Registerallokation und das "sharen" von Teilstrukturen. Außerdem besteht immer die Möglichkeit, daß ein Programmentwickler oder ein höheres Entwicklungstool eine RELFUN-Klausel in solch einer Form vorgibt. Für ein beliebi-

[19]Soweit auf dieser Ebene entscheidbar ist, ob eine RELFUN-Funktion deterministisch ist. Zur Zeit werden die LISP-Durchgriffe auf diese Art und Weise behandelt.

ges Compilersystem könnte dies ein unangenehmer Sonderfall sein, der nicht optimal behandelt wird oder einen beträchtigen Mehraufwand, z. B. in Form von zusätzlichen Compilercode, bedeutet. Unser Ansatz ist sehr flexibel gegenüber Sonderfälle , da sie immer in eine definierte Normalform transformiert wird, auf welcher weitere Optimierungen aufsetzen.

Das Regelsystem *SFL* (eine Erweiterung eines Systems in [Bol90a]) unterscheidet in seinen Regeln, ob das eingebettete Literal im Kopf (Regel SFL_1), in einem is-Literal (Regeln SFL_2 bis SFL_5) oder einem anderen Rumpfliteral (SFL_5 bis SFL_6) vorkommt.

Definition 2.5 *SFL (static flattening)*

SFL_1[20]

$(ct\ (h \ldots t \ldots) \ldots)$ — Falls *t* **keine** Variable oder Konstante ist.

$(ct\ (h \ldots \_k \ldots)\ (\text{is}\ \_k\ `t) \ldots)$ — Die Variable *_k* wird neu erzeugt.

SFL_2

$(ct\ head \ldots (\text{is}\ r\ (f \ldots (\text{is}\ p\ q) \ldots)) \ldots)$

$(ct\ head \ldots\ (\text{is}\ p\ q)\ (\text{is}\ r\ (f \ldots p' \ldots)) \ldots)$ — Mit p' := mk-inst(p)[21]

SFL_3[22]

$(ct\ head \ldots (\text{is}\ r\ \overset{\curvearrowleft}{(}f \ldots t \ldots)) \ldots)$ — Falls *t* **keine** Variable oder Konstante ist.

$(ct\ head \ldots (\text{is}\ \_k\ \overset{\curvearrowleft}{(}t)\ (\text{is}\ r\ \overset{\curvearrowleft}{(}f \ldots \_k \ldots)) \ldots)$ — Die Variable *_k* wird neu erzeugt.

SFL_4

$(ct\ head \ldots (\text{is}\ r\ \ (\text{is}\ p\ q)) \ldots)$ — Mit p' := mk-inst(p)

$(ct\ head \ldots\ (\text{is}\ p\ q)\ (\text{is}\ r\ p') \ldots)$

[20] Man erinnere sich, daß Terme im Kopf implizit passiviert sind. Wird ein Term aus dem Kopf herausgezogen, muß die Passivierung durch den inst-Operator explizit gemacht werden.

[21] Siehe Abschnitt 2.1

[22] Die Notation $\overset{\curvearrowleft}{(}f \ldots)$ soll besagen, daß nur wenn in der Ausgangsklausel der inst-Operator steht, kommt er auch in der resultierenden Klausel an den entsprechenden Stellen vor.

SFL$_5$

$$\frac{(ct\ head \ldots (f \ldots\ (\text{is}\ p\ q) \ldots) \ldots)}{(ct\ head \ldots\ (\text{is}\ p\ q)\ (f \ldots\ p' \ldots) \ldots)}$$

Mit p' := mk-inst(p)

SFL$_6$

$$\frac{(ct\ head \ldots\ {}^{\wedge}/(f \ldots t \ldots) \ldots)}{(ct\ head \ldots\ (\text{is}\ \_k\ {}^{\wedge}/t)\ {}^{\wedge}/(f \ldots \_k \ldots) \ldots)}$$

Falls t **keine** Variable oder Konstante ist.

Die Variable $\_k$ wird neu erzeugt.

Die Klauseltransformation von C in die flatter-Normalform C' mit Hilfe von $\mathcal{SFL}$ wird durch eine endliche Ableitungsfolge $(C_0, C_1, \ldots, C_n)$ beschrieben mit $C = C_0$ und $C' = C_n$. Eine Ableitungsfolge nennen wir korrekt relativ zu einem Regelsystem $\mathcal{R}$, wenn man C_{i+1} von C_i aus durch Anwenden einer Regel aus $\mathcal{R}$ erreichen kann. Die Termination von Regelsystemen ist im allgemeinen nicht entscheidbar, aber wünschenswerte Voraussetzung, wenn man sie für Klauseltransformationen einsetzen möchte. Das Regelsystem $\mathcal{SFL}$ ist ein Beispiel für ein terminierendes Regelsystem. Ein formaler Beweis der Termination würde allerdings den Rahmen dieser Arbeit sprengen. Die Beweisidee besteht darin, eine wohlfundierte (Noethersche) Ordnung über die Komplexität der Literale (etwa die Anzahl der Symbole in einem Literal kombiniert mit ihrer Schachtelungstiefe) in einer Klausel anzugeben. Anhand dieser Ordnung zeigt man, daß jede Regelanwendung die Klausel bezüglich der Ordnung verkleinert. Die Wohlfundiertheit sichert, daß nur endlich viele Regeln nacheinander angewendet werden können.

Hat man kein (nachweisbar) terminierendes Regelsystem, reicht es, zumindest für unsere Zwecke, eine Kontrollstrategie anzugeben, die die Termination garantieren kann. Eine Kontrollstrategie gibt in gewissen Grenzen vor, welche Regeln in welcher Reihenfolge angewendet werden. Bei den später noch zu definierenden Regeln haben wir mit diesem Problem zu tun. Es werden nämlich Regeln definiert, die eine unendliche Folge von Regelanwendungen nach sich ziehen würden, hätte man keine Kontrollstrategie, die hier die Endlosschleife verhindert.

Eine weitere wichtige Frage ist die Korrektheit des Regelsystems. Für $\mathcal{SFL}$ muß also geklärt werden, ob die Semantik der flatter-Normalform der Semantik der ursprünglichen Klausel entspricht. Die Semantik einer RELFUN-Klausel ist durch den Interpreter gegeben. Offensichtlich reicht der Nachweis, daß die einmalige Anwendung einer Regel aus $\mathcal{SFL}$ die Semantik der Klausel nicht ändert. Eigentlich braucht auch nur das Abflachen der passiven Strukturen untersucht werden, da die Semantik der geschachtelten, evaluativen Aufrufe im Interpreter durch die dynamischen Abflachungsregeln definiert ist (Siehe hierzu auch

Abschnitt 1.2). Für die Regel SFL_1 wird exemplarisch begründet, warum auch die Regeln für die passiven Strukturen korrekt sind[23]. Die Regel SFL_1 lautet in einer etwas informativeren Syntax als in Definition 2.5:

SFL_1

$$\frac{(ct^{24}\ (h \ldots t \ldots)\ p_1 \ldots p_n)}{(ct\ (h \ldots \_k \ldots)\ (\text{is}\ \_k\ `t)\ p_1 \ldots p_n)}$$

Falls *t* **keine** Variable oder Konstante ist.

Die Variable *_k* wird neu erzeugt.

Wird die Klausel $C = (ct\ (h \ldots t \ldots)\ p_1 \ldots p_n)$ zur Laufzeit durch ein goal *g* aufgerufen, so sind *g* und das Kopfliteral von *C* miteinander unifizierbar. Mit den Bindungen, die bei der Unifikation entstehen, wird versucht, die Prämisse p_1 zu beweisen. Für die Korrektheit der Regel SFL_1 muß gezeigt werden, daß für $C' = (ct\ (h \ldots \_k \ldots)\ (\text{is}\ \_k\ `t)\ p_1 \ldots p_n)$ gilt:

1. Jedes goal *g*, das mit dem Kopfliteral von *C* unifizierbar ist, ist auch mit dem Kopfliteral von *C'* unifizierbar.

2. Gibt es ein goal g', welches mit dem Kopfliteral von *C'* unifizierbar ist, aber nicht mit dem Kopfliteral von *C*, so scheitert das is-Literal (is _k `t).

3. Die Bindungsumgebung in *C'*, bevor die Prämisse p_1 aufgerufen wird, ist identisch, bis auf die Bindung der Variablen *_k*, die nicht in *C* vorkommt, mit der Bindungsumgebung in *C* an der entsprechenden Stelle.

Die Bedingung 1 garantiert, daß die neue Klausel *C'* für alle goals aufrufbar bleibt, die es auch für *C* waren. Bedingung 2 garantiert, daß die Verallgemeinerung im Kopfliteral von *C'* nicht zu einer Lösung für ein neues goal g' führen kann. In die gleiche Richtung zielt Bedingung 3: Da die Variable *_k* nicht in *C* (d. h auch nicht in p_1 bis p_n) vorkommt, ist das Verhalten zur Laufzeit ab dem Literal p_1 in beiden Klauseln identisch. Bedingungen 1 bis 3 garantieren also, daß die Semantik der beiden Klauseln relativ zum Interpreter gleich ist.

Es bleibt noch zu klären, ob die Regel SFL_1 die drei Bedingungen garantiert. Das kann man sich aber schnell klar machen, denn zum einen subsumiert das Kopfliteral von *C'* das von *C* (Bedingung 1 gilt!) und zum anderen bedeutet ein Aufruf von *C'*, der nicht auch *C* matchen würde, daß die Variable *_k* an einen Term *t'* gebunden wird, der nicht mit *t* unifizierbar ist. Wäre dies nicht der Fall, würde der Aufruf auch *C* matchen. Nun scheitert aber das is-Literal (is *_k t*), denn *_k ist* nach Voraussetzung an *t'* gebunden und *t'* ist nicht mit *t* unifizierbar. Somit gilt auch Bedingung 2. Die Bedingung 3 folgt ebenfalls aus diesen Überlegungen.

---

[23] Das SFL-System ist allerdings nicht auf Klauseln mit (initialen) Cut-Operator anwendbar.

[24] *ct* steht für *clause tag* und besagt, daß an dieser Stelle sowohl hn als auch ft stehen kann.

Das System $\mathcal{SFL}$ braucht keine speziellen Kontrollstrategie, da jede beliebige Reihenfolge von Regelanwendungen aus $\mathcal{SFL}$ zur flatter-Normalform führt. In Bild 2.4 wird beispielhaft die Anwendung einiger Regeln aus $\mathcal{SFL}$ illustriert. Das Beispiel mag im ersten Moment etwas sonderbar erscheinen. Es soll aber auch als Beispiel für die später zu definierenden Regeln dienen und berücksichtigt teilweise deren Besonderheiten. Das Literal, welches die Regel triggert, wurde der Übersicht wegen fettgedruckt.

(ft (foo **(f b)** (f _x) _x) (is _y `(f b)) (is _y `(f _x)) (p _y _x (is a a)))
$\longrightarrow_{SFL_1}$
(ft (foo _1 **(f _x)** _x) (is _1 `(f b)) (is _y `(f b))
(is _y `(f _x)) (p _y _x (is a a)))

$\longrightarrow_{SFL_1}$ (ft (foo _1 _2 _x) (is _2 `(f _x)) (is _1 `(f b)) (i s _y `(f b))
(is _y `(f _x)) (p _y _x **(is a a)**))

$\longrightarrow_{SFL_5}$ (ft (foo _1 _2 _x) (_2 `(f _x)) (is _1 `(f b)) (is _y `(f b))
(is _y `(f _x)) (is a a) (p _y _x a))

2.2.2 Weitere Transformationsregeln

Nachdem RELFUN-Klauseln abgeflacht sind und die flatter-Normalform haben, sollen sie durch weitere Transformationsregeln in den RELFUN-Kern überführt werden. Für die folgenden Regelsysteme wird deswegen vorausgesetzt, daß die Eingabe-Klauseln flach sind und alle is-Literale die is-Normalform (Abschnitt 2.1) haben.

2.2.2.1 Eliminieren von denotativen Rumpfliteralen

In einer RELFUN-Klausel darf an beliebiger Literalposition, mit Ausnahme der Kopfposition ein denotatives Literal stehen. Es hat dann die Bedeutung einer Prämisse, die zur Laufzeit evaluiert wird und sich selbst denotiert. Da hierbei kein Prozeduraufruf erfolgt, kann eine solche Prämisse weder scheitern noch eine Bindung einer Variablen bewirken. Eine denotative Prämisse hat demnach keine Auswirkung auf die Semantik der Klausel, wenn sie als Rumpfliteral in der Klausel vorkommt. Handelt es sich allerdings um eine ft-Klausel, dann bestimmt ein denotatives Literal in der foot-Position den Wert der gesamten Klausel. Diese

Überlegungen resultieren in das Regelsystem $\mathcal{DBE}$. Die komplexe Bindungsumgebung $\mathcal{B}_C$ soll hier nicht weiter interessieren; sie wird im nächsten Abschnitt behandelt.

Definition 2.6 *$\mathcal{DBE}$ (denotative-body elimination)*

DBE_1

$$\frac{\text{(ft } \textit{head} \ldots d \ \ldots \text{ e)} \ , \mathcal{B}_C}{\text{(ft } \textit{head} \ldots \ldots \text{ e)} \ \ , \mathcal{B}_C}$$

Falls d denotativ ist

DBE_2

$$\frac{\text{(hn } \textit{head} \ldots d \ldots) \ , \mathcal{B}_C}{\text{(hn } \textit{head} \ldots \ldots) \ \ , \mathcal{B}_C}$$

Falls d denotativ ist

Die Regel DBE_1 eliminiert alle denotativen Literale von ft-Klauseln, die nicht an der foot-Position vorkommen. Die Regel DBE_2 eliminiert alle denotativen Literale von hn-Klauseln. Zur Erinnerung sei gesagt, daß hn-Klauseln bei erfolgreicher Abarbeitung implizit den Wert *true* denotieren. Das folgende Beispiel illustriert die Anwendung von $\mathcal{DBE}$.

(hn (p _w) `(f a) (q a) a)

$\longrightarrow_{DBE2}$ (hn (p _w) (q a) a)

$\longrightarrow_{DBE2}$ (hn (p _w) (q a))

Hätte es sich um eine ft-Klausel gehandelt, so wäre die Regel DBE_1 nur einmal anwendbar gewesen. Die Termination des Regelsystems $\mathcal{DBE}$ ist offensichtlich, da bei jeder Anwendung einer Regel die Anzahl der Literale in der Klausel verkleinert wird. Die Korrektheit von $\mathcal{DBE}$ ist ebenfalls offensichtlich und wurde bereits anfangs des Abschnitts skizziert.

2.2.2.2 Propagieren denotativer *is-rhs*-Terme

Is-Literale werden häufig dazu benutzt, ähnlich wie beim is-Primitiv in PROLOG, Variablen an Terme zu binden. In PROLOG sind dies üblicherweise arithmetische Ausdrücke, in RELFUN beliebige Ausdrücke. Als spezielles is-Literal haben wir das denotative is-Literal kennengelernt. In diesem Zusammenhang wurden auch die Bindungsumgebungen *bindings* = (*s-bindings c-bindings*) definiert (Def. 2.1). Die dort angedeuteten Ideen zur Propagierung von

Variablenbindungen, die aus der Auswertung der denotativen is–Literale resultieren (partial evaluation), werden nun formalisiert.

Das Regelsystem $\mathcal{DIP}$ faßt die beiden zentralen Ideen zusammen. Die Regeln DIP_1 und DIP_2 propagieren Bindungen von Variablen an Konstanten und anderen Variablen explizit über die Klausel, indem sie weitere Vorkommen der Variablen rechts vom is-Literal und innerhalb des aktuellen chunks durch die entsprechende Bindung ersetzen. Erzeugt ein is-Literal eine komplexe Bindung, so wird diese nicht explizit durch rewriting propagiert, sondern in die Bindungsumgebung $\mathcal{B}_C$ aufgenommen (Regel DIP_3). Die Bindungsumgebung $\mathcal{B}_C$ entspricht dem Regelsystem *c-bindings* aus Abschnitt 2.1. Am Ende der Klauseltransformation braucht die Bindungsumgebung nicht mehr beachtet werden, da sie die Aufgabe hat, statische Bindungen zu verwalten, die in der Klausel weiterhin implizit Vorhanden sind. Eine Kern-Klausel ist somit ohne $\mathcal{B}_C$ definiert.

Definition 2.7 $\mathcal{DIP}$ (*denotative is-rhs propagation*)

DIP_1

$$\frac{(ct\ head \ldots (\text{is } \_v\ t)\ e_1 \ldots e_n),\ \mathcal{B}_C}{(ct\ head \ldots t\ e'_1 \ldots e'_n),\quad \mathcal{B}_C}$$

Falls $f_A(\_v) = \mathcal{TA}\text{-}FirstVar$[25] und t ist eine Variable oder Konstante.

Es gilt dann: $e'_i = e_i\downarrow_{(\_v \rightarrow t)}$

[25] f_A ist die in Abschnitt 2.1.2 (Def 2.3) definierte Attributsfunktion.

DIP$_2$

$$\frac{(ct\ head \ldots a_1 \ldots (\text{is } \_v\ t) \ \ldots a_m\, e_1 \ldots e_n),\ \mathcal{B}_C}{(ct\ head' \ldots a'_1 \ldots f\ (\text{is } \_v\ t) \ldots a'_m\ e'_1 \ldots e'_n),\ \mathcal{B}_C'}$$

Falls $f_A(\_v) = \mathcal{TA}\text{-}\mathcal{FreeVar}$ und $\_v$ kommt in a_i, e_i oder auf der rechten Seite einer Regel aus $\mathcal{B}_C$ vor und t ist eine Variable oder Konstante.
Weiter bilden $a_1 \ldots (\text{is } \_v\ t) \ldots a_m$ einen chunk.

Es gilt dann:

$\mathcal{B}_C'' = \mathcal{B}_C \downarrow_{\{\_v \to t \downarrow_{\mathcal{B}_C}\}}$

$head' = head \downarrow_{\{\_v \to t\}}$, falls a_1 zum Kopfchunk gehört.
$e'_i = e_i \downarrow_{\{\_v \to t\}}$
$a'_i = a_i \downarrow_{\{\_v \to t\}}$

f = (is $\_v_1$ $\_v'_1$) ... (is $\_v_k$ $\_v_k'$), wobei $\_v_i \neq \_v'_j$ gilt und es existieren Regeln $\_v_i \to t$ und $\_v'_i \to t$ in $\mathcal{B}_C''$.

$\mathcal{B}_C' = \mathcal{B}_C'' - \{\_v_i \to t \mid \_v_i$ kommt in f vor$\}$[26]

DIP$_3$

$$\frac{(ct\ head \ldots (\text{is } \_v\ t) \ldots),\ \mathcal{B}_C}{(ct\ head \ldots (\text{is } \_v\ t) \ldots),\ \mathcal{B}_C'}$$

Falls $f_A(\_v) = \mathcal{TA}\text{-}\mathcal{FreeVar}$ und t ist eine Struktur und t kommt in keiner Regel aus $\mathcal{B}_C$ auf der rechten Seite vor.

Es gilt dann:
$\mathcal{B}_C' = \mathcal{B}_C + \{\_v \to t \downarrow_{\mathcal{B}_C}\}$

Die Regel DIP$_1$ propagiert *simple-bindings* einer Variable durch die Klausel. Die Bindung entsteht aufgrund des is-Literals (is $\_v$ t). Da die Variable $\_v$ zum erstenmal vorkommt, kann sie noch keine Bindung haben, ist also eine freie Variable. Dies gilt nicht nur zur Transformationszeit, sondern auch zur Laufzeit. Die Unifikation von $\_v$ und t, die vom is-Literal angestoßen wird, kann demnach nicht scheitern und bewirkt die Bindung $\_v \to t$ in der Laufzeitumgebung. Diese Bindung wird nun statisch durch die Klausel nach rechts propagiert ($e'_i = e_i \downarrow_{\{\_v \to t\}}$), womit das is-Literal aus der Klausel eliminiert werden kann. Man beachte, daß die eigentliche Elimination durch die Regeln aus $\mathcal{DBE}$ geschieht, denn die rechte Seite vom is-Literal wird an der Position des is-Literals ersetzt, welche durch Anwendung der Regel $\mathcal{DBE}$ wieder gelöscht wird, falls es sich nicht um die foot-Position handelt.

[26] Mit "-" bezeichnen wir die Wegnahme und mit "+" die Hinzunahme von Mengenelementen.

Hatte dagegen _v schon vorher ein Vorkommen in der Klausel (Regel DIP_2), kann eine Bindung der Variablen _v bestehen, die zur Transformationszeit nicht festzustellen ist. Zur Laufzeit kann also insbesondere die Ausführung des is-Literals scheitern, weshalb es in diesem Fall auch nicht eliminiert werden darf. Bild 2.3 zeigt das fehlerhafte Verhalten einer angenommenen Regel DIP'_2, in der wie in DIP_1 das is-Literal durch seine rechte Seite ersetzt wird. Nach der Anwendung der Regel DIP'_2 wird, ohne daß es im Beispiel aufgeführt ist, die Regel DBE_1 angewendet, die das generierte denotative Rumpfliteral *t* löscht. Der Übersichtlichkeit wegen, betrachten wir $\mathcal{B}_C$ nicht, da es hier noch keine Rolle spielt.

```
(ft (foo) (is _v a) (bar _v _w) (is _w b) `(tup _v _w) )          (1)
(hn (bar a c) )

⟶DIP1
(ft (foo) (bar a _w) (is _w b) `(tup a _w) )                     (2)
(hn (bar a c) )

⟶DIP'2
(ft (foo) (bar a _w) `(tup a b) )                                (3)
(hn (bar a c) )
```

Bild 2.3: Inkorrekte Regel DIP'_2. In (2) ist die Klausel noch semantisch äquivalent zu (1). In (3) ist die Semantik verändert.

In Programm (1) scheitert ein möglicher Aufruf (*foo*). Die Begründung dafür ist, daß das goal (*bar _v _w*) mit der Bindung _v →a zwar erfolgreich ist und die Variable _w an die Konstante *c* bindet, aber die Auswertung des folgenden is-Literals (is _w *b*) zu unknown führt. Das Programm (2) verhält sich analog und scheitert ebenfalls am is-Literal. Das Programm (3) führt dagegen nicht zu unknown, sondern liefert stattdessen die Liste (tup a b) als Wert. Das Verhalten ist also verschieden von Programm (1) und somit die Transformationsregel DIP'_2 nicht korrekt.

Die Regel DIP_3 behandelt Bindungen von Variablen an Strukturen. Diese Bindungen werden nicht explizit durch die Klausel propagiert, damit die Struktur nicht unnötigerweise mehrmals auf dem Heap aufgebaut wird (siehe auch Abschnitt 2.1.1). Vielmehr werden die Bindungen in dem Regelsystem $\mathcal{B}_C$ vermerkt. Das Regelsystem $\mathcal{B}_C$ ist mit Vorsicht zu behandeln, da die Definitionen der einzelnen Transformationsregeln den Eindruck vermitteln, daß $\mathcal{B}_C$ für die ganze Klausel gilt. Tatsächlich werden die Bindungen nicht über die gesamte Klausel propagiert, sondern nur nach rechts. Die Kontrollstrategie gewährleistet, daß eine Regel nie mit der "falschen" Bindungsumgebung $\mathcal{B}_C$ angewendet wird.

Die Bedingung $f_A(\_v)$ = *TA-FreeVar* (äquivalent zu $\_v\downarrow_{\mathcal{B}C} = \_v$) in den Regeln gewährleistet, daß die Variable $\_v$ noch keine Bindung hat. Der Fall, in welchem für $\_v$ bereits eine Bindung festgestellt ist, wird in den nachstehenden Regeln (Def. 2.8) erfaßt.

Die Anwendung der Regel DIP_2 zieht einige Aktionen nach sich, die mit "Es gilt dann:" eingeleitet und nun erläutert werden. Oben haben wir schon festgestellt, daß in DIP_2 die Variable $\_v$ schon frühere Vorkommen in der Klausel hat. Es ist also möglich, daß $\_v$ in einer rechten Seite einer Regel aus $\mathcal{B}_C$ vorkommt. Da mit der Anwendung von DIP_2 die statische Bindung von $\_v$ an eine Konstante oder Variable propagiert wird, muß man auch in $\mathcal{B}_C$ die neue Bindung bekannt machen. Hierdurch kann es geschehen, daß rechte Seiten von Regeln aus $\mathcal{B}_C$ syntaktisch gleich werden, also über eine einzige Variable geshared werden können. Betrachten wir folgendes Klauselfragment mit zugehöriger Bindungsumgebung $\mathcal{B}_C$.

... (is _v a) ...

$\mathcal{B}_C$ = { $\_v_1 \rightarrow$`(g a a), $\_v_2 \rightarrow$`(g _v a), $\_v_3 \rightarrow$`(g a _v)}

Wendet man nun die Regel DIP_2 an und aktualisiert die $\mathcal{B}C$ *mit* $_v \rightarrow$a, so ergibt sich:

$\mathcal{B}_C$"= {$\_v_1 \rightarrow$`(g a a), $\_v_2 \rightarrow$`(g a a), $\_v_3 \rightarrow$`(g a a)}

Alle drei Variablen in $\mathcal{B}_C$" sind nun an die Struktur `(g a a) gebunden. Im Aktionsteil von $DIP_2$ wird deswegen die Stringvariable f = (is $\_v_2$ $\_v_1$) (is $\_v_3$ $\_v_1$) erzeugt und als neue Bindungsumgebung erhalten wir $\mathcal{B}_C$' = {$\_v_1 \rightarrow$`(g a a)}. Die Stringvariable f wird in die Klausel fragmentiert und Anwendungen von Regeln auf die neuen is-Literale propagieren die neue Information durch die Klausel, wie das folgende Beispiel, in dem auch schon die später definierten System *CSE* und *SIU* angewendet werden, zeigt:

(ft (foo _v) **(is _v `(f b))** (bar _w) (is _w `(f _u)) (is _u b) (bar _u)), $\mathcal{B}_C = \emptyset$
$\longrightarrow_{DIP_3}$
(ft (foo _v) (is _v `(f b)) (bar _w) **(is _w `(f _u))** (is _u b) (bar _u))
$\mathcal{B}_C$= {_v →`(f b)}
$\longrightarrow_{DIP_3}$
(ft (foo _v) (is _v `(f b)) (bar _w) (is _w `(f _u)) **(is _u b)** (bar _u))
$\mathcal{B}_C$= {_v → `(f b), _w → `(f _u)}
$\longrightarrow_{DIP_2}$
(ft (foo _v) (is _v `(f b)) (bar _w) (is _w `(f _u)) **(is _w _v)** **(is _u b)** (bar b))
$\mathcal{B}_C$= {_v → `(f b)}
$\xrightarrow{*}_{DIP_{2,1}}$
(ft (foo _v) (is _v `(f b)) (bar _w) (is _v `(f _u)) (is _w _v) (bar b))

$\mathcal{B}_C$= {_v → `(f b)}
$\xrightarrow{*}_{CSE,\ SIU}$
(ft (foo _v) (is _v `(f b)) (bar _w) (is _w _v) (bar b))
$\mathcal{B}_C$= {_v → `(f b)}

2.2.2.3 Statische *is-Term*-Unifikation

Die Unifikation ist das zentrale Element bei der Abarbeitung von logischen und RELFUN-Programmen. Eine Effizienzsteigerung bei der Programmausführung läßt sich deswegen dadurch erreichen, daß man zur Transformationszeit möglichst viel von der allgemeinen Unifikation vorwegnimmt. Als das schon fast klassische Beispiel für diesen Ansatz steht die WAM. Die allgemeine Unifikation des Interpreters wird ersetzt durch spezielle *unify-, put-* und *get*-Instruktionen. In vielen Fällen braucht so nicht der allgemeine Unifikationsalgorithmus aufgerufen werden, sondern es werden Instruktionen für einfache Zuweisungen oder Vergleiche generiert. Die allgemeine Unifikation würde letztendlich mit wesentlich mehr Aufwand das gleiche Ergebnis liefern.

Die nun folgenden Regeln *SIU* (*static is-term unification*) gehen noch einen Schritt weiter und unifizieren zur Transformationszeit weitgehenst die beiden Seiten eines is-Literals miteinander. Komplexe Strukturen, in denen nur einzelne Komponenten verglichen werden, können so effizient verarbeitet werden. Hierzu betrachten wir das Bild 2.4. Die Intention des is-Literals ist offensichtlich, Wertzuweisungen an Variablen zu erzeugen, die in einem festen Kontext, hier durch die äußere Termstruktur gegeben, vorhanden sind.

(is (f (g _u) b) `(f (g (h _v) _w))) (a)

(is _u `(h _v)) (is _w b) (b)

Bild 2.4 Statische is-Term-Unifikation. Komplexe Strukturen (a) können in einfachere is-Literale transformiert werden (b), wenn die äußeren Strukturen übereinstimmen

Im Beispiel ist für jede mögliche dynamische Bindung der Variablen *_u*, *_v* und *_w* das is-Literal aus (a) erfolgreich, genau dann wenn die beiden is-Literale aus (b) erfolgreich sind. Während in (a) die Darstellungsform für einen Programmierer vorteilhaft sein kann, da sie den Kontext zeigt, in welchem die Variablen gebunden werden, ist die Darstellungsform (b) für einen WAM-Compiler vorteilhaft. Im Fall (a) würde dieser nämlich entweder unnötige WAM-Instruktionen für die äußere Termstruktur erzeugen, oder er erkennt auf dieser niedrigeren Ebene, daß diese Instruktionen redundant sind. In diesem Fall wäre die gleiche Optimierung nicht mehr so klar durchschaubar, da auf der WAM-Ebene der Zusammenhang einer Termstruktur verschwimmt und der Compiler dann nur anhand der Instruktionen erkennt, daß zur Laufzeit Berechnungen bzw. Operationen unnötigerweise ausgeführt werden.

Definition 2.8 *SIU (static is-term unification)*

SIU$_1$

(*ct head* ... (is t t') ... e), $\mathcal{B}_C$

(*ct head* ... (is $\_x_1$ t_1) ... (is $\_x_n$ t_n) ...e), $\mathcal{B}_C$

Falls t' ein denotatives Literal ist und $\mu = mgu^{27}(t\downarrow_{\mathcal{B}_C}, t'\downarrow_{\mathcal{B}_C})$ existiert.

Hierbei gilt für die neuen is-Literale: $\mu(\_x_i) = t_i \neq \_x_i$

SIU$_2$

(*ct head* ... (is t t') ...), $\mathcal{B}_C$

(*ct head* ... unknown ...), $\mathcal{B}_C$

Falls t' ein denotatives Literal ist und $mgu(t\downarrow_{\mathcal{B}_C}, t'\downarrow_{\mathcal{B}_C})$ **nicht** existiert.

SIU$_3$

(ft *head* ... (is t t')), $\mathcal{B}_C$

(ft *head* ... (is t t') `t), $\mathcal{B}_C$

Falls t' ein denotatives Literal ist.

Betrachten wir zuerst die Regel SIU$_3$. Sie hat offensichtlich nur die Aufgabe, den Wert der ft-Klausel zu erhalten, der durch den Wert des is-Literals gegeben ist. Ein kleines Beispiel (ohne $\mathcal{B}_C$) macht sofort die Notwendigkeit von SIU$_3$ deutlich. Hierbei sei noch angemerkt, daß die Anwendung von SIU$_3$ immer eine Anwendung von SIU$_1$ oder SIU$_2$ nach sich zieht.

(ft (foo a) **(is b b)**)
$\longrightarrow_{SIU3}$
(ft (foo a) **(is b b)** b)
$\longrightarrow_{SIU1}$
(ft (foo a) b)

Die Regel SIU$_1$ unifiziert die beiden Seiten des is-Literals zur Transformationszeit. Hierbei werden die bereits gefundenen komplexen Bindungen $\mathcal{B}_C$ der Variablen berücksichtigt. Existiert der mgu μ, so wird das is-Literale durch eine Reihe von neuen is-Literalen ersetzt. Sie beschreiben μ extensional, falls der mgu existiert. Existiert der mgu nicht (Regel SIU$_2$), wird dies frühzeitig erkannt und das is-Literal durch unknown ersetzt. Wir greifen nachstehend erneut das Beispiel aus Abschnitt 2.2.2.1 auf. Die Literale, die zur Transformation anstehen,

[27] mgu steht für *most general unifier*.

sind wiederum fettgedruckt. Die Bezeichnung "$\xrightarrow{*}_{\mathcal{SFL}}$" bedeutet, daß endlich viele Regeln aus dem System $\mathcal{SFL}$ angewendet werden. Die Klausel wird also letztendlich in die flatter-Normalform überführt.

(ft **(foo `(f b) `(f _x) _x)** (is _y `(f _x)) **(p _y _x (is a a))**)
$\mathcal{B}_C = \emptyset$

$\xrightarrow{*}_{\mathcal{SFL}}$
(ft (foo _1 _2 _x) **(is _2 `(f _x))** (is _1 `(f b)) (i s _y `(f b))
(is _y `(f _x)) (is a a) (p _y _x a))
$\mathcal{B}_C = \emptyset$

$\longrightarrow_{DIP3}$
(ft (foo _1 _2 _x) (is _2 `(f _x)) **(is _1 `(f b))** (is _y `(f b))
(is _y `(f _x)) (is a a) (p _y _x a))
$\mathcal{B}_C = \{\_2 \rightarrow (f\ \_x)\ \}$

$\longrightarrow_{DIP3}$
(ft (foo _1 _2 _x) (is _2 `(f _x)) (is _1 `(f b)) **(is _y `(f b))**
(is _y `(f _x)) (is a a) (p _y _x a))
$\mathcal{B}_C = \{\_2 \rightarrow `(f\ \_x), \_1 \rightarrow `(f\ b)\ \}$

$\longrightarrow_{DIP3}$
(ft (foo _1 _2 _x) (is _2 `(f _x)) (is _1 `(f b)) (is _y `(f b))
(is _y `(f _x)) (is a a) (p _y _x a))
$\mathcal{B}_C = \{\_2 \rightarrow `(f\ \_x), \_1 \rightarrow `(f\ b), \_y \rightarrow `(f\ b)\ \}$

$\longrightarrow_{SIU1}$
(ft (foo _1 _2 _x) (is _2 `(f _x)) (is _1 `(f b)) (is _y `(f b))
(is _x b) (is a a) (p _y _x a))
$\mathcal{B}_C = \{\_2 \rightarrow `(f\ b), \_1 \rightarrow `(f\ b), \_y \rightarrow `(f\ b)\ \}$

$\longrightarrow_{DIP2}$
(ft (foo _1 _2 b) (is _2 `(f b)) (is _1 `(f b)) (is _y `(f b))
(is a a) (p _y b a))
$\mathcal{B}_C = \{\_2 \rightarrow `(f\ b), \_1 \rightarrow `(f\ b), \_y \rightarrow `(f\ b)\ \}$

$\longrightarrow_{SIU_1}$

(ft (foo _1 _2 b) (is _2 `(f b))  (is _1 `(f b)) (is _y `(f b))
                   (p _y b a))

$\mathcal{B}_C$ = { _2 → `(f b), _1 → `(f b), _y → `(f b) }

Bild 2.5 Anwenden der Regeln *SFL*, *DIP* und *SIU*.

Es werden immer die Normalformen $t\downarrow_{\mathcal{B}_C}$ und $t'\downarrow_{\mathcal{B}_C}$ zu den Termen t und t' bei der Bildung des mgu betrachtet. Die Variablen $\_x_i$ kommen in $t\downarrow_{\mathcal{B}_C}$ oder $t'\downarrow_{\mathcal{B}_C}$ vor und sind deswegen ebenfalls irreduzibel bezüglich $\mathcal{B}_C$. Es gilt demnach $\_x_i\downarrow_{\mathcal{B}_C} = \_x_i$ und deswegen bringt eine erneute Anwendung von *SIU* auf die generierten is-Literale nichts Neues. Die Kontrollstrategie trägt dem Rechnung, indem nicht versucht wird, auf die generierten is-Literale erneut die Regeln *SIU* anzuwenden.

Die Korrektheit der Regel SIU_1 ist gegeben, da der Unifikationsalgorithmus aus dem Interpreter zur Bestimmung des mgu μ benutzt wird. Zur Laufzeit würde der Interpreter genau die dynamischen Variablenbindungen erzeugen, die durch die Regel explizit gemacht worden sind. Für jede Variableninstantiierung, die das Literal (is t t') scheitern läßt, existiert entsprechend ein $\_x_i$, so daß auch das Literal (is $\_x_i$ $\_t_i$) scheitert. Die Regel SIU_2 dagegen erweitert den von RELFUN verwendeten Unifikationsalgorithmus um den *occur-check*, was die Semantik von RELFUN in einem "positiven Sinne" ändert, da RELFUN (genau wie PROLOG) die Unifikation ohne occur-check realisiert. Man macht dies wegen der Performanz, denn der occur-check kostet viel Zeit. Da für die optimierende Compilation die Übersetzungszeit (anders als die Ausführungszeit) nur eine untergeordnete Rolle spielt, ist der occur-check von diesem Gesichtspunkt aus in der Regel SIU_2 unbedenklich. Kommt sie zur Anwendung wird aber eine Warnung an den Programmierer ausgegeben, der im allgemeinen nicht gewollt haben wird, daß seine Klausel an dieser Stelle unknown liefert. Er hat dann die Möglichkeit, die Klausel zu editieren. Wenn wir uns Erinnern, daß die Transformation in den RELFUN-Kern im RFM-System (Abschnitt 1 und [Bol91]) über das RELFUN-Kommando *horizon* angestoßen wird, könnte eine RELFUN-Sitzung folgenden Charakter haben:

```
rfi> az (ft (test _x) (is _x `(f _x)))
rfi> l
(ft (test _x)
   (is _x `(f _x)) )
rfi> horizon

***** WARNING ****
Occur-check has failed
```

```
In procedure: (test 1)
In clause:
(ft (test _x) (is _x `(f _x)))
rfi> l
(ft (test _x)
   unknown )
```

2.2.2.4 Sharen von Ausdrücken und Strukturen

Ein wichtiges Kennzeichen eines optimierten Programms ist, daß möglichst wenig überflüssige Berechnungen ausgeführt werden. Wir wollen Berechnungen betrachten, die mehrmals ausgeführt werden oder unnötigen Speicherplatz beanspruchen. Hierfür wird das Ergebnis einer Berechnung *f* in eine Variable zwischengespeichert und bei einer Wiederholung von *f* direkt durch die Variable ersetzt. Wir müssen allerdings voraussetzen, daß die Berechnung *f* streng funktional ist, also weder (a) einen Wert durch Bindung einer Anfragevariable noch (b) mehrere "echte" Rückgabewerte liefert. Unter der Annahme, daß wir ein builtin-Prädikat *is-var/1* zur Verfügung haben, das für (is-var _v) testet, ob _v eine ungebundene Variable ist, erklärt sich die Bedingung (a) durch folgendes Beispiel:

```
(ft (val-if-var _v) (is-var _v) (is _v a) )
(ft (no-success _u) (is _v (val-if-var _u))  (is _w (val-if-var _u))  (tup _u _v _w) )
```

Die RELFUN-Funktion tup/3 liefere hierbei ihre Argumente in eine tup-Struktur (die Listenrepräsentation von RELFUN) zurück. Es ist leicht zu sehen, daß der Aufruf (no-success _u) scheitert, denn der erste Aufruf von (val-if-var _u) gelingt mit der Bindung _u = a und der zweite Aufruf (val-if-var _u) scheitert, da in diesem Fall das Prädikat (is-var a) nicht bewiesen werden kann. Die RELFUN-Funktion val-if-var/1 ist aber deterministisch und seiteneffektfrei, und wir können somit die Regel CSE_1 (s. u.) anwenden, falls wir Bedingung (a) nicht beachten. Wenden wir zusätzlich noch *DIP* an, erhalten wir:

```
(ft (val-if-var _v) (is-var _v) (is _v a) )
(ft (no-success _u) (is _v (val-if-var _u))  (tup _u _v _v) )
```

Nun liefert der Aufruf (no-success _u) den Wert (tup a a a) mit der Bindung _u = a und wir haben ein anderes Verhalten des Programms. Ähnlich verhält es sich, wenn wir die Bedingung (b) weglassen und somit Funktionen mit mehreren Rückgabewerten zulassen:

(ft (ab) a)
(ft (ab) b)
(ft (fourpairs) (tup (ab) (ab)))

Die RELFUN-Funktion fourpairs/0 liefert mit Nachfordern durch *more* nacheinander die Werte: (tup a a), (tup a b), (tup b a) und (tup b b). Nach dem Anwenden von $\mathcal{SFL}$ auf die Definition von fourpairs ergibt sich die Klausel:

(ft (fourpairs) (is _1 (ab)) (is _2 (ab)) (tup _1 _2))

Wendet man nun auch die Regel CSE_1 und DIP_1 an, erhält man das Programm:

(ft (ab) a)
(ft (ab) b)
(ft (fourpairs) (is _1 (ab)) (tup _1 _1))

Der Aufruf von (fourpairs) liefert jetzt aber nur die zwei Werte (tup a a) (tup b b).

Die Voraussetzungen (a) und (b) können z. B. für grund-instantiierte und deterministische RELFUN-Prozeduren garantiert werden. Desweiteren dürfen die Berechnungen keine Seiteneffekte wie beispielsweise das Ausdrucken von diversen Meldungen besitzen, da beim Substituieren der erneuten Berechnung durch das vorher gespeicherte Ergebnis keine Wiederholung der Meldung geschehen würde.

Wir zählen in dieser Stelle denotative Literalen zu den *streng funktionalen* Berechnungen. Hierdurch wird erreicht, daß Strukturen, die in der Klausel mehrmals vorkommen, über eine Variable *geshared* und somit nur einmal auf dem WAM-Heap aufgebaut werden.

Definition 2.9 *CSE (common subexpression extraction)*

CSE_1

(*ct head* ... (is _v *e*) ... (is _*w e*) ...), $\mathcal{B}_C$ — Falls *e* deterministisch und seiteneffektfrei ist.

(*ct head* ...(is _v *e*) ... (is _w _v)...), $\mathcal{B}_C$

CSE_2

(*ct head* ... (is _v *t*)...), $\mathcal{B}_C$

(*ct head* ...(is _v _w) ...), $\mathcal{B}_C$

Falls eine Regel _w → *t* in $\mathcal{B}_C$ existiert und _v ≠ _w gilt

Die Regel CSE_1 behandelt echte Berechnungen, während CSE_2 das structure sharing realisiert. Für die Anwendung der Regel CSE_1 bestehen starke Bedingungen, die im allgemeinen für ein evaluatives RELFUN-Literal nicht entscheidbar sind. In der Praxis beschränken wir uns zur Zeit auf die einfachen Fälle, in denen das Literal *e* einen seiteneffektfreien LISP-Durchgriff darstellt (oder ein denotatives Literale ist, obwohl diese Fälle meist CSE_2 zukommen). Sie sind zwar evaluativ aber deterministisch und erfüllen somit offensichtlich die Bedingungen für die Anwendung von CSE_1. In Abschnitt 3 dieser Arbeit wird skizziert, wie mit Hilfe der globalen Datenflußinformation der Bedingungsteil der Regeln auch für RELFUN-Prozeduren teilweise testbar ist. An dieser Stelle sei schon ein Vorgriff auf das Regelsystem $\mathcal{PVE}$ (Def. 2.11) gestattet. Es behandelt ebenfalls evaluative Literale mit dem Unterschied, daß hier zur Transformationszeit das Literal auswertbar sein muß.

Als Anwendungsbeispiel betrachten wir nochmals die Klausel aus Bild 2.5. Sie wurde bereits durch Anwendung von $\mathcal{SFL}$, $\mathcal{DIP}$ und $\mathcal{SIU}$ transformiert und hatte schließlich die Darstellung wie in Bild 2.6 und wird nun mit Hilfe der zusätzlichen Regeln $\mathcal{CSE}$ weiter umgeformt.

(ft (foo _1 _2 b) **(is _2 `(f b))** (is _1 `(f b)) (is _y `(f b)) (p _y b a))
$\mathcal{B}_C = \emptyset$

$\longrightarrow_{DIP3}$
(ft (foo _1 _2 b) (is _2 `(f b)) **(is _1 `(f b))** (is _y `(f b)) (p _y b a))
$\mathcal{B}_C$ = {_2 → `(f b)}

$\longrightarrow_{CSE2}$
(ft (foo _1 _2 b) (is _2 `(f b)) **(is _1 _2)** (is _y `(f b)) (p _y b a))
$\mathcal{B}_C$ = {_2 → `(f b)}

$\longrightarrow_{DIP1}$
(ft (foo _2 _2 b) (is _2 `(f b)) **(is _y `(f b))** (p _y b a))
$\mathcal{B}_C$ = {_2 → `(f b)}

$\longrightarrow_{CSE2}$
(ft (foo _2 _2 b) (is _2 `(f b)) **(is _y _2)** (p _y b a))

$\mathcal{B}_C$ = {_2 → `(f b)}

$\longrightarrow_{DIP_1}$
(ft (foo _1 _2 b) (is _2 `(f b)) (p _2 b a))
$\mathcal{B}_C$ = {_2 → `(f b)}

Bild 2.6 Fortführung des Beispiels aus Bild 2.5: Anwenden der Systeme *DIP* und *CSE*

2.2.2.5 Propagieren von *unknown*

Die Transformationsregeln *SIU* erzeugen für den Fall, daß die statische Unifikation scheitert, ein explizites *unknown* als Literal in der Klausel. Zur Laufzeit wird sie an dieser Stelle scheitern und auf keinen Fall die goals rechts von *unknown* aufrufen können. Es ist demnach sinnvoll und semantisch korrekt, diese Literale aus der Klausel zu eliminieren. Es stellt sich die Frage, ob auch alle Literale links von *unknown* gleichermaßen behandelt werden können. Dies ist zumindest dann zu verneinen, wenn das Literal einen Seiteneffekt bewirkt. Würde man ein solches Literal löschen, würde der Seiteneffekt nicht ausgeführt werden und das Verhalten der Klausel nicht der ursprünglichen entsprechen. Auch ein allgemeiner Cut-Operator wäre hierbei ein Beispiel eines Literals mit Seiteneffekt. Ähnlich verhält es sich, wenn ein Literal gelöscht wird, das zu einer nicht terminierenden Berechnung führen würde. Obwohl dieser Fall akzeptiert werden könnte, da der Programmierer die Endlosschleife kaum gewollt haben wird, verbieten wir in den folgenden Regeln alle "Linkslöschungen" dieser Art.

Man kann man davon ausgehen, daß ein explizites *unknown* in einer Klausel vom Programmierer selten beabsichtigt gewesen ist[28]. In der Implementierung erhält der Programmierer deswegen eine Warnung, daß ein *unknown* generiert bzw. entdeckt worden ist. Er hat es dann selbst in der Hand, sein Programm zu überarbeiten oder das unknown stehen zu lassen.

Definition 2.10 *UNP (unknown propagation)*

UNP$_1$

(*ct head ... unknown ...*), $\mathcal{B}_C$

(*ct head ... unknown*), $\mathcal{B}_C$

[28]Eine Ausnahme ist die übliche Definition von *negation as failure* in PROLOG, die in RELFUN entsprechend definiert werden kann.

2.2.2.6 Vorauswertung von Prämissen

Für einige Prämissen von Klauseln kann schon zur Transformationszeit der Wert eindeutig bestimmt werden. Ersetzt man diese Prämissen durch ihren Wert, entfällt die Berechnung zur Laufzeit und kann somit die Performanz des Programms erheblich steigern. In Fällen mit sehr aufwendigen Berechnungen kann der Laufzeitgewinn beträchtlich sein. Das Regelsystem $\mathcal{PEV}$ beschreibt diese Vorauswertung eines Literals in der Klausel. Wie in ähnlichen Fällen der vorherigen Regelsysteme, beschränken wir uns in der Praxis zunächst nur auf die LISP-Durchgriffe.

Definition 2.11 $\mathcal{PEV}$ *(pre-evaluation)*

PEV$_1$

(*ct head* ... (is _v *e*)...), $\mathcal{B}_C$

(*ct head* ... (is _v *t*)...), $\mathcal{B}_C$

Falls $e\downarrow_{\mathcal{I}C}$ deterministisch, seiteneffektfrei und grund-instantiiert ist.

Es gilt hierbei:
t = PREVAL[29]$(e\downarrow_{\mathcal{I}C})$

PEV$_2$

(*ct head* ... *e*...), $\mathcal{B}_C$

(*ct head* ... *t*...), $\mathcal{B}_C$

Falls $e\downarrow_{\mathcal{I}C}$ deterministisch, seiteneffektfrei und grund-instantiiert ist.

Es gilt hierbei:
t = PREVAL$(e\downarrow_{\mathcal{I}C})$

Damit die Regel PEV$_1$ (oder PEV$_2$) angewendet werden darf, ist es ist nicht notwendig, daß *e* grund-instantiiert ist. Es können also durchaus Variablen in *e* vorkommen. Diese müssen aber eine statische Bindung an einen Term haben, der keine Variablen enthält. Die Regel erlaubt einem Programmierer beispielsweise, feste arithmetische Werte durch eine Berechnungsformel auszudrücken, wenn es ihm sinnvoll erscheint, ohne dadurch das Programm ineffizienter zu machen.

Sei PREVAL korrekt und terminierend für alle erlaubten Eingaben mit einem denotativen Literal. Die Korrektheit von $\mathcal{PEV}$ ergibt sich dann sofort anhand der starken Vorbedingungen zur Anwendung der Regeln. Außerdem wird die Anzahl der evaluativen Literale nach jeder

[29]PREVAL sei ein Auswertungsmechanismus für deterministische Funktionen wie etwa LISP's EVAL.

Anwendung einer Regel um eins reduziert. Hieraus ergibt sich, daß diese Regeln nur endlich oft angewendet werden können.

Ein abschließendes Beispiel illustriert die Anwendung von $\mathcal{PEV}$.

(ft (foo _u _v) (is _u 20) (is _v 50) **(is _erg (+ (/ _u 10) (/ _v 10)))** _erg)
$\mathcal{B}_C = \emptyset$

$\xrightarrow{*}_{\mathcal{SFL}}$
(ft (foo _u _v) **(is _u 20) (is _v 50)** (is _1 (/ _u 10)) (is _2 (/ _v 10)
(is _erg (+ _1 _2)) _erg)
$\mathcal{B}_C = \emptyset$

$\xrightarrow{*}_{\mathcal{DIP}}$
(ft (foo 20 50) **(is _1 (/ 20 10)) (is _2 (/ 50 10)** (is _erg (+ _1 _2)) _erg)
$\mathcal{B}_C = \emptyset$

$\xrightarrow{*}_{\mathcal{PEV}}$
(ft (foo 20 50) **(is _1 2) (is _2 5)** (is _erg (+ _1 _2)) _erg)
$\mathcal{B}_C = \emptyset$

$\xrightarrow{*}_{\mathcal{DIP}}$
(ft (foo 20 50) **(is _erg (+ 2 5))** _erg)
$\mathcal{B}_C = \emptyset$

$\longrightarrow_{PEV_1}$
(ft (foo 20 50) **(is _erg 7)** _erg)
$\mathcal{B}_C = \emptyset$

$\longrightarrow_{DIP_1}$ (ft (foo 20 50) 7)

Bild 2.6 Anwenden der Regeln $\mathcal{PEV}$

2.2.2.7 Umsortieren der denotativen is-Literale

Dieser Abschnitt stellt kein neues Regelsystem für eine weitere Normalisierungstransformation vor, sondern möchte jedeglich die Anordnung der denotativen is-Literale in den chunks und im speziellen für den Kopfchunk diskutieren. Wir werden feststellen, daß es hier große Unterschiede in der Effizienz zwischen den verschiedenen Anordnungen gibt, während die Semantik natürlich bei allen äquivalent ist. Die Diskussion wird an einem kleinen Beispiel fest-

gemacht, das die wichtigsten Ideen aufzeigt. Betrachten wir hierzu folgendes Klauselfragment einer mit den oben vorgestellten Regelsystemen normalisierten Klausel:

(ft (foo _1) (is _2 `(g b)) (is _1 `(f _2)) ...) (a)

Wir wollen nun drei verschiede Arten eines Aufrufs foo/1 betrachten. Der Aufruf kann zum einen mit einer ungebundenen Variablen erfolgen etwa (foo _v). Mit diesem Aufruf gelingt sicherlich der Match mit der Klausel (a). Die folgenden zwei is-Literale bauen dann nacheinander die Struktur `(f (g b)) auf und binden sie letztendlich an die Variable _v. Betrachten wir nun ein Klauselfragment einer semantisch äquivalenten Klausel, in der nur die beiden is-Literale vertauscht sind.

(ft (foo _1) (is _1 `(f _2)) (is _2 `(g b)) ...) (b)

Das Verhalten für den Aufruf (foo _v) mit der ungebundenen Variablen _v ist in (b) identisch mit dem obigen. Bis hierher scheint die Anordnung der is-Literale demnach gar nicht so wichtig zu sein, betrachten wir jetzt aber den Aufruf (foo _v) mit einer gebundenen Variablen _v etwa mit der Bindung an die Struktur `(h c). In (a) wird aufgrund des ersten Literals die Variable _2 an die Struktur `(g b) gebunden, bis dann die Klausel aufgrund des zweiten is-Literals scheitert, denn hier wird erkannt, daß das Toplevel Funktionssymbol von _v (nämlich h) mit f nicht unifizierbar ist. Der selbe Aufruf auf Klausel (b) angewandt, stellt dies schon im ersten is-Literal fest. Projiziert man diese Überlegung auf die RFM-Ebene (siehe Bild 2.9), wird in der RFM-Realisierung für die Klausel (a) zuerst einmal blind die Struktur `(g b) auf dem Heap mittels *put_structure-* und *unify_constant* -Instruktionen aufgebaut, was im betrachteten Fall völlig unnötig gewesen ist und nur Speicherplatz sowie teure RFM-Instruktionen kostet. Der RFM-Code für Klausel (b) ist da schon wesentlicher geschickter, denn durch *get_structure f, X_1* wird zuerst geprüft, ob das Eingabeargument an eine Struktur mit dem Funktor f gebunden ist. Für unseren Beispielaufruf wird hier also direkt ein *failure* erzeugt und insbesondere keine Strukturen auf dem Heap aufgebaut. Wie ungeschickt die Anordnung der is-Literale in (a) ist, wird besonders an einem Aufruf deutlich, bei welchem _v schon an die Struktur `(f (g b)) gebunden ist. Dies bedeutet nämlich, daß wir die Struktur schon auf dem Heap repräsentiert vorfinden.

```
foo/1 ; RFM-Code für (a)
        ...
        put_structure g, X2
        unify-constant b
        get_structure f, X1
        unify_value X2
        ...

foo/1   ; RFM-Code für (b)
        ...
        get_structure f, X1
        unify_variable X2
        get_structure g, X2
        unify-constant b
        ...
```

Bild 2.9 Die RFM-Realisierung der Klauseln (a) und (b)

Der Code für (a) erzeugt wiederum zuerst die Struktur `(g b) auf dem Heap (unnötigerweise!) und testet erst danach mittels *get_structure f, $X_1$*, ob die Toplevelfunktionssymbole identisch sind. Danach wird die Bindung des X-Registers X2 mit der Struktur `(g b) unifiziert (*unify_value X_2*). Man beachte, daß hier nun eine allgemeine Unifikation der übergebenen mit der aufgebauten Struktur `(g b) angestoßen wird. Dies ist sehr zeitaufwendig, besonders wenn man sich das Beispiel um einige is-Literale erweitert vorstellt. Der Code für (b) hingegen verhält sich wesentlich besser. Zum einen geht der RFM-Emulator nach der ersten *get_structure*-Instruktion in den *read-mode*[30] und baut deswegen keine unnötigen Strukturen auf dem Heap auf und zum anderen werden immer nur einfache Tests durch die Instruktionen repräsentiert. D. h. es werden nur Funktoren und Konstanten (zumindest beim vorausgesetzten Aufruf) verglichen und nicht der allgemeine Unifikationsalgorithmus angestoßen.

Das Dilemma bei der Anordnung der is-Literale in (a) besteht offensichtlich darin, daß auf der linken Seite vom is-Literal Variablen stehen, die an dieser Stelle zum erstenmal in der Klausel und insbesondere im chunk vorkommen und somit immer den Charakter einer *put_structure*-Instruktion haben, nämlich auf jeden Fall Strukturen aufbauen. Stehen auf der linken Seite Variablen, die schon einmal im selben chunk vorgekommen sind, haben sie automatisch einen *get-structure*-Charakter, sie bauen also nicht nur Strukturen auf, sondern vergleichen diese teilweise nur, wenn sie schon auf dem Heap vorhanden sind. Als eine optimale Anordnung der is-Literale wollen wir deswegen solche Anordnungen nennen, in denen auf der linken Seite der is-Literale in einem chunk möglichst wenige Variablen zum erstenmal im chunk vorkommen. Natürlich ist somit die optimale Anordnung nicht eindeutig definiert, und wir haben es vielmehr mit einer ganzen Klasse von optimalen Anordnungen zu tun, aber im obigen Beispiel kann die Anordnung in Klausel (a) auf jeden Fall als nicht optimal klassifiziert werden, da wir mit (b) eine semantisch äquivalente Klausel haben, die "optimaler" ist.

---

[30] In der WAM-Technologie werden ein und dieselben get_structure- und unify-Instruktionen zum Aufbau und zum Testen von Strukturen auf dem Heap benutzt. Hierzu kennt der Emulator (bzw die Maschine) den *read* und *write mode*. Beispielsweise schaltet die Instruktion get_structure f, X_i in den write mode, wenn X_i eine ungebunden Variable repräsentiert. Die unify-Instruktionen bauen im write mode mit Hilfe des impliziten *structure pointer* Strukturen in dem Heap auf, bzw unifizieren im read mode die Struktur im Heap mit der Bindung des X-Registers, welches sie als einziges Argument haben.

In der LISP-Implementierung des *Normalisierers* (siehe Abschnitt 2.3) wird deswegen eine Umsortierung der is-Literale innerhalb eines chunks vorgenommen, so daß sie nach rechts geschoben werden, wenn auf ihrer linken Seite eine Variable steht, die im chunk noch nicht vorgekommen ist.

2.3 Kontrollalgorithmus für die Anwendung der Regelsysteme

Bevor der Kontrollalgorithmus vorgestellt wird, fassen wir die in Abschnitt 2.2 definierten Regelsysteme in einer geordneten Menge NSET zusammen:

$$\text{NSET} := \{\mathcal{SFL}, \mathcal{DBE}, \mathcal{DIP}, \mathcal{SIU}, \mathcal{CSE}, \mathcal{UNP}, \mathcal{PEV}\}$$

Die totale Ordnung auf NSET sei durch die obige Reihenfolge der Regelsysteme von links nach rechts gegeben. Mit dieser Festlegung kann der Kontrollalgorithmus wie folgt angegeben werden.

Algorithmus: **NORMALIZE**

Eingabe: C ; Die zu normalisierende Klausel.
Ausgabe: C_K ; Die Kern-Klausel.

Variablen: C_N ; Repräsentiert den aktuellen Stand der Normalisierung.
B_C ; Die aktuelle komplexe Bindungsumgebung.
cnk ; Die Nummer des aktuellen chunks (Kopfchunk = 1).
X ; Das benutzte Regelsystem (X∈NSET).
L ; Das Literal, welches die Regelanwendung triggert.

1 Initialisiere $C_N := C{\downarrow_{\mathcal{SFL}}}$; $B_C := \emptyset$; cnk := 1; Gehe nach 2.

2 Falls cnk ≤ maximale chunk-Nummer in C_N, dann gehe nach 2.1, sonst gehe nach 3.

2.1 Falls keine Regel aus NSET im chunk mit Nummer cnk mehr anwendbar ist, dann gehe nach 2.2, sonst gehe nach 3.1.

2.1.1 C'_N, B'_C entstehe aus C_N, B_C durch Anwendung einer Regel aus X∈NSET getriggert durch das Literal L aus dem chunk mit der Nummer cnk. Wähle hierbei X minimal. Gehe nach 2.1.2

2.1.2 Setze $C_N := C'_N$ und $B_C := B'_C$. Gehe nach 3

2.1.3 Falls X = $\mathcal{UNP}$ gehe nach 3, sonst gehe nach 2.1.

2.2 Setze cnk := cnk +1; Gehe nach 2.

3 Setze $C_K := C_N$ und stoppe.

Der Algorithmus hat eine relativ einfache Struktur, da die eigentlichen Transformationsschritte in den Regelsystemen enthalten sind. Seine primäre Aufgabe besteht darin, die chunkweise Abarbeitung der Literale zu gewährleisten und somit die korrekte Verwendung der Bindungsumgebung zu garantieren. Genau wie in den Regeln DIP_1 und DIP_2 einfache Bindungen nur nach rechts und innerhalb des aktuellen chunks propagiert werden, dürfen wir eine in $\mathcal{BC}$ erzeugte komplexe Bindung nicht in einem vorherigen (weiter links stehenden) chunk verwenden. Man macht sich das "Verbot der Linksersetzung"[31] von komplexen Bindungen über chunk-Grenzen hinaus schnell an einem kleinen Beispiel klar:

```
(ft (foo) (is _u `(f _v)) (is _u _w) (bar _v) (is _w `(f b) )
(hn (bar a ) (rf-print "message"))
```

Man beachte, daß die Klausel foo/2 aus 2 chunks besteht, wobei der erste chunk mit dem Literal (bar _v) abschließt und der zweite chunk nur aus dem Literal (is _w `(f b)) besteht. Eine Analyse des Aufrufs (foo) zeigt, daß im ersten chunk die Variable _w an die Struktur `(f _v) gebunden wird. Beim Aufruf von (bar _v) ist _v eine frei Variable und wird demnach an die Konstante a gebunden. Weiterhin erfolgt die Meldung "message" auf den Ausgabestream. Die Bindung der Variablen _v hat zur Folge, daß _w nun an die Struktur `(f a) gebunden ist und das Literal (is _w `(f b)) scheitert. Wir halten also fest, daß das ursprüngliche Programm folgende Semantik hat: Der Aufruf von (foo) gibt per Seiteneffekt eine Meldung aus und scheitert.

Geben wir die strikte links-rechts Abarbeitung der chunks auf, können wir mit unseren Regelsystemen folgende Ableitung erzielen:

(ft (foo) (is _u `(f _v)) (is _u _w) (bar _v) **(is _w `(f b)**)
$\mathcal{B}_C = \emptyset$

$\longrightarrow DIP_3$
(ft (foo) (is _u `(f _v)) **(is _u _w)** (bar _v) (is _w `(f b))
$\mathcal{B}_C = \{\_w \rightarrow `(f\ b)\}$

$\xrightarrow{*} DIP_2, DIP_1, DBE$
(ft (foo) **(is _w `(f _v)** (bar _v) (is _w `(f b))
$\mathcal{B}_C = \{\_w \rightarrow `(f\ b)\}$

$\longrightarrow CSE_2$
(ft (foo) **(is _v b)** (bar _v) (is _w `(f b))
$\mathcal{B}_C = \{\_w \rightarrow `(f\ b)\}$

[31] Realisiert durch die chunkweise links-rechts Abarbeitung der Klausel.

$\xrightarrow{*}$ DIP$_1$, $\mathcal{DBE}$
(ft (foo) (bar b) (is _w `(f b))
$\mathcal{B}_C$ = {_w → `(f b)}

Bild 2.8 Inkorrektes Verhalten, falls die chunks nicht strikt von links nach rechts abgearbeitet werden

Offensichtlich haben wir für den Aufruf (foo) eine andere Semantik, denn der Aufruf scheitert jetzt schon beim Versuch das goal (bar b) zu beweisen, mit dem Effekt, daß die vorherige Meldung nicht mehr erfolgt.

Im folgenden wird der Algorithmus NORMALIZE etwas näher betrachtet. In Anweisung 1.1 wird die Eingabe-Klausel auf flatter-Normalform gebracht. Daß diese Transformation korrekt ist und terminiert, wurde bereits in Abschnitt 2.2.1 besprochen. Außerdem wird die komplexe Bindungsumgebung B_C auf die leere Menge initialisiert, da für eine RELFUN-Klausel keine initialen Bindungen existieren. Im weiteren werden in B_C die Bindungen von Variablen in der Klausel an Strukturen vermerkt. Wie dies im einzelnen geschieht, ist in den Regelsystemen aus NSET definiert. Die Einführung der chunk-Nummer cnk garantiert die chunkweise Abarbeitung der Literale in der Klausel. Man beachte, daß zwar die chunks sequentiell abgearbeitet werden, dies aber im allgemeinen nicht für die Literale innerhalb eines chunks gilt, denn hier ist die Reihenfolge durch die Triggereigenschaft und durch die Ordnung (auf NSET) der Regelsysteme festgelegt. Die Initialisierung von cnk erfolgt auf den Kopfchunk der Klausel (cnk := 1). In Anweisung 1.2 wird dann in eine (while-artige) Schleife über die aktuelle chunk-Nummer eingetreten, wobei die chunk-Nummer in 2.2 um jeweils 1 erhöht wird. Da eine RELFUN-Klausel nur aus endlich vielen Literalen besteht, hat sie auch nur endlich viele chunks. Wir können also festhalten, daß die Schleife terminiert, falls garantiert werden kann, daß nach endlich vielen Schritten im Algorithmus die Anweisung 1.3 oder 2.2 erreicht wird. Die Begründung dafür ist einfach, da a.) die Anweisung 1.3 den Algorithmus aufgrund der stop-Instruktion direkt terminieren läßt und b.) die Anweisung 2.2 die Programmvariable cnk inkrementiert, bis die maximale chunk-Nummer erreicht ist und somit nur endlich oft ausgeführt werden kann. Eine Termination der Schleife führt wiederum zur Ausführung der Anweisung 1.3; also terminiert der Algorithmus.

Es bleibt noch zu klären, ob die Anweisung 2.2 immer nach endlich vielen Schritte erreicht wird (falls nicht schon vorher Anweisung 1.3 erreicht wird). Das Problem reduziert sich auf die Frage, ob die Länge der Regelableitung innerhalb des aktuellen chunks begrenzt ist. Im Algorithmus findet man dieses Problem anhand der inneren Schleife (Anweisung 2.1) wieder, die über die Anwendbarkeit von Regeln in einem chunk definiert ist. Die Termination der einzelnen Regelsysteme aus NSET haben wir schon bei der jeweiligen Definition skizziert. Hier muß nun untersucht werden, ob paarweise Anwendung von unterschiedlichen Regelsystemen

eine Schleife bedingen können. Hierbei brauchen die Systeme $\mathcal{SFL}$, $\mathcal{DBE}$, $\mathcal{UNP}$ und $\mathcal{PEV}$ nicht weiter untersucht werden, da das System $\mathcal{SFL}$ nur in der Anweisung 1.1 benutzt wird, die Anwendung von $\mathcal{DBE}$ Literale eliminiert, die Anwendung von $\mathcal{UNP}$ direkt zur Termination führt und die Anwendung von $\mathcal{PEV}$ die Anzahl der evaluativen Literale in der Klausel reduziert und es keine Regel gibt, die ein evaluatives Literal erzeugt.

Bleiben noch die Systeme $\mathcal{DIP}$, $\mathcal{SIU}$, $\mathcal{CSE}$ zu untersuchen. Alle drei haben aber gemeinsam, daß sie entweder Variablen durch Konstanten oder andere (aber nicht neue) Variablen aus der Klausel ersetzen (siehe $\mathcal{DIP}$) oder vorhandene Strukturen durch (nicht neue) Variablen ersetzen (siehe $\mathcal{CSE}$) bzw. miteinander unifizieren und somit verkleinern (siehe $\mathcal{SIU}$). Da die Anzahl der Variablen in der Klausel endlich ist und eine maximale Komplexität der statischen Strukturen in der Klausel existiert und die Regeln entweder die Anzahl der relevanten Variablen[32] oder die Komplexität der Strukturen verkleinern, können sie nur endlich oft angewendet werden. Weiterhin garantieren die starken Vorbedingungen (z. B. für DIP_1), daß eine Regel nicht mehrmals durch dasselbe Literal getriggert wird und keinen weiteren Effekt hat, da beispielsweise bereits die Bindung propagiert wurde und eine erneute Anwendung die Bindung nochmals (aber effektlos) propagiert.

Die Korrektheit des Algorithmus folgt aus der Korrektheit der einzelnen Regelsysteme und der Tatsache, daß die chunkweise Abarbeitung erzwungen wird und somit jeweils die richtige komplexe Bindungsumgebung benutzt wird.

2.4 Implementierungshinweise zum Normalisierer

Dieser Abschnitt beschreibt die grundlegenden Ideen der Implementierung des Normalisierers, den wir ab jetzt mit *normalizer* bezeichnen möchten. Beim Programmdesign wurde großen Wert auf einen rein funktionalen Programmierstil gelegt. Der so entstandene normalizer ist deswegen in puren Lisp geschrieben und benutzt unter anderem keine globalen Variablen, die an einigen Stellen zwar Hilfreich gewesen wären, aber den angestrebten funktionalen Charakter widersprechen: Diese Progammierdiziplin unterstützt im hohen Maße eine spätere Reimplementierung des normalizer in RELFUN selbst. In Abschnitt 3 werden wir sehen, daß diese Vorgehensweise auch für den dort beschriebenen *mode-Interpreter* durchgehalten wurde.

Die Beschreibung der Implementierung besteht aus 3 Teilen: Teil 1 beschreibt in groben Zügen die Aufrufhierachien der einzelnen Funktionen und ordnet ihre Stellung in das Gesamtprogramm ein. Für zukünftige Erweiterungen findet der Entwickler in diesem Teil lokalisiert, an

[32]Unter den "relevanten Variablen" verstehen wir die Variablen, die im aktuellen oder einem weiter rechts stehenden chunk vorkommen. Es handelt sich hier also um eine Teilmenge aller Variablen in der Klausel.

welchen Stellen im Programm seine Erweiterungen eine sinnvolle Ergänzung bedeuten. In Teil 2 wird mehr auf die Funktionalität der Programmteile eingegangen. U. a. werden die verwendeten Selektoren, Konstruktoren und Prädikate beschrieben, die zusammen die abstrakte Syntax des normalizers definieren. Teil 3 ist das kommentierte Programmlisting. Es ist -nicht zuletzt des streng funktionalen Programmierstils wegen- ein wesentlicher Bestandteil der Implementierungshinweise. Das Listing befindet sich im Anhang dieser Arbeit.

2.4.1 Programmarchitektur

Der normalizer stellt zwei Schnittstellen zur Verfügung, über die man ihn sinnvoll Aufrufen kann:

normalize-database: DB → DB-NORM

normalize-clause: CLAUSE x PROC-INFO → CLAUSE-NORM x PROC-INFO

Die Funktion *normalize-database* normalisiert eine Liste von Klauseln, die als Datenbasis[33] bezeichnet wird. Die Funktion ihrerseits ruft für alle Klauseln aus der Datenbasis nacheinander die Funktion *normalize-clause* auf und liefert als Wert die normalisierte Datenbasis zurück.

Die Funktion *normalize-clause* normalisiert eine einzelne Klausel. Sie bekommt als Eingabe die zu normalisierende Klausel *cl* und Informationen *proc-info* über die Prozedur, zu welcher die Klausel gehört. Allerdings wird bisher *proc-info* im Programm nicht genutzt; es soll aber zukünftigen Erweiterungen dienen. Beispielsweise werden bisher nur LISP-Durchgriffe in den Regelsystemen $\mathcal{PEV}$ und $\mathcal{CSE}$ beachtet. Eine denkbare Verwendung wäre also, daß in *proc-info* Informationen darüber gesammelt wird, ob eine RELFUN-Prozedur seiteneffektfrei bzw. deterministisch ist. Die Regeln aus $\mathcal{PEV}$ und $\mathcal{CSE}$ könnten dann auch auf diese Prozeduren angewendet werden. Weitere nützliche Informationen wären, ob die Prozeduren einen *cut* enthalten oder direkt *unknown* liefern (also insbesondere scheitern).

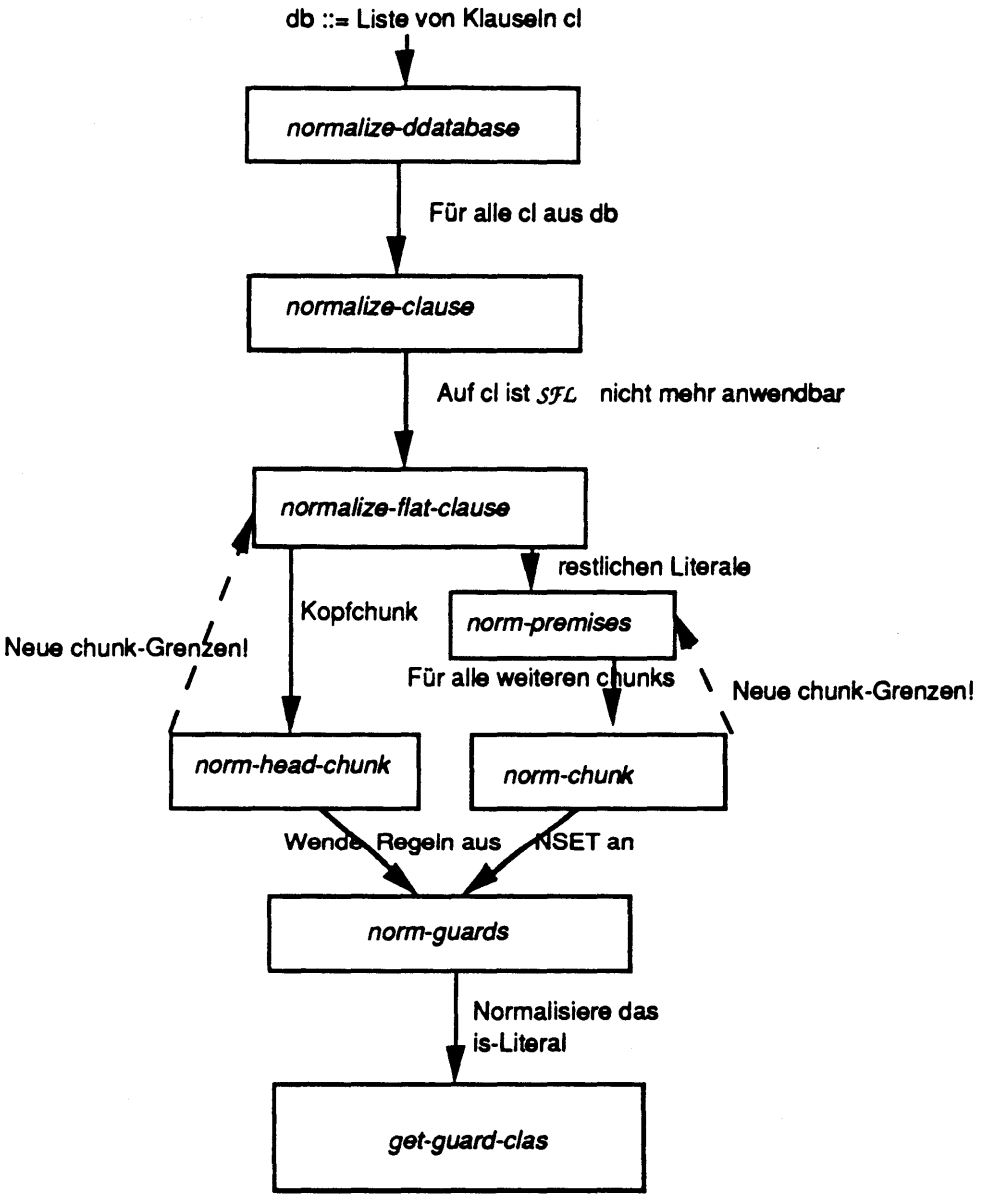
Der Aufbau von *normalize-clause* entspricht dem Kontrollalgorithmus aus dem Abschnitt 2.3. Zuerst wird die Klausel mittels der aus RELFUN importierten Funktion *flatten-struc-clause* abgeflacht und die Funktion *normalize-flat-clause* aufgerufen, die nun die Kontrolle für die Normalisierung übernimmt. Daraufhin werden die Grenzen des Kopfchunks bestimmt und mittels der Funktion *norm-head-chunk* die chunkweise Anwendung der Regelsysteme erreicht. Da die chunk-Grenzen sich noch ändern können, weil z. B. ein evaluatives Literal (die vorherige chunk-Grenze) ausgewertet wird und somit die Bedeutung eines chunk-guards bekommt,

[33]Es muß sich nicht notwendigerweise um die gesamte RELFUN-Datenbasis handeln; vielmehr kann eine beliebige Teilmenge betrachtet werden.

wird dieser Fall innerhalb *normalize-flat-clause* erkannt und erneut *norm-head-chunk* mit den neu berechneten chunk-Grenzen aufgerufen.

Ist der Kopfchunk normalisiert, werden die restlichen Prämissen mittels der Funktion *norm-premises* bearbeitet. Hierfür werden nun die chunk-Grenzen der *body-chunks* berechnet, die Funktion *norm-chunk* aufgerufen und gegebenfalls eine Korrektur der chunk-Grenzen vorgenommen.

Beide Funktionen *norm-head-chunk* und *norm-chunk* repräsentieren die innere Schleife des Kontrollalgorithmus aus Abschnitt 2.3 und rufen als "Arbeitspferd" die Funktion *norm-guards* auf, die letztendlich entscheidet, welches Regelsystem aus NSET zur Anwendung kommt. In *norm-guards* wird die Funktion *get-guardClas* aufgerufen, die im wesentlichen die is-Normalform (Abschnitt 2.1) für die is-Primitive berechnet. Wir können also nachstehendes Architekturschema festhalten:



2.9 Grobe Architektur des normalizers

2.4.2 Funktionale Beschreibung

Der aktuelle Stand der Normalisierung einer Klausel ist durch

i.) die bereits abgearbeiteten und noch abzuarbeitenden Literale und
ii.) die Datenstruktur *cl-info*

im normalizer repräsentiert. Zu den abgearbeiteten Literalen zählen wir Literale, die einem chunk angehören, der bereits durch *norm-chunk* (bzw. *norm-head-chunk*) normalisiert worden

ist. Innerhalb des aktuellen chunks müssen wir etwas genauer differenzieren, da Literale beispielsweise nach verschieben einer chunk-Grenze mehrmals betrachtet werden können. Genaueres findet man hierzu unter den Hinweisen zur Implementierung der Funktion *norm-chunk* und *norm-head-chunk*.

Die Datenstruktur *cl-info* beinhaltet im wesentlichen Informationen über die statisch festgestellten Bindungen der Variablen in einer Klausel. Der Aufbau von *cl-info* ist in Bild 2.10 wiedergegeben.

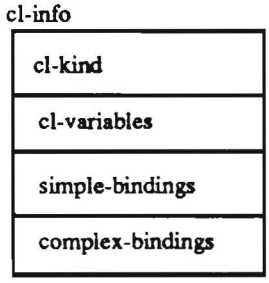


Bild 2.10 Aufbau der Datenstruktur *cl-info*.

Der Eintrag *cl-kind* zeigt an, ob es sich um eine hn- oder ft-Klausel handelt. Im Eintrag *cl-variables* wird vermerkt, welche Variablen bis zum aktuellen Transformationszeitpunkt in der Klausel vorgekommen sind. Diese Information ist für den Test der Anwendbarkeit einer Regel aus $\mathcal{DIP}$ und zur Berechnung der is-Normalform wichtig. Der Eintrag *simple-bindings* und *complex-bindings* verwalten die Bindungen von Variablen an *simple-terms* (Variablen oder Konstanten) bzw. an *complex-terms* (zusammengesetzte Terme). Es handelt sich jeweils um Listen der Form: ((*variable binding*)...). Die *simple-bindings* kommen in den Regelsystemen aus NSET nicht vor, da dort mittels der Aktion aus DIP_1, einfache Bindungen sofort propagiert werden. In der Implementation wird die Propagierung verzögert und erst dann vorgenommen, wenn ein Literal abgearbeitet wird, deswegen sind in *simple-bindings* die Bindungen gespeichert. Der Zugriff auf die Struktur und deren Komponenten erfolgt über Selektoren und Konstruktoren, die nachfolgend aufgeführt sind:

Selektoren

s-cl-info-kind: CL-INFO → CL-KIND

s-cl-info-variables: CL-INFO → CL-VARIABLES

s-simple-bindings: CL-INFO → SIMPLE-BINDINGS

s-complex-bindings: CL-INFO → COMPLEX-BINDINGS

Die Semantik der obigen Selektoren bedarf keiner weiteren Erklärungen. Um konkrete Bindungen von Variablen zu selektieren existiert der Selektor:

s-binding: VARIABLE x BINDINGS → TERM

Er selektiert für VARIABLE die Bindung aus der Bindungsumgebung BINDINGS. Existiert keine Bindung, wird die VARIABLE als Wert zurückgegeben. Dies ist im Einklang damit, daß eine ungebundene Variable sich selbst denotiert.

Konstruktoren

mk-cl-info: CL-KIND x CL-VARIABLES x SIMPLE-BINDING x COMPLEX-BINDINGS
→ CL-INFO

Auch hier ist die Semantik des Konstruktors offensichtlich.

Der Rest dieses Abschnitts beschreibt informell die Funktionalität der wichtigsten Funktionen des normalizers. Es wird noch einmal daran erinnert, daß das kommentierte Listing im Anhang einen zentralen Bestandteil der Funktionsbeschreibung darstellt und zusammen mit den nachstehenden Erläuterungen betrachtet werden sollte.

normalize-database: DB → DB-NORM
Die einzugebende Datenbasis DB wird in eine semantisch äquivalente Datenbasis DB-NORM transformiert. In DB-NORM' sind ausschließlich Kern-Klauseln enthalten

normalize-clause: CLAUSE x PROC-INFO → CLAUSE-NORM x PROC-INFO
normalize-flat-clause: CLAUSE x PROC-INFO → CLAUSE-NORM x PROC-INFO
Die einzugebende Klausel CLAUSE wird in eine semantisch äquivalente Kern-Klausel CLAUSE-NORM transformiert. Das zweite Argument PROC-INFO wird nicht weiter beachtet. Es soll in zukünftigen Erweiterungen des normalizers Informationen über die Prozedur, zu der die Klausel gehört, enthalten und innerhalb der Funktion aktualisiert werden. Die Funktion selektiert zuerst den Klausel-tag, den Klauselkopf sowie die Prämissen der Klausel. Anschließend werden die guards des Kopfchunks (mittels *collect-guards*) zusammengefaßt und der Kopfchunk (mittels *norm-head-chunk*) normalisiert. Das Ergebnis von *norm-head-chunk* unterscheiden wir in drei STATUS-Fälle:
1.) (STATUS = unknown) Das Regelsystem $\mathcal{UNP}$ ist zur Anwendung gekommen. Die restlichen Prämissen (PREM-REST) werden ignoriert und die Normalisierung der Klausel ist abgeschlossen.
2.) (STATUS = lisp-evaluate) Es wurde ein auswertbarer LISP-builtin ($\mathcal{PEV}$kann angewendet werden) entdeckt. Bei der Bestimmung der chunk-Grenze ist der LISP-Ausdruck noch als *call-*

Literal erkannt worden, so daß nun die chunk-Grenze, nachdem der Wert des Ausdrucks berechnet wurde, neu ermittelt werden muß. Dies geschieht über einen rekursiven Aufruf von *normalize-flat-clause*. In der neuen Eingabeklausel sind die bei der bisherigen Normalisierung eliminierten Literale nicht mehr vorhanden und der LISP-Ausdruck ist durch seinen Wert ersetzt. Die *simple-bindings*, die bisher festgestellt worden sind, werden zuvor über die neue Klausel propagiert. Dies ist wichtig, da Literale, die eine solche Bindung wegen der Regel DIP_1 erzeugten, womöglich eliminiert worden sind und eine erneute Abarbeitung der Klausel diese Bindung nicht mehr erkennen würde[34].

3.) (STATUS = ok) Die Bearbeitung des Kopfchunks ist abgeschlossen und hat als Ergebnis u. a. die aktuelle *cl-info* geliefert. Die weitere Verarbeitung wird mit diesen Informationen an die Funktion *norm-premises* übergeben.

norm-premises: CL-NEW x CL-REST x CL-INFO x PROC-INFO
→ CL-NORM x PROC-INFO

Der Aufbau von *norm-premises* ist ähnlich dem von *normalize-flat-clause*, mit dem Unterschied, daß hier die *body-chunks* behandelt werden und die Kern-Klausel im Argument CL-NEW zusammengesetzt wird. Als zentrale Funktion wird *norm-chunk* aufgerufen (anstatt *norm-head-chunk* wie oben). Im Argument CL-INFO wird die Struktur *cl-info* nach jeder chunk-Bearbeitung aktualisiert.

norm-head-chunk: CL-KIND x CONCL x GUARDS PREM-REST
→ STATUS x HEAD-CHUNK x CL-INFO

Die Funktion verhält sich fast genauso wie *norm-chunk* (s.u), wobei das Kopfliteral (Argument CONCL) besonders behandelt wird, indem die *simple-bindings* auch in das Kopfliteral propagiert werden.

norm-chunk: GUARDS x PREM-REST x CL-INFO
→ STATUS x CHUNK x CL-INFO

Die Funktion *norm-chunk* normalisiert den chunk, der durch die chunk-guards und dem call-Literal, welches das erste Literale der Liste *prem-rest* ist, gegeben ist. Die guards werden mittels der Funktion *norm-guards* abgearbeitet. Existiert kein call-Literal, muß der letzte guard (das foot-Literal) gesondert behandelt werden, da *norm-guards* im allgemeinen die Reihenfolge der guards nicht beibehält. Im obigen Fall denotiert der letzte guard den Wert der Klausel, er muß also in irgendeiner Form weiterhin die foot-Position behalten. Dies wird garantiert, indem an das Ergebnis von *norm-guards* noch die Funktion *norm-foot* nachgeschaltet wird. Der Wert von *norm-chunk* setzt sich also folgendermaßen zusammen: Signalisiert *norm-guards*, daß $\mathcal{UNP}$ angewendet wurde, so wird dies im Rückgabewert STATUS weitergereicht. Der zurück-

[34]Man beachte, daß die Propagierung der einfache Bindungen nicht sofort geschieht, sondern für jedes Literal verzögert wird, bis es durch eine Regel aus NSET abgearbeitet wird.

zugebende *chunk* und die *cl-info* bleiben unbestimmt. Handelt es sich beim call-Literal um ein vollinstantiierten LISP-Ausdruck, so wird das über den Wert STATUS an den Aufrufer zurückgereicht und der LISP-Ausdruck mittels der aus RELFUN importierten Funktion *lisp-exec*[35] ausgewertet. *Chunk* ist somit kein richtiger chunk, sondern nur eine Liste von guards. Die *cl-info* wird in diesem Fall unverändert übergeben. Im Normalfall (STATUS = ok) ist der chunk gemäß der Anweisungen 3 des Kontrollalgorithmus abgearbeitet und die *cl-info* aktualisiert.

norm-guards: GUARDS x VAR-BEFORE x BINDINGS-S x BINDINGS-C
→ NEW-GUARDS x BINDINGS-S x BINDINGS-C x VAR-AFTER

Die Funktion *norm-guards* ist das "Arbeitspferd" des normalizers. In ihr sind beispielsweise beinahe alle Regelsysteme aus NSET codiert. Die Funktion hat eine Art Fixpunktsemantik, denn ihre Vorgehensweise arbeitet zuerst alle guards ab, erzeugt gegebenfalls aufgrund der komplexen Bindungsumgebung neue is-Literale[36] und bearbeitet in diesem Fall die guards zusammen mit den erzeugten is-Literale erneut, solange bis keine neuen is-Literale mehr erzeugt werden. Die guards werden in drei Kategorien unterschieden. Zum einen GUARD-REST, das sind die in einer Iteration noch zu bearbeitenden Literale. NEW-GUARDS sind bereits abgearbeitete guards, die auch bei einer erneuten Iteration nicht weiter beachtet werden brauchen. CONFLICT-GUARDS sind ebenfalls schon abgearbeitet, müssen jedoch bei der nächsten Iteration wieder in die Liste der zu betrachtenen guards aufgenommen werden.

Für jede Iteration gilt anfangs die komplexe Bindungsumgebung BINDINGS-C, die beim Aufruf übergeben wird. Innerhalb der Iteration werden neue komplexe Bindungen in NBINDINGS-C zwischengespeichert. Stoppt die Iteration, ergeben BINDINGS-C und NBINDINGS-C zusammen die neue Bindungsumgebung für den nächsten chunk. Andernfalls wird NBINDINGS-C vor jeder Iteration zur leeren Liste initialisiert. Die einfachen Bindungen werden in BINDINGS-S gesammelt und überdauern, anders als die neuen komplexen Bindungen , die Iteration. Eine neue Iteration beginnt immer dann, wenn GUARD-REST = nil gilt und die Funktion *collect-is-set-from-new-bindings* keine neuen is-Literale erzeugt.

Innerhalb der Iteration werden die guards nacheinander mittels *get-guardClas* klassifiziert. Die guard-Klasse bestimmt letztendlich, welche Regel anzuwenden ist. Entsprechend der Regel wird dann das resultierende Literal berechnet, die Bindungen aktualisiert, sowie die Vorkommen der neuen Variablen notiert.

---

[35] Es existieren noch gewisse Einschränkungen bei den Argumenten eines in RELFUN erlaubten LISP-Aufrufs, die innerhalb von *lisp-exec* getestet werden. Je nach Implementation können an dieser Stelle also durchaus Warnungen oder Fehlermeldungen vom RFM-System generiert werden.

[36] Hierdurch wird die bedingte Aktion 2 der Regel DIP_1 realisiert.

norm-foot: TERM x BINDING-C → TERM
Handelt es sich um eine ft-Klausel mit einem denotativen foot-Literal und ist der Term t schon vorher vorgekommen, so existiert eine Bindung _v → t in *bindings-c*. In diesem Fall wird das foot-Literal t durch _v ersetzt. Es wird also die Anwendbarkeit von *CSE* überprüft. Dies kann nicht innerhalb der Funktion *norm-guards* geschehen, da dort die Reihenfolge der Literale i. a. nicht erhalten bleibt.

get-guardClas: GUARD x VARIABLES x BINDINGS-C → GUARD x TAG
Die Funktion *get-guardClas* klassifiziert einzelne guards in Bezug auf die bisher vorgekommenen Variablen und der aktuellen komplexen Bindungsumgebung. In dem zu klassifizierenden guard wurde bereits die einfachen Bindungen propagiert. Handelt es sich um ein is-Literal wird gemäß Abschnitt 2.1 als Rückgabewert die is-Normalform übergeben. Die guard-Klasse wird durch ein tag angegeben. Es wird folgendermaßen unterschieden:

unknown Falls guard = *unknown* gilt oder es sich um ein is-Literal handelt, das zwei unterschiedliche Konstanten miteinander vergleicht.

deno-prem Falls es sich um ein denotatives Literal oder um ein is-Literal mit zwei bis auf den passivierenden inst-Operator syntaktisch gleichen Seiten handelt[37]. Im zweiten Fall wird der rhs-Term des is-Literals zurückgegeben.

new-simple Dieser und die folgenden tags beschreiben denotative is-Literale. Der tag **new-simple** beschreibt is-Literale dessen linke Seite eine Variable ist, die zum erstenmal in der Abarbeitung vorkommt und dessen rechte Seite eine Variable oder Konstante ist. Man beachte, daß der guard zuvor auf is-Normalform gebracht wird. Beispielsweise wird der guard (is a _v), wobei _v nicht in VARIABLES vorkommt, zu **new-simple** klassifiziert und als neuer guard wird (is _v a) zurückgegeben. Wegen diesen Überlegungen ist auch verständlich, daß z. B. der tag simple-new (oder complex-new usw.) nicht existiert.

new-complex Dieser tag beschreibt is-Literale mit einer neuen Variablen auf der linken und einem komplexen Term auf der rechten Seite.

unbound-simple Entsprechend wie **new-simple**, nur daß die Variable nicht zum erstenmal vorkommt.

unbound-complex Entsprechend wie **new-complex**, nur daß die Variable nicht zum erstenmal vorkommt.

complex-complex Beide Seiten des is-Literals sind entweder komplexe Terme oder Variablen, die auf einer linken Seite der Bindungen aus *bindings-c* vorkommen.

collect-guards: PREMISES → GUARDS x PREMISES
Aus der Liste PREMISES werden von links nach rechts solange Literale in GUARDS gesam-

[37]Zum Beispiel: (is (f _x) `(f _x))

melt, bis ein call-Literal erreicht wird. Das call-Literal bedeutet die chunk-Grenze und wird als erstes Element zusammen mit den restlichen Prämissen als zweiten Wert zurückgegeben. Die aufgesammelten guards werden als Liste in GUARDS übergeben.

rewrite: TERM x BINDINGS → TERM
Alle Vorkommen von linken Seiten aus *bindings* im Term werden durch die entsprechenden rechten Seiten ersetzt. Der so entstandene Term wird als Wert zurückgegeben.

rhs-rewrite: LHS x RHS x BINDINGS → BINDINGS
Falls eine Regel LHS → *t* in BINDINGS existiert, so wird *t* durch RHS ersetzt und die so entstandene Bindungsumgebung als Wert zurückgegeben.

rewrite-with-isset: TERM x IS-SET → TERM
Die Menge der is-Literale in IS-SET wird als Bindungen interpretiert und entsprechend der Funktion *rewrite* auf einen Term angewandt.

2.4.3 Debug-Hilfe

Bei der Implementierung des normalizer wurde gleichzeitig ein kleines Tool zur Unterstützung der Fehlersuche entwickelt. Im wesentlichen ist dies ein eigener kleiner *tracer* und ein *error-handler*. Die Funktionen sind noch im Programmtext des normalizers eingebunden, jedoch defaultmäßig inaktiv (*tracer-level* und *error-level* sind auf "0" gesetzt). Das Tool kann über folgenden Funktionen genutzt werden:

set-trac: LEVEL → LEVEL
Setzt die globale Variable *trace-level* auf einen neuen Level.

set-err: LEVEL → LEVEL
Setzt die globale Variable *error-level* auf einen neuen Level.

trace-in-handler: F-NAME x ARGS x ENVIRONMENT x &LEVEL → NIL
Tracen eines Funktionsaufrufs. Die Argumente sind der Funktionsname, die Funktionsargumente und die Umgebung der Funktion. Der Level kann noch extra angegeben werden, wenn bei diesem Aufruf nicht der globale Level *tracer-level* beachtet werden soll.
Bei einem LEVEL ≤ 0 oder LEVEL > 2 gibt die Funktion nur den LEVEL zurück (Kein sinnvoller Level.) Ansonsten erfolgt eine Meldung darüber, daß die Funktion F-NAME observiert wird und für LEVEL = 2 wird weiterhin die *documentation*[38] angezeigt und ansonsten genauso wie für LEVEL = 1 die Argumente und das environment ausgegeben.

[38]Mittels der Common-LISP-Funktion (*documentation func-name* 'function)

trace-out-handler: F-NAME x VALUE &LEVEL → VALUE
Bei einem LEVEL ≤ 0 oder LEVEL > 2 gibt die Funktion sofort VALUE zurück (Kein sinnvoller Level.) Ansonsten erfolgt eine Meldung darüber, daß die Funktion F-NAME terminiert und der Wert VALUE wird angezeigt und als Wert von *trace-out-handler* zurückgegeben.

error-handler:ERR-NO x ERROR-ARGS → *t*
Es werden 5 verschiedene Levels (*error-level* = 0,...,3) unterschieden. Zusätzlich existiert noch die globale Variable *error-stream*, dessen erstes Element den Prozedurnamen und die Klausel enthält, die aktuell normalisiert werden. Die Fehlernummer ERR-NO bestimmt den Fehlertext, der im Anhang (-> Listing) nachgelesen werden kann. Gilt *error-level* = 0, so wird keine Meldung ausgegeben. Eine Meldung auf das aktuelle Textfenster wird beim *error-level* = 1(2) ausgegeben, wobei beim *error-level* = 2 die Normalisierung unterbrochen wird. Gilt *error-level* = 4, so erfolgt die Meldung nicht auf das Textfenster, sondern wird in die globale Variable *error-stream* vermerkt und kann nach der Normalisierung betrachtet werden.

3 Globale Datenflußanalyse von RELFUN

Eine RELFUN-Prozedur p/n ist durch ihre m > 0 Klauseln (ct_i (p_i t_{i1} ... t_{in}) ...), $1 \le i \le m$, definiert. Zu den Annehmlichkeiten der deklarativen Programmiersprachen gehört, daß der Programmierer keinerlei Spezifikation der Prozeduren geben braucht. Der Abarbeitungsmechanismus muß dementsprechend allgemein gehalten werden. Üblicherweise verlangen die Prozeduren aber gar nicht diese Allgemeinheit, sondern besitzen vielmehr typische Aufrufmuster während des Programmablaufs. Sinn und Zweck einer (informierten) Compilation sollte es sein, diese speziellen Muster zu beachten und effizientere Instruktionen als in dem (uninformierten) allgemeinen Fall zu generieren. Typische imperative Programmiersprachen wie etwa *MODULA 2, PASCAL, FORTRAN, C,...* unterstützen die (informierte) Compilation mit einem mehr oder weniger strengen Typkonzept. Bei der Definition einer Prozedur muß man angeben, welchem Typ die Argumente der Prozedur angehören. Oft existieren Standard-Typen wie beispielsweise *integer, real, char* usw. oder vom Benutzer definierte Typen wie etwa die *Record-Typen* in PASCAL. Die Typvereinbarungen müssen konsistent sein, was innerhalb der Compilierungsphase nachgeprüft wird. Eine Typverletzung wird vom Compiler an den Programmierer zurückgemeldet, und es wird kein ausführbarer Programmcode erzeugt. Für die Programmausführungsphase bedeutet das eine gewisse Garantie, daß alle Prozeduren mit typgerechten Argumenten aufgerufen werden. Je strenger das Typkonzept in der Sprache realisiert ist, desto mehr kann man dieser Garantie vertrauen. MODULA 2 ist zum Beispiel eine Programmiersprache mit einem sehr ausgeprägten Typkonzept, während es in C bekanntermaßen sehr lose realisiert ist.

In RELFUN werden den Argumenten einer Prozedur keine Typen zugeordnet, da hier kein Typkonzept verwendet wird. Dennoch soll dieser Abschnitt aufzeigen, wie typähnliche, im Programm implizit vorhandene Information über die Argumente t_{ij} automatisch aus dem Source-Code extrahiert werden können. Konkret wird das Ein- und Ausgabeverhalten der Argumente untersucht, welches wir als *modes* der Prozedur bezeichnen. Die modes beschreiben, wie die Argumente der Prozedur bei einem Aufruf instantiiert sind und unterscheiden im wesentlichen, ob es sich um Grundterme oder freie Variablen handelt. Kann man keine genauen Angaben machen, wird diese Unwissenheit als ein eigenständiger mode ausgedrückt. Die modes werden mit *c, f* und *d* bezeichnet, wobei *c* (*closed*) für Grundterme, *f* (*free*) für freie Variablen und *d* (*don't know*) für "keine genaue Angabe machbar" stehen. In dieser Arbeit wird der mode *c* weiter durch *t* (*true*) verfeinert, weil die Konstante true in RELFUN einen besonderen Stellenwert als impliziter Rückgabewert einer hn-Klausel besitzt. Den mode <*c, c, f*> für eine Prozedur p/3 kann man dann so interpretieren, daß die Prozedur p/3 zwei Input-Argumente an den beiden ersten Argumentstellen hat, d. h. hier stehen immer Grundterme (mode = *c*) und es können somit keine Werte durch Instantiierung von Variablen

zurückgegeben werden. Das dritte Argument hingegen ist ein Output-Argument, d. h. es ist immer ungebunden (mode = *f*) und wird üblicherweise dazu verwendet, Werte durch Bindung zurückzuliefern. Die bekannte relationale Definition der Prozedur append/3 hat gerade das oben beschriebene Muster <*c, c, f*>, falls append/3 nur wie eine Funktion verwendet wird, die zwei gegebene Listen konkateniert. Ein weiteres denkbares Aufrufmuster für append/3 wäre etwa <*f, c, c*>, was so interpretiert werden kann, daß append/3 im Programm nur dazu verwendet wird, eine Liste l1 zu finden, die mit vorgegebener Liste l2 konkateniert die ebenfalls vorgegebene Liste l3 ergibt.

Der Inhalt dieser Arbeit soll zeigen, wie die modes von Prozeduren durch eine Datenflußanalyse (oder auch mode-Analyse) automatisch berechnet werden. Hierbei kann man aber nicht erwarten, daß stets der strengste mode einer Prozedur gefunden wird. Die Begründung ist recht einfach und wird anhand des Halteproblems gegeben, das als unentscheidbar bekannt ist. Betrachten wir dazu folgendes Programmfragment:

	LISP-artige Syntax:	PROLOG-artige Syntax:
Goal:	(q 1) (p _v) (q _w)	?- q(1), p(V), q(W).
Source:	(hn (q _t))	q(T).
	(hn (p _u) ...)	p(U) :- ....

Die Prozedur q/1 wird innerhalb der Goal-Konjunktion nur dann mit einem *nonground*-Term aufgerufen (nämlich im 3. Konjunkt), falls die Abarbeitung von (p _v) erfolgreich terminiert. Man kann aber das Halteproblem für eine Turing-Maschine in die Frage nach einer erfolgreichen Termination des Prozeduraufrufs p/1 codieren [Man87]. Letztendlich ist also nicht entscheidbar, ob die Prozedur q/1 den mode <*c*> oder <*d*> besitzt. Wir sprechen deswegen auch von einer "konservativen" Bestimmung der modes und meinen damit, daß die modes zwar nicht streng (scharf) berechnet werden, aber der Korrektheitsanforderung genügen, daß, wenn eine Klausel mit einer gewissen Instantiierung der Argumente aufgerufen wird, ist dies auch verträglich (kompatibel) mit den berechneten modes der Prozedur ist. Im obigen Beispiel müßten wir uns etwa mit dem unbestimmten mode <d> für q/1 zufrieden geben.

Der Rest von Abschnitt 3 gliedert sich in 3 Teile. Abschnitt 3.1 zeigt anhand von Anwendungen einige Einsatzmöglichkeiten der modes. In Abschnitt 3.2 wird die Datenflußanalyse formal als abstrakte Interpretation eines RELFUN-Programms beschrieben und in Abschnitt 3.3 findet man einen Implementierungsvorschlag für einen *mode-Interpreter* sowie Hinweise zur LISP-Implementierung.

3.1 Anwendungen der mode-Informationen

Die Anwendungen der mode-Informationen liegen nach [Som86] hauptsächlich in

(a) Vermeidung von Programmierfehlern,
(b) Dokumentation von Programmen,
(c) Kontrolle der Programmausführung,
(d) Steigern der Programmeffizienz.

Die Punkte (a) und (c) werden in Abschnitt 3.1.2 unter dem Stichwort *Vorverarbeitung* aufgegriffen. Die *Determinismus-Analyse* in Abschnitt 3.1.1 und die Einbindung der modes in ein Indexierungskonzept auf RFM-Ebene in Abschnitt 3.1.3 kann man den Punkten (c) und (d) zuordnen. Zu Punkt (b) wollen wir nur vermerken, daß die modes sicherlich eine große Verständnishilfe für einen Programmierer bedeuten, da er dadurch ein Gefühl dafür bekommt, wie die Prädikate im Programm verwendet werden und er so leichter deren Bedeutung erschließen kann.

3.1.1 Modes und Determinismus-Analyse

Im allgemeinen ist es unentscheidbar, ob die Berechnungsfolge für ein Prädikat terminiert (s. o.). In einigen Fällen kann man jedoch hinreichende (aber oft nicht notwendige) Kriterien für die Termination angeben. Für PROLOG-Prädikate[39] wurde beispielsweise in [Mel85] festgestellt:

Ein Prädikat P ist deterministisch, falls
(1) jede Klausel, bis auf höchstens der letzten, den *cut*-Operator als Konjunkt enthält.
(2) Jedes Prädikat, welches nicht vor einem *cut*-Operator in den obigen Klauseln vorkommt, selber deterministisch ist.

Es handelt sich auch hier um schwache (konservative) Bedingungen und jedes Prädikat, das (1) und (2) erfüllt, ist sicherlich deterministisch. Es gibt jedoch (nachweisbar) deterministische Prädikate, die diese Bedingungen nicht erfüllen. Betrachten wir aber zuerst folgendes PROLOG-Beispiel, das als "PROLOG-Klassiker" *naive-reverse* bekannt ist (hier etwas unüblich geschrieben).

[39] Die Ergebnisse sind ohne weiteres direkt auf RELFUN übertragbar.

```
nrev([X|Y], Z) :- !, nrev(Y, Z1), append(Z1, [X], Z).
nrev([],[]).

append([X|Y], Z, [X|Y1]) :- !, append(Y, Z, Y1).
append([], X, X).
```

Folgen wir den Ausführungen aus [Mel85] und drücken mit dem Wahrheitswert von <p> aus, ob das Prädikat als deterministisch nachweisbar ist (<p> = TRUE) oder nicht (<p> = FALSE), dann können wir für das Beispiel *naive reverse* ein Gleichungssystem angeben, welches die obigen Bedingungen (1) und (2) modelliert.

```
<nrev>   = <nrev> and <append>
<append> = <append>
```

Der Operator "and" ist als die logische Konjunktion über den Wahrheitswerten {FALSE, TRUE} von <p> zu interpretieren. Die Determinismus-Analyse kann nun auf die Lösung des Gleichungssystems reduziert werden. Es können aber mehrere Lösungen existieren und nicht alle erfüllen unsere Erwartungen. Zum Beispiel findet man immer die triviale Lösung, daß für alle p_i aus dem Gleichungssystem $<p_i>$ = FALSE gilt. Deswegen formulieren wir die Aufgabe so um, daß wir nach der minimalen Lösung suchen. Hierzu betrachten wir folgenden (trivialen) Verband:

```
FALSE
  |
TRUE
```

Eine minimale Lösung bekommt man nun dadurch, daß man Werte über das Gleichungssystem propagiert und diesen Vorgang iteriert. Zuerst werden alle $<p_i>$:= TRUE gesetzt (man nimmt an, daß alle Prädikate deterministisch sind!). Im weiteren setzt man die Werte in den rechten Seiten der Gleichungen ein und berechnet so die neuen Werte $<p_i>$. Mit den berechneten Werten wird dann eine erneute Iteration gestartet. Wird ein Fixpunkt erreicht (keine Änderung der Werte nach einer Iteration) stoppt die Iteration und eine minimale Lösung ist gefunden. Die Termination der Iteration wird durch den endlichen Wertebereich und die Monotonie des and-Operators garantiert. Im *naive reverse* Beispiel ist der Fixpunkt schon nach der ersten Iteration erreicht, und man kann für beide Prädikate notieren, daß sie deterministisch sind.

Das nächste Beispiel zeigt, wie man modes benutzen kann, um die Determinismus-Analyse zu verfeinern.

```
human(X) :- mother(X, Mother), human(Mother).
```

```
animal(X) :- human(X).

mother(fred, jane).
mother(abel, eve).
```

Es ergeben sich folgende Gleichungen:

<human> = <mother> and <human>
<animal> = <human>
<mother> = FALSE

Die Gleichung <mother> = FALSE ist entstanden, weil das Prädikat mother/1 nicht der Bedingung (1) genügt. Startet man die Iteration mit

<human> = TRUE
<animal> = TRUE
<mother> = TRUE

ergeben sich die Werte in den Iterationen wie folgt:

<human> = TRUE
<animal> = TRUE
<mother> = FALSE

<human> = FALSE
<animal> = TRUE
<mother> = FALSE

<human> = FALSE
<animal> = FALSE
<mother> = FALSE

Letztendlich konnte keines der Prädikate als deterministisch nachgewiesen werden. Das Prädikat human/1 ist aber seiner normalen Intention nach nur mit einem grundinstantiierten Wert, der einen konkreten Menschen repräsentiert, sinnvoll aufzurufen. Nehmen wir also an, für human/1 ist der mode <*c*> für das Eingabeargument bekannt. Dann kann man schließen, daß auch das Prädikat mother/2 immer mit einem Grundterm in der ersten Argumentstelle aufgerufen wird. (In unserer Annahme hat es folglich den mode <*c*,*f*>). Weiter halten wir fest, daß in den Klauseln von mother/2 an der ersten Argumentstelle verschiedene Grundterme stehen (nämlich fred und abel ->*clash*). Ein Aufruf von mother/2 kann also nicht mehr als eine Lösung liefern. Erweitert man die Bedingung (1) für unseren Test auf Determinismus eines

Prädikats um diese Überlegung, würde man mother/2 als deterministisch erkennen und das gleiche auch für die Prädikate human/1 und animal/1 feststellen.

Mit der Information über das deterministische Verhalten der Prädikate kann ein Compiler-System gesteuert werden, so daß der Compiler Instruktionen erzeugt, die verhindern, daß zur Laufzeit unnötige Lösungsversuche unternommen (d. h. *Choicepoints* aufgebaut) werden. Im Sinne von Complab betrachtet man die Information auf höherer Ebene. So ist es beispielsweise denkbar, die Information innerhalb der "Klassifizierten Klauseln" (siehe Abschnitt 1.1) explizit zu machen. Eine weitere Möglichkeit wäre, daß eine horizontale Compilations-Komponente (ähnlich dem normalizer) an den entsprechenden Stellen sogenannte *green-cuts*[40]einfügt. Das obige Beispiel würde in diesem Fall folgendermaßen aussehen[41]:

```
human(X) :- mother(X, Mother), human(Mother).

animal(X) :- human(X).

mother(fred, jane) :- !.
mother(abel, eve).
```

3.1.2 Modes und Vorverarbeitung/Fehlererkennung

Dieser Abschnitt skizziert, wie mode-Informationen statische Vorverarbeitung unterstützen sowie Hilfestellungen beim Erkennen von trivialen Programmierfehlern geben. Mit der im vorherigen Abschnitt untersuchten Determinismus-Analyse und der damit verbundenen Generierung von *green-cuts*, haben wir schon ein Beispiel für die mode-unterstützte Vorverarbeitung kennengelernt.

Im weiteren konzentrieren wir unsere Aufmerksamkeit auf die *builtins* in logischen Programmiersprachen. Von PROLOG kennen wir u. a. die builtins *var/1 und is/2*. Man kann für sie festhalten, daß sie entweder Aussagen über die Instantiierung ihrer Argumente treffen oder Anforderungen an ihre Argumente stellen, damit sie ordnungsgemäß arbeiten und nicht etwa einen Systemfehler verursachen. Beispielsweise wird mit var(X) getestet, ob X eine freie Variable ist, und der Ausdruck X is Y + Z führt in PROLOG zu einem Systemfehler, wenn die

---

[40] Ein *green-cut* ist ein normaler cut-Operator, der die Semantik des Programms nicht ändert, sondern nur unnötige Beweisversuche verhindert. Üblicherweise spricht man beim cut-Operator zusätzlich von einem *red-cut*, welcher ein wichtiger Bestandteil der Programmkontrolle ist und dessen Fehlen ein anderes Programmverhalten nach sich zieht. Ein Beispiel für einen *red-cut* ist der in der üblichen Fakultätsdefinition von PROLOG verwendete cut-Operator.

[41] Natürlich ist hier die Einführung des green-cuts nur dann korrekt, wenn human/1 nicht "unnormal" mit einer freien Variablen aufgerufen wird.

Variablen Y und Z zur Laufzeit nicht an eine Zahl gebunden sind. Ähnliche Anforderungen stellt RELFUN an einen LISP-Durchgriff, der immer grundinstantiiert sein muß.

Setzen wir nun die Kenntnis der modes voraus (entweder durch mode-Analyse oder Benutzerdeklaration), kann man in den Fällen, für welche die Variable X den mode *c* oder *f* besitzt, das builtin var(X) bereits statisch auswerten und das Scheitern (mode = *c*) bzw. den Erfolg (mode = *f*) von var(X) notieren. Auf fast dieselbe Art und Weise können Inkonsistenzen beim Aufruf von is/2 in PROLOG bzw. den LISP-Durchgriffen in RELFUN erkannt werden, falls eines der Argumente den mode *f* hat.

3.1.3 Modes und Indexierung in der RFM (WAM)

RELFUN-Klauseln werden im RFM-System (siehe Abschnitt 1.1 Bild 1.3) nach RFM-Instruktionen compiliert. Neben den Instruktionen, die die Klauseln repräsentieren, gibt es unter anderem auch *switch*-Instruktionen wie etwa *switch_on_term*, *switch_on_constant* und *switch_on_structure*, die es gestatten, prozedurweise optimierten RFM-Code mittels Indexierung zu generieren. Die Indexierung erfolgt häufig ausschließlich nach dem ersten Argument. Die Idee des Indexierens wird kurz an den 3 Klauseln

(hn (foo a 10))
(hn (foo b 10))
(hn (foo c 20))

vorgestellt. Eine Indexierung über das erste Argument bedeutet hier, daß eine Tabelle der Form ((a label1) (b label2) (c label3)) angelegt wird. Erfolgt ein Aufruf von foo/2 mit einer Konstante im ersten Argument wird zum entsprechenden Label der Tabelle verzweigt. Im Code-Bereich der RFM stehen, an der durch den Label repräsentierten Adresse beginnend, die RFM-Instruktionen der jeweiligen Klausel (oder gegebenenfalls Klauseln). Ist die Konstante in der Tabelle nicht vorhanden, wird ein *failure* ausgelöst. Die Indexierung über das erste Argument scheint zumindest im Beispiel eine gute Wahl[42] zu sein, da foo/2 über das erste Argument am stärksten differenziert ist. Wird allerdings durch eine mode-Analyse der mode <*f*, *c*> für foo/2 festgestellt, muß ein intelligenter Compiler erkennen, daß ein Aufruf von foo/2 an der ersten Argumentposition immer eine ungebundene Variable und an zweiter Argumentposition immer einen Grundterm besitzt. Es bringt demnach überhaupt keinen Sinn,

[42]Es ist auch eine relativ gute Heuristik, immer das erste Argument zum Indexieren zu wählen, weil beispielsweise in PROLOG häufig über das erste Argument "case-artig" die verschiedenen Fallunterscheidungen eines Prädikataufrufs abgehandelt werden.

über das erste Argument zu indexieren, vielmehr sollte der Compiler nun das zweite Argument für die Indexierung wählen.

Diese Art Ein flexibler Indexierungsansatz dieser Art wird in [Sin91] untersucht. Hierbei wurde zum einen ein Konzept vorgeschlagen, das die mode-Informationen für eine horizontale Vorverarbeitung nutzt und dabei im wesentlichen die Argumente einer Prozedur permutiert, so daß das für die Indexierung vielversprechendste Argument an die erste Position rückt. Hierdurch wird erreicht, daß alle Arbeit auf einer hohen Ebene erledigt wird, was genau der Philosophie vom *Complab* (siehe Abschnitt 1) entspricht. Die niedrigeren RFM-Ebenen würden hiervon nicht berührt werden und es wären somit auch keine neuen Instruktionen nötig. Bei unserem Beispiel, mit bekanntem mode <*f*, *c*> für foo/2, sähe das Ergebnis der horizontalen Compilation folgendermaßen aus:

```
(ft (foo _u _v) (foo' _v _u))
(hn (foo' 10 a ))
(hn (foo' 10 b))
(hn (foo' 20 c))
```

Das Prädikat foo/2 besteht jetzt nur noch aus einer einzigen Klausel, die das in den Argumenten permutierte Prädikat foo'/2 aufruft. Der mode für foo'/2 ist <*c*, *f*>.

Der zweite Ansatz, der in [Sin91] diskutiert wird, setzt auf die RFM-Instruktions-Ebene auf. Die oben erwähnten *switch*-Instruktionen beziehen sich implizit auf das X-Register mit der Nummer 1(1. Argument). Eine Verallgemeinerung der *switch*-Instruktionen, die den Zugriff auf beliebige X-Register erlaubt, gestattet es, aufgrund der mode-Informationen beispielsweise das zweite X-Register als Indexierungsregister zu wählen. Hierfür wird die neue RFM-Instruktion *set_index_number* vorgeschlagen. Der Vorteil dieser Vorgehensweise liegt in der flexiblen Auswahl der Argumentpositionen nach deren indexiert werden soll. Verschiedene Prozeduren können so auch nach unterschiedlichen Argumenten indexiert werden. Es ist sogar möglich, innerhalb einer Prozedur Partitionierungen zu finden und in den Partitionen nach jeweils dem geeignetsten Argument zu indexieren. Ein Nachteil dieser Variante ist die (portierungsabträgliche) Änderung bzw. Neueinführung von RFM-Instruktionen.

3.2 Abstrakte Interpretation von RELFUN

Betrachten wir nun die Generierung von mode-Informationen, die über eine abstrakte Interpretation eines RELFUN-Programms geschieht. Der formale Aufbau und die Definitionen sind stark an die Arbeit von Debray [Deb88] angelehnt. Die funktionalen Aspekte von

RELFUN machten an einigen Stellen Erweiterungen nötig, da sich obige Arbeit nur auf die rein relationale Sprache PROLOG bezieht.

Um die modes von einem RELFUN-Programm zu berechnen, simulieren wir die Ausführung des Programms. Statt mit konkreten Termen zu rechnen, betrachten wir eine endliche Anzahl von Termklassen, die die in einer Programmausführung vorkommenden möglichen Instantiierungen der Terme beschreiben. Dadurch wird die ursprünglich unendliche Programmdomäne der syntaktischen korrekten Terme auf die endliche Domäne Δ von Termklassen reduziert. Wir definieren Δ wie folgt:

Def 3.1

$\Delta ::= \{c, d, e, f, t\}$ ist eine Menge von Termklassen, wobei jede Klasse eine (i. a. nicht endliche) Menge von Termen darstellt. Es gilt:

e (empty) beschreibt die leere Menge. $e \simeq \{\}$

t (true) beschreibt den Term *true*. $t \simeq \{\text{true}\}$.

c (closed) beschreibt die Menge der grundinstantiierten Terme. Dies sind Konstanten sowie alle Terme, deren Argumente in *c* liegen.

$c \simeq \{t \mid t \text{ ist eine Grundterm}\}$

f (free) beschreibt die Menge der freien Variablen.

$f \simeq \{t \mid t \text{ ist eine freie Variable oder t ist an einen Term t' gebunden mit } t' \in f\}$

d (don't know) beschreibt alle Terme. $d \simeq \{t \mid t \text{ ist ein Term}\}$

Wir benutzen die Sprechweise, daß ein Term t durch die Klasse $\delta \in \Delta$ **approximiert** wird, falls $t \in \delta$[43] gilt.

Die modes beschreiben nicht einen konkreten Term, sondern alle möglichen Instantiierungen relativ zu einem Programm. Faßt man diese Instantiierungen zu einer Menge zusammen, stellt sich die Frage, ob man für alle Terme eine gemeinsame und möglichst gut approximierende Termklasse angeben kann. Als ein Maß dafür, wie gut oder schlecht eine Approximation ist, benutzen wir die Inklusionsbeziehung $\subseteq$ zwischen den Termklassen. Die Intention ist klar: je kleiner die Termklasse, desto schärfer ist die Approximation, und folglich die gewonnene Information gehaltvoller. Betrachtet man Δ bezüglich der Inklusion $\subseteq$, ergibt sich ein voll-

[43]Nach der Definition der Termklassen $\delta \in \Delta$ ist die Schreibweise "$t \in \delta$" nicht ganz sauber, aber es ist klar, was damit gemeint ist.

ständiger Verband. Die Termklasse *e* ist nicht von praktischer Interesse, sondern wird nur dafür gebraucht, daß in $(\Delta, \sqsubseteq)$ ein kleinstes Element existiert.

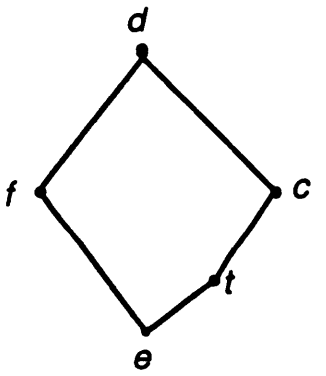


Bild 3.1 Der Verband $(\Delta, \sqsubseteq)$

Eine Menge T von Termen können wir nun durch die Instantiierung $i(T)$ der Menge T beschreiben:

$$i(T) ::= \sqcap\{\delta \in \Delta \mid T \sqsubseteq \delta\}$$

Der Operator "$\sqcap$" bezeichnet die Schnittbildung[44] im Verband $(\Delta, \sqsubseteq)$. Beispielsweise gilt für T = {a, true, `(f alpha)}, daß $i(T) = c$ und für T = {a, true , `(f _x)} gilt $i(T) = d$. Bezeichnen wir eine Variable *v*, die in einem Programm vorkommt, als Programmvariable, so kann *v* während der Laufzeit mit einer Menge von Termen instantiiert werden. Die Menge wird als *Instantiierungsstatus i-state(v)* der Programmvariablen *v* bezeichnet und ist in Def 3.2 definiert. Die in der Definition aufgeführte σ-*Aktivierung* (eine Abbildung V -> V der Programmvariablen V) einer Klausel C wird dem Umstand gerecht, daß bei einem Resolutionsschritt in der RELFUN-Abarbeitung die Variablen der Klauseln umbenannt werden, damit sie in den Klauseln eindeutig sind. Diese Umbenennung bezeichnen wir als σ-Aktivierung der Klausel.

[44] Hier verwendet zum Finden des kleinsten Elements.

Def. 3.2 *Instantiierungsstatus*

Sei V die Menge der Programmvariablen in einer Klausel C. Der Instantiierungsstatus ξ in einem Punkt der Klausel ist eine Abbildung

$$\xi: V \rightarrow \Delta,$$

so daß für jedes $v \in V$ und jede σ-Aktivierung von C während der Abarbeitung gilt:

$$\sigma(v) \Rightarrow^{45} t, \text{ dann gilt } t \in \xi(v)$$

Die natürliche Erweiterung ξ': Term -> Δ des i-state ξ auf einen beliebigen Term t wird im weiteren ebenfalls mit ξ bezeichnet, wenn aus dem Zusammenhang eindeutig hervorgeht, daß die Erweiterung von ξ gemeint ist. Am Anfang der Abarbeitung einer Klausel C gilt immer der initiale Instantiierungsstatus ξ_{init}. Er ist durch $\xi_{init}(v) = f$ für alle Variablen v in der σ-Aktivierung von C definiert. Eine Sequenz $<\delta_1, ..., \delta_n>$ von i-states wird als Instantiierungspattern (*i-pattern*) bezeichnet. Es dient dazu, Instantiierungen von Prozeduraufrufen (goals) zu beschreiben. Ausgezeichnete Instantiierungspattern sind die Klauselaufrufpattern *callp* und die Klauselerfolgpattern *succp*. Für eine n-stellige Prozedur p/n haben die zugehörigen Aufrufpattern *callp* eine Länge von n; das korrespondierende Erfolgspattern *succp* hat eine Länge von n+1. Der zusätzliche i-state im Erfolgspattern beschreibt den vom Prozeduraufruf zurückgegebenen Wert.

Wir haben schon das typische Aufrufpattern <*c, c, f*> von append/3 besprochen. Dies entspricht der Konkatenationsverwendung einer relationalen append-Definition in RELFUN; das Erfolgspattern hat dann die Gestalt <*c, c, c, t*>. Definiert man *append* funktional, wie etwa in Bild 3.2 gezeigt, so ergibt sich das entsprechenden Aufrufpattern zu $I_C = <c, c>$ und das Erfolgspattern zu $I_S = <c, c, c>$.

```
(ft (append-f (tup) _l2) _l2)
(ft (append-f (tup _h | _t) _l2)  (tup _h | (append-f _t  _l2)))
```

Bild 3.2 Funktionale Definition der Listenkonkatenation in RELFUN

Bei der Simulation einer Abarbeitung, die wir im weiteren *abstrakte Interpretation* (über der Domäne Δ) nennen wollen, geht es darum, innerhalb der Klausel aufgrund des aktuellen i-states das Aufrufpattern des nächsten goals zu bestimmen, daraus das entsprechende Erfolgspattern zu berechnen und den i-state daraufhin zu aktualisieren.

Für die abstrakte Interpretation brauchen wir ein Gegenstück zur Unifikation in RELFUN, welches auf Elemente aus Δ also auf Mengen von Termen angewandt wird. Die gewünschte abstrakte Unifikation liefert Definition 3.3:

---

[45] "$\Rightarrow$" beschreibt eine mögliche Instantiierung der Variablen zur Programmausführung.

Def 3.3 *Abstrakte Unifikation* (von Termmengen)

Sind T_1 und T_2 zwei Mengen von Termen, so ist $s\_unify(T_1, T_2)$ durch die kleinste Menge T gegeben, für die gilt, daß für jedes Paar $t_1 \in T_1$ und $t_2 \in T_2$ mit $\sigma = mgu(t_1, t_2)$ folgt, daß $\sigma(t_1) \in T$ gilt.

Im Wesen der herkömmlichen Unifikation liegt, daß wenn t durch die Unifikation zweier Terme t_1 und t_2 entsteht, der Term t eine Instanz von beiden Termen t_1 und t_2 ist. Die Variablen in t sind somit höchstens mehr gebunden als die Variablen in t_1 und t_2. Diese Eigenschaft muß sich auch in der abstrakten Unifikation widerspiegeln, so daß beispielsweise für $\delta_1 = f$ und $\delta_2 = c$ folgt: s_unify(δ_1,δ_2) = c. Das ist sicherlich einleuchtend, denn δ_2 beschreibt alle Grundterme und δ_1 alle freien Variablen und durch die Unifikation von zwei Termen der jeweiligen Termklasse erhält man wieder einen Grundterm. Für die Domäne Δ läßt sich die abstrakte Unifikation durch folgende Tabelle beschreiben:

s-unify	*f*	*d*	*c*	*t*
f	*f*	*d*	*c*	*t*
d	*d*	*d*	*c*	*t*
c	*c*	*c*	*c*	*t*
t	*t*	*t*	*t*	*t*

Bild 3.2 Wertetabelle der Operation s-unify(δ, δ')

Man erkennt, daß durch *s-unify* eine Partialordnung $\leq_s$ auf Δ definiert ist, mit $\delta \leq_s \delta'$, falls s-unify(δ, δ') = δ. Wir erhalten somit die Subsumptionsordnung (Δ, $\leq_S$), mit: $t <_s c <_s d <_s f$.

Bevor wir formal die abstrakte Interpretation definieren können, brauchen wir noch Funktionen auf der abstrakten Ebene, die den Abarbeitungsmechanismus in RELFUN modellieren. Hierfür betrachten wir zuerst eine Funktion, die die aktuelle Instantiierung eines Literals beschreibt. Wir benutzen die Schreibweise (p T) für Literale der Form (p t_1 ... t_n), so daß T ein Tupel <t_1 ... t_n> von Termen bezeichnet und die Selektion des Terms t_k mittels T/k notiert wird. Weiterhin ist die Projektion des i-states auf ein Tupel von Termen folgendermaßen definiert:

Def 3.4 *i-state Projektion*

Sei T ein n-Tupel von Termen und ξ ein Instantiierungsstatus (i-state); dann ist die Projektion π_T von ξ auf T definiert durch:

$$\pi_T(\xi) := <\xi(t_1) \ldots \xi(t_n)>$$

Durch die Projektion des i-states berechnet man gerade das Aufrufpattern für das Literal p zu einem gewissen Instantiierungsstatus. Zu einem Literal mit Aufrufpattern gehört ein entsprechendes Erfolgspattern succp. Anhand des succp muß nun der neue i-state berechnet werden, wobei die Idee ist: Hat (p T) das Aufrufpattern $\pi_T(\xi) = <\xi(t_1), \ldots, \xi(t_n)>$ und ist das Erfolgspattern durch $I := <\delta_1, \ldots, \delta_n, \delta_{n+1}>$ gegeben und gilt T/k = v, so berechnet sich $\zeta_{neu}(v) :=$ s_unify($\xi(v)$,δ_k). Wir haben somit eine Beschreibungsmöglichkeit für das Fortschreiten der Instantiierungen bekommen. Allerdings müssen wir noch klären, was passiert, wenn die Variable _v mehrmals in T vorkommt. Dazu betrachten wir alle Positionen k, an denen die Variable vorkommt und bilden die Menge M der s_unifier, so daß sich $\zeta_{neu}(v)$ aus dem Minimum $\min_{\leq_s}$ von M bezüglich der Subsumptionsordnung (Δ, $\leq_s$) ergibt. Diese Vorgehensweise wird *i-state Transformation* genannt und in Def 3.5 formal eingeführt.

Def 3.5 *i-state Transformation*

Sei T ein n-Tupel von Termen, ξ ein i-state und I ein i-pattern ($I \in \Delta^{n+1}$), dann bezeichnet ϱ die i-state Transformation und ist gegeben durch:

$$\varrho(T,I,\xi)(v) = \min_{\leq}(\{s\_unify(\xi(v),I/k) \mid T/k = v\}) \quad \text{, falls v in T vorkommt}$$
$$\varrho(T,I,\xi)(v) = \xi(v) \quad \text{, sonst}$$

Die abstrakte Interpretation läßt sich nun so beschreiben, daß zu einem Prädikat p das Aufrufpattern bekannt ist und anhand der Klauseln, die p definieren, das Erfolgspattern berechnet wird. Hierbei werden weitere Prädikate aufgerufen, deren Aufrufpattern sich aus dem i-state und der i-state Projektion ergeben und deren Erfolgspattern auf dieselbe Art und Weise wie oben berechnet werden. Ist nun ein Aufrufpattern gegeben, stellt sich die Frage, ob das Paar <Aufrufpattern, Erfolgspattern> ein korrektes Verhalten des Programms bzw. Prädikats beschreibt. Falls dies der Fall ist, sprechen wir von einem zulässigen Erfolgspattern (relativ zu dem Aufrufpattern). Genauer definieren wir:

Def 3.6 *Zulässige Erfolgspattern*

Sei das Aufrufpattern I_c und das Erfolgspattern I_s für ein Prädikat gegeben. Das Tupel $<I_c, I_s>$ liegt in succpat(p) (ist ein zulässiges Erfolgspattern), genau dann wenn eine Klausel

$(ct\ (p\ T_0)\ (q_1\ T_1) \ldots (q_n\ T_n))$ $n \geq 0$ falls ct 0 hn bzw. $n \geq 1$ falls ct = ft

in der Datenbasis existiert, so daß $I_s = \pi_{T0}(\xi_n)+val$[46], wobei ξ_{init} der initiale i-state der Klausel ist und $\xi_0 = \varrho(T_0, I_c, \xi_{init})$; $\xi_j = \varrho(T_j, I_j, \xi_{j-1})$ $1 \leq j \leq n$ und $<\pi_{Tj}(\xi_{j-1}), I_j> \in$ succpat(q_j) gilt und $val = t$, falls ct = hn bzw. $val = I_{k+1}$ mit $k = |T_n|$ sonst.

Das succpat für ein Prädikat p wird also berechnet, indem die succpats der Prämissen berechnet werden. In Def 3.6 erkennt man die Vorgehensweise des normalen RELFUN-Interpreters wieder, der ebenfalls eine Klausel von links nach rechts abarbeitet. Um die eingeführten Begriffe zu verdeutlichen betrachten wir ein einfaches Beispiel:

```
(ft (p _u _v) (q _u _w) (r _w _v) )
(hn (q a _v) )
(hn (r b b) )
```

Für die ft-Klausel kann $T_0 = <\_u, \_v>$, $T_1 = <\_u, \_w>$, $T_2 = <\_w, \_v>$ festgehalten werden. Angenommen, das Aufrufpattern für p/2 ist mit $I_c = <d, d>$ vorgegeben, dann erhalten wir in einem ersten Schritt, unter Berücksichtigung von $\xi_{init} = \{\_u->f, \_v->f, \_w->f\}$, den i-state $\xi_0 = \varrho(T_0, I_c, \xi_{init}) = \{\_u->d, \_v->d, \_w->f\}$. Durch i-state Projektion erhalten wir das Aufrufpattern für q/2: $\pi_{T1}(\xi_0) = <d, f>$. Da q/2 nur aus einer hn-Klausel besteht, läßt sich einfach $<<d, f>, <c, f, t>> \in$ succpat(q/2) bzw $I_1 = <c, f, t>$ berechnen und somit erhält man $\xi_1 = \delta(T_1, I_1, \xi 0) = \{\_u->c, \_v->d, \_w->f\}$. Für das nächste goal (r _w _v) erhält man entsprechend: $\pi_{T2}(\xi_1) = <f, d>$, $<<f, d>, <c, c, t>> \in$ succpat(r/2), $I_2 = <c, c, t>$, $\xi_2 = \{\_u->c, \_v->c, \_w->c\}$ und schließlich $I_s = \pi_{T0}(\xi_2)+val = <c, c, t>$, wobei $val = I_2/3$ gilt. Zusammenfassend kann für das Programm mit der approximierten Anfrage $<d, d>$ für p/2 gesagt werden, daß folgende mode-Informationen gelten:

p/2	calling: $<d, d>$	returning: $<c, c, t>$
q/2	calling: $<d, f>$	returning: $<c, f, t>$
r/2	calling: $<f, d>$	returning: $<c, c, t>$

[46] "+" bedeutet in diesem Zusammenhang, daß das n-Tupel um *val* auf ein n+1-Tupel erweitert wird.

Denotative Literale t werden wie nullstellige Prädikate betrachtet, deren succpat sich für einen i-state ξ sofort zu $\{<<>, \xi(t)>\}$ ergibt. Für das is-Literal gilt bezüglich der i-state Transformation eine Besonderheit. Betrachten wir dazu das Klauselfragment

(... (is t (q T)) ...)

und nehmen an, daß vor dem is-Literal der i-state ξ gilt. Dann wird zuerst so getan, als ob nur das goal (q T) vorhanden wäre und $<\pi_T(\xi), I_S> \in$ succpat(q) bzw. ξ_{neu} berechnet. Ist t eine Variable, so wird anschließend mittels ζ'_{neu} = s_unify($\xi_{neu}(t)$,I_S/k) der i-state korrigiert und $I_S/k = \xi'_{neu}(t)$ gesetzt, wobei $k = |T|+1$ gilt. Diese Vorgehensweise modelliert die Semantik des is-Literals in RELFUN, wo zuerst die rechte Seite ausgewertet und dann der Wert mit der linken Seite unifiziert wird. Die linke Seite denotiert dann im Erfolgsfall den Wert des is-Literals (siehe auch Abschnitt 1.2 und 2.1).

Schon anfangs von Abschnitt 3 wurde erwähnt, daß die automatische mode-Generierung von einer Approximation der möglichen Anfragen an die Datenbasis ausgeht. Darauf aufbauend werden dann mögliche Aufrufe von weiteren Prädikaten bestimmt. Bei einem tatsächlichen Programmablauf müssen Aufrufe dieser Art nicht zwangsläufig erfolgen. Es handelt sich also um eine konservative Abschätzung, die aber trotzdem den Sinn hat, einige Aufrufmuster auszuschließen. Mit anderen Worten: Nicht alle möglichen Aufrufpattern sollen *zulässige Aufrufpattern* sein. Die Menge der zulässigen Aufrufpattern für ein Prädikat p wird mit callpat(p) bezeichnet und ist nachfolgend definiert:

Def 3.7 *Zulässige Aufrufpattern*

Sei die Menge Q von Anfrageapproximationen vorgegeben; dann ist die Menge callpat(p) der zulässigen Aufrufpattern für ein Prädikat p die kleinste Menge, so daß gilt:

(1) Ist I ein Aufrufpattern für p, welches in Q vorkommt, so gilt $I \in$ callpat(p).

(2) Ist q_0 ein Prädikat in der Datenbasis und $I_c \in$ callpat(q_0) und existiert eine Klausel

$$(ct\ (q_0\ T_0)\ (q_1\ T_1)\ \ldots\ (q_n\ T_n)),$$

mit $\xi_0 = \delta(T_0, I_c, \xi_{init})$ und ξ_i ist der i-state unmittelbar nach dem Literal $(q_i\ T_i)$, dann ist $cp_i = \pi_{T_i}(\xi_{i-1})$ in callpat(q_i). Die ξ_i berechnen sich durch $\xi_i = \delta(T_i, sp_i, \xi_{i-1})$ mit $<cp_i, sp_i> \in$ succpat(q_i).

In [Deb86] wurde gezeigt, daß ein mode-Interpreter über der eingeschränkten Domäne $\Delta' = \{d, c, e\}$, welcher die zulässigen Aufrufpattern nach Def 3.7 inferiert, korrekt ist.[47] Für eine gegebene Menge Q von Anfrageapproximationen und eine echte (mit Q kompatible) Anfrage an die Datenbasis, ist also jedes während der Abarbeitung aufgerufene goal $(q\ t_1\ \ldots\ t_n)$ kompatibel mit einem Aufrufmuster $I_c \in$ callpat(q). Die Kompatibilität eines Aufrufs mit einem Aufrufpattern wurde bereits anfangs von Abschnitt 3 erläutert.

Für die Domäne Δ gilt dies im allgemeinen nicht. In einigen Fällen macht der mode *f* Schwierigkeiten, wie beispielsweise nachstehende kleine Datenbasis zeigt.

```
(ft (p _u _v) (q _u _v) (r _u) (s _v) )
(hn (q _z _z))
(hn (r a))
(hn (s _w))
```

Bild 3.3 Fehlerhafte *zulässige Aufrufpattern* durch *aliasing*

Setzen wir $Q = \{ <f, f> (\in$ callpat(p))$\}$ voraus, ergibt sich nach Def 3.6 succpat(q) = $\{<<f, f>, <f, f, t>>\}$ sowie succpat(r) = $\{<<f>, <c, t>>\}$. Demnach gilt unmittelbar vor dem Literal (s _v) der i-state ξ_2 = {_u->*c*, _v->*f*}. Das bedeutet aber, daß als einziges zulässiges Aufrufpattern für s/1 das i-pattern <f> berechnet wird. Zur Laufzeit wird aber s/1 immer mit der Konstante a aufgerufen, da das goal (q _u _v) die Variablen _u und _v auf dasselbe Objekt _z zeigen läßt und das goal (r _u) die Variable _u = _z (und damit auch _v = _z!) an die Konstanten a bindet. Der Aufruf (s _v) erfolgt schließlich mit der an a gebundenen Variablen

[47] Debrays Arbeit behandelt PROLOG, die Ergebnisse sind aber ohne weiteres auf RELFUN übertragbar.

und nicht wie inferiert mit einer freien Variablen. Das goal (s _v) wird also zur Laufzeit mit einer zu callpat(s) nicht kompatiblen Instantiierung aufgerufen und ist somit nicht korrekt bestimmt worden.

Den Effekt, daß verschiedene Variablen durch Unifikation identifiziert werden, nennen wir *aliasing*. Die Problematik ergibt sich, weil in dem Instantiierungsstatus keine Information über aliasing von Argumenten vorhanden ist. Um trotzdem korrekte Aufrufpattern zu berechnen, kann man Klauseln mit einem *alias-Bit* kennzeichnen, wenn sie potentiell die Gefahr des aliasing in sich bergen. Falls für eine Klausel das alias-Bit = 1 ist, wird dann etwas anders als in Def 3.6 und Def 3.7 verfahren (s. u.).

Das *alias-Problem* läßt sich in die Teilprobleme des *call-aliasing* und *return-aliasing* gliedern. Call-aliasing liegt vor, wenn in einem goal Variablen mehrmals vorkommen und in den zugehörigen Prozedurklauseln *Kopfvariablen* mehrmals in der Klausel vorkommen (siehe z. B. Bild 3.4). Unter return-aliasing fällt das *sharen* von verschiedenen Variablen aufgrund eines Aufrufs. Das vorherige Beispiel zeigte eine Instanz dieser Problemklasse. Folgendes Beispiel illustriert das Problem des call-aliasing.

```
(ft (p) (q _v _v) )
(ft (q a _w) (r _w))
```

Bild 3.4 Problem des *call- aliasing*

Ausgehend von dem Aufruf (p) wird hier zwar korrekt callpat(q) = {<*f*, *f*>} bestimmt, doch für r/1 erhalten wir fehlerhaft callpat(r) = {<*f*>}, da nicht berücksichtigt wird, daß die Variable _w aufgrund des Aufrufs (q _v _v) an die Konstante a gebunden ist und somit der Aufruf von r/1 mit einem grundinstantiierten Argument erfolgt, was sicherlich nicht kompatibel mit dem gefundenem zulässigen Aufrufpattern von r/1 ist.

Es wurden schon unter verschiedenen Gesichtspunkten Algorithmen zur Erkennung von aliasing entworfen (siehe etwa [Bru86], [Cha85] und [Deb86]), die auf einer komplexen statischen Programmanalyse beruhen. In dieser Arbeit wird ein relativ einfaches Verfahren vorgestellt, das die Gefahr des aliasing nur konservativ behandeln kann. Im wesentlich wird zu jedem Prädikat ein alias-Bit bestimmt, das mögliches aliasing (alias-Bit = 1) kennzeichnet. Für Prädikate, die Klauseln besitzen, in denen eine Kopfvariable mehrmals in der Klausel vorkommt, wird das alias-Bit = 1 gesetzt, denn hier besteht potentiell die Gefahr des call-aliasing. Dann werden Prädikate p mit alias-Bit(p) = 0 getestet, ob bei ihnen ein return-aliasing auftreten kann, weil sie Prädikate mit gesetztem alias-Bit aufrufen. Der Algorithmus hat somit folgende Form:

Algorithmus: **SET-ALI-BIT**

Eingabe: db ; Die Datenbasis.
Ausgabe: alias-Bit ; Information über alle Prädikate, ob ihr alias-Bit gesetzt ist oder nicht.

1 Für alle Prädikate p aus db

1.1 Falls eine Klausel in p existiert, in der Kopfvariablen mehrmals in der Klausel vorkommen, so setze alias-Bit(p) := 1, andernfalls setze alias-Bit(p) := 0.

2 Solange es noch Änderungen bei den alias-Bits gibt

2.1 Für alle Prädikate p aus db mit alias-Bit(p) = 0 tue

2.1.1 Falls eine Klausel aus p ein Literal q/n als Prämisse besitzt und alias-Bit(q/n) = 1, dann setze alias-Bit(p) = 1.

3 Stop.

Ändert man Def 3.6 und 3.7 derart, daß für Klauseln, die zu einem Prädikat p mit alias-Bit(p) = 1 gehören, der initiale Instantiierungsstatus $\xi_{init}(v) = d$ (anstatt $\xi_{init}(v) = f$) für Kopfvariablen v gesetzt wird, so betrachtet man in den "gefährlichen" Fällen automatisch nur die eingeschränkte Domäne $\Delta' = \Delta \backslash \{f\}$, für die die zulässigen Aufrufpattern korrekt sind. Die freien Variablen läßt man nur für die unproblematischen (sicheren) Fällen zu.

Ist ein Prädikat q durch m Klauseln definiert und betrachtet man ein Literal (q T) mit einem i-state ξ, so können prinzipiell m Paare $<\pi_T(\xi_i), I_{s_i}> \in$ succpat(q) $1 \le i \le m$ nach Def 3.6 berechnet werden. Werden alle diese Paare für die weitere Berechnung beachtet, kommt es schnell zu einer explosionsartigen Ausbreitung der zu untersuchenden Aufrufpattern für die noch folgenden Literale in der Klausel. Es ist also sinnvoll, die Vielfalt vernünftig einzugrenzen. Die Menge der zulässigen Erfolgspattern kann dahingehend eingeschränkt werden, daß statt der Menge $\{<I_c, I_1>, \ldots, <I_c, I_m>\}$ nur das Paar $<I_c, I_s>$ mit $I_s = LUB\{I_i \mid 1 \le i \le m\}$ betrachtet wird. LUB ist die natürliche Erweiterung des *least upper bound* (*lub* (oder auch "$\sqcup$") bzgl. $(\Delta, \sqsubseteq)$ s. a. Bild 3.1) auf i-pattern, so daß gilt, falls $I_i = <\delta_{i1}, \ldots, \delta_{in+1}>$ und $I_s = <\delta_{s1}, \ldots, \delta_{sn+1}>$, dann $\delta_{sk} = \bigsqcup \{\delta_{ik} \mid 1 \le i \le m\}$. Offensichtlich bleibt die so inferierte Menge der zulässigen Aufrufpattern weiterhin korrekt, da die Menge der zulässigen Erfolgspattern durch ihre kleinste obere Schranke ersetzt worden ist. Es handelt sich also um eine weniger gehaltvolle Information und demnach um keine Restriktion, sondern um eine konservativere Abschätzung.

Hat man für ein Prädikat q, die Menge der zulässigen Aufrufpattern callpat(q) bestimmt, so existiert zu jedem $I_c \in$ callpat(q) genau ein Erfolgspattern I_s. Die Paare $<I_c, I_s>$ beschreiben nun, nach Aufrufpattern differenziert, die möglichen Instantiierungen des Prädikats q. Der mode von q ergibt sich durch $LUB\{I_c \mid I_c \in callpat(q)\}$. Der im nächsten Abschnitt beschriebene Algorithmus MODE-INTERPRET berechnet die Paare $<I_c, I_s>$ mit $I_c \in$ callpat(q) für die Prädikate q einer Datenbasis. Der erste Teil des Abschnitts bezieht sich noch sehr stark auf diesen gerade vorgestellten mehr theoretischen Teil, bevor dann im zweiten Teil auf die

LISP-Implementierung eingegangen wird. Es ist also möglich, zumindest den Anfang von Abschnitt 3.3 als Ergänzung zum Abschnitt 3.2 zu sehen.

3.3 Implementierungshinweise

Für die folgenden Überlegungen und vorgestellten Algorithmen (im wesentlichen der mode-Interpreter) wird davon ausgegangen, daß weiterhin folgende Einschränkungen gelten:[48]

(1) Die RELFUN-Datenbasis besteht nur aus KERN-Klauseln gemäß Abschnitt 2.

(2) Die RELFUN-Datenbasis darf nicht dynamisch modifiziert werden. Insbesondere sind weder dynamisches *assert* noch *retract* erlaubt.

(3) Die Menge Q (Approximation der möglichen Anfragen) besteht aus genau einem Aufrufpattern I_c für genau ein Prädikat q aus der Datenbasis. ($Q = \{(q\ I_c)\}$

Über die Einschränkung (1) brauchen wir nicht weiter reden, sonst wäre der Abschnitt 2 ziemlich witzlos. Die Einschränkung (2) ist notwendig, da die Ergebnisse aus Abschnitt 3.2 nur für statische Datenbasen gilt. Man hat die Vorstellung eines Moduls M, das von einer anderen Datenbasis DB nachgeladen wird. In DB werden die Prozeduren nicht in der vollen Allgemeinheit aufgerufen, sondern z. B nur mit gewissen Aufrufpattern. Entsprechend kann das Modul M für die Aufrufpattern speziell compiliert werden. Die Einschränkung (3) ist keine echte Einschränkung, sondern vereinfacht nur die Schnittstelle des mode-Interpreters. Soll eine beliebige endliche Menge von Anfragen bearbeiten werden etwa $Q' = \{(p_1\ I_1), \ldots, (p_n\ I_n)\}$, kann die Anfrage (md-query I_q) mit $Iq = <\delta_{1_1}, \ldots, \delta_{1_{m_1}}, \delta_{2_1}, \ldots, \delta_{n_{m_n}}>$ mit $I_i = <\delta_{i_1}, \ldots, \delta_{i_{m_i}}>$ generiert werden. Zusätzlich muß die *Pseudoklausel* [49]

$$(\text{hn (md-query } \_v1_1 \ldots \_vn_{m_n})\ (p_1\ \_v1_1 \ldots \_v1_{m_1}) \ldots\ (p_n\ \_vn_1 \ldots \_vn_{m_n})\ )$$

kurzzeitig zur Datenbasis hinzugenommen werden. Die Abarbeitung des Aufrufs (md-query I_q) hat dann den gewünschten Effekt. Die Anzahl der Variablen lassen sich noch verringern, wenn man die modes *d*, *c* und *t* über eine Variable shared. Möchte man beispielsweise die modes der Prozeduren einer Datenbasis relativ zu {(p *c d*), (q *c c*)} berechnen, so erzeugt man die Pseudoklausel (hn (md-query _v1 _v2) (p _v1 _v2) (q _v1 _v1)) und den Aufruf (md-query *c d*). Der mode *f* muß vorsichtiger behandelt werden, damit nicht *aliasing-Effekte* entstehen.

---

[48] Ohne besonders darauf hingewiesen worden ist, galten diese Bedingungen auch schon im Abschnitt 3.2.

[49] Ein *Pseudoklausel* ist eine Klausel, die kurzzeitig, für den Benutzter unsichtbar, zur Datenbasis hinzugefügt wird.

Die restliche Abschnittsgliederung beinhaltet in Abschnitt 3.3.1 die Architekturbeschreibung des mode-Interpreters. In Abschnitt 3.3.2 wird speziell auf die LISP-Implementierung eingegangen und in Abschnitt 3.3.3 wird die Schnittstelle zum RFM-System beschrieben.

3.3.1 Architekturbeschreibung des mode-Interpreters

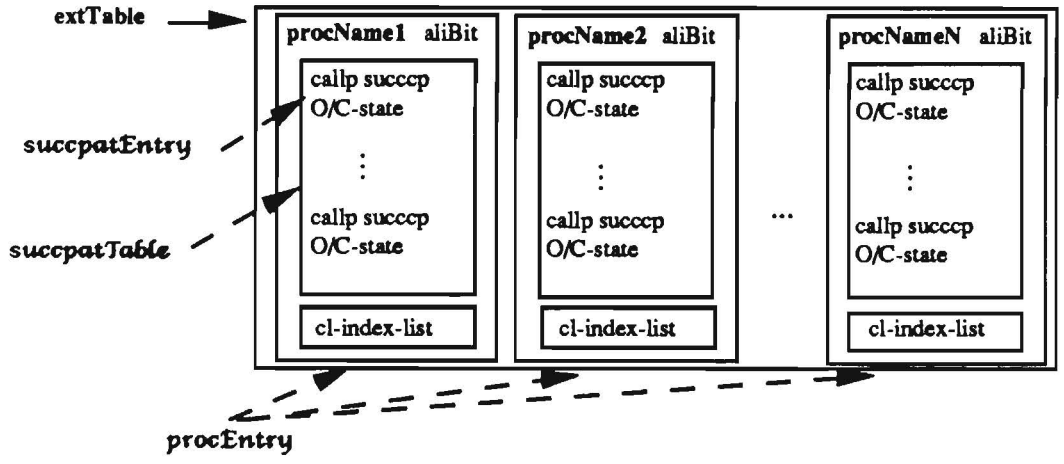


Bild 3.5 Zentrale Datenstruktur *extension table* (extTable) des mode-Interpreters

Der mode-Interpreter nutzt intensiv die Datenstruktur *extension table* (Bild 3.5). Sie ist in Prozedureinträgen (*procEntry*) gegliedert. Ein Prozedureintrag wird anhand des Prozedurnamen unterschieden. Besteht für eine Prozedur die Gefahr des *aliasing*, wird dies im *aliBit* gekennzeichnet. Die Klauselindexliste *cl-index-list* enthält Verweise auf die Klauseln, die die Prozedur definieren. Der wesentliche Eintrag im *procEntry* ist der *succpatTable*. In ihm sind die inferierten Aufrufpattern mit zugehörigem Erfolgspattern notiert. Weiterhin existiert zu jedem Paar <Aufrufpattern, Erfolgspattern> ein Open/Close-Zustand *o/c-state*. Zusammen bilden sie einen *succpatEntry*. Der *o/c-state* besagt, ob zu einem Aufrufpattern bereits alle Klauseln der Prozedur betrachtet worden sind. Ist dies der Fall, so ist der Status *close* ansonsten *open* oder nicht definiert (s. u.). Erfolgt ein *abstrakter Aufruf* einer Prozedur p mit einem Aufrufpattern I_c, werden drei Fälle unterschieden:

(1) *o/c-state* = *undef*: Dies ist immer beim ersten Aufruf von p mit I_c der Fall. Es wird dann ein *succpatEntry* $<I_c\ succp_{min}\ open>$ in die entsprechende succpatTable eingetragen. Gilt $|I_c| = n$, so steht $succp_{min}$ für $<\delta_1,\ldots,\delta_{n+1}>$ mit δ_i = *empty*. Im weiteren werden die Erfolgspattern für diesen abstrakten Aufruf berechnet und mittels der LUB-Bildung (siehe Abschnitt 3.2) die Einträge im *succpatEntry* aktualisiert.

(2) *o/c-state = close*: Das Ergebnis des abstrakten Aufrufs (p I_c) wurde schon vorher berechnet und ist bereits im entsprechenden *succpatEntry* vermerkt. Das Ergebnis kann also direkt von dort übernommen werden und braucht nicht erneut berechnet werden.

(3) *o/c-state = open*: Dies entspricht einem rekursiven Aufruf von (p I_c). Es wird nun aufgrund der Art der Rekursion (s. u.) und dem bisher ermittelten Erfolgspattern eine sichere Approximation des Erfolgspattern in den *succpatEntry* eingetragen und der Open/Close-Zustand auf *close* gesetzt.

Die Fälle (1) und (2) sind ohne weiteres verständlich. Bei Fallunterscheidung (3) stellen sich die zwei Fragen: a.) Warum wird o/c-state = close gesetzt und somit die Berechnung für diesen Aufruf abgeschlossen (obwohl womöglich noch nicht alle Klauseln der Prozedur p betrachtet worden sind)? und b.) Wie wird das abschließende Erfolgpattern im succpatEntry berechnet?. Betrachten wir hierzu einen möglichen Rekursionsfall:

(ft (p T) ... (p T') (q T'') ...)

Angenommen die Klausel wurde mit dem Aufrufpattern I_c aufgerufen und unmittelbar vor der Prämisse (p T') gilt der Instantiierungsstatus ξ, so daß weiterhin $I_p = \pi_{T'}(\xi) = I_c$ gilt. Somit erfolgt also ein rekursiver abstrakter Aufruf (p I_c). In der *extension table* ist dies anhand von o/c-state = open im entsprechenden succpatEntry erkennbar. Das Problem ist nun, daß das Erfolgspattern succp auch von der Prämisse (q T'') abhängt. Der Aufruf (q I_q) ergibt sich aus der i-state Projektion $I_q = \pi_{T''}(\xi')$, allerdings hängt der i-state ξ' wiederum von dem Erfolgpattern des Aufrufs (p I_c) ab. Man erkennt, daß hier schnell die Gefahr einer nicht-terminierenden Rekursion entsteht. Dies beantwortet Frage a.). Das Problem wird umgangen, indem eine konservative Abschätzung des rekursiven Aufrufs mittels der Funktion *conserv* erfolgt, so daß succp ::= conserv(I_c) gilt. Die Abbildung conserv ist auf Aufrufpattern I folgendermaßen erklärt:

conserv(I) ::= $\langle \delta_1, \ldots, \delta_n, d \rangle$ mit δ_i := s_unify(I/i, *d*) , $1 \le i \le |I|$

Die konservative Abschätzung für succp ergibt sich also aus dem Aufrufpattern I_c, indem dort die modes *f* in *d* geändert werden und als Wert des Aufruf ebenfalls *d* angenommen wird. Für spezielle Rekursionen ist es möglich, schärfere Abschätzungen anzugeben. Hierfür bezeichnen wir eine rekursive Prozedur, die nur in einer Klausel rekursiv definiert ist und in der die Rekursion nur im letzten Aufruf vorkommt, als *sichere Rekursion*. Wir sagen auch, die Prozedur ist *safe-rec* (bzw. *unsafe-rec*, wenn die Prozedur nicht *safe-rec* ist). Beispielsweise ist die rekursive Prozedur p/2 *unsafe-rec*, wenn sie wie folgt definiert ist:

```
(hn (p a b))
(ft  (p _v _w) (p a _v) (p _x _w))
```

Sie ist zwar nur in einer nämlich der zweiten Klausel rekursiv definiert und erfüllt damit den ersten Teil der Bedingungen für *safe-rec*, jedoch erfolgt ein rekursiver Aufruf nicht nur im letzten Literal, sondern auch im vorletzten. Zu den Prozeduren die *unsafe-rec* sind, zählen auch diejenigen, die indirekte Rekursion enthalten. Das folgende Beispiel zeigt die rekursive Prozedur foo/2, die *safe-rec* ist.

```
(hn (foo a b))
(hn (foo c d))
(ft (foo _x _w) (foo _w _x) )
```

Hier ist ausschließlich die ft-Klausel rekursiv definiert und der rekursive Aufruf erfolgt nur im letzten Literal der Klausel. Es erfüllt somit die geforderten Bedingung für eine sichere Rekursion. Ist ein Aufrufpattern I_C gegeben, so daß unmittelbar vor der Prämisse (foo _v _w) der i-state ξ mit $\pi_T(\xi) = I_C$ (mit T = <_v, _w>) gilt, dann existiert auch ein succpatEntry $<I_C$, succp, open>. Mit succp = I_S+val können die zwei Fälle $I_S = I_C$ und $I_S \neq I_C$ unterschieden werden. Der Fall $I_S = I_C$ bedeutet, daß der Instantiierungsstatus sich bei der Abarbeitung der Klausel nicht mehr geändert hat. Der (sichere) rekursive Aufruf (foo I_C) würde also ebenfalls keine Änderungen bewirken, da er sich vom vorherigen nicht unterscheidet. Für diesen Fall kann demnach die Abarbeitung des Aufrufs direkt beendet und der succpatEntry unter Beibehaltung von succp geschlossen werden.

Für $I_S \neq I_C$ (=> I_S kleiner als I_C bezüglich $(\Delta^*, \leq_s)$[50]) ist das bisherige Erfolgspattern succp noch nicht korrekt, sondern es muß noch das Erfolgspattern des Aufrufs (foo I_S) mitberücksichtigt werden, denn I_S ist ja gerade die Approximation der Instantiierungen von foo/2, die bei einem rekursiven Aufruf in der ersten Rekursionstiefe entstehen können, wenn man von (foo I_C) ausgeht. Die so entstehende Kette der foo/2 Aufrufe (Rekursion auf auf der abstrakten Ebene!) kann nicht unendlich lange Laufen, da $(\Delta^*, \leq_s)$ wohlfundiert ist. Somit ist auch Frage b.) beantwortet und es kann die Funktion **MD-INTERP-GOAL** vorgestellt, die die Abarbeitung eines abstrakten Aufrufs (p I_C) im mode-Interpreter definiert.

Funktion **MD-INTERP-GOAL**

Eingabe: goal = (p I_C) ; Gegeben durch Prozedurnamen und Aufrufpattern.

Ausgabe: succp = (I_S+val) ; Das Erfolgspattern.

Variablen: extTable ; Globale Variable.

---

[50] Δ^* steht für Δ^n $n \geq 1$. Die Subsumptionsordnung $\leq_s$ ist in Abschnitt 3.2 erklärt.

callp	; Das Aufrufpattern von goal.
o/c-state	; Der Open/Close-Zustand des *succpatEntry* zu goal.
succp', I_c'	; Hilfspattern für die Berechnung von succp.
safe-rec	; Flag, das eine *sichere Rekursion* anzeigt.

Bemerkung: Die extTable (*extension table*) wird per Seiteneffekt verändert, falls das goal nicht schon vorher berechnet wurde (o/c-state = close).

1 Hole aus der extTable den bisherigen succp und o/c-state des *succpatEntry* zu goal.

2 Falls o/c-state = close, dann gebe als Ergebnis succp zurück.

3 Falls o/c-state = undef, dann öffne ein *succpatEntry* zu goal. Für alle Klauseln C der Prozedur p rufe in der Reihenfolge gemäß der Klauselindexliste die Funktion **MD-INTERP-CLAUSE**(goal, C) auf.
Ermittle den Wert succp aus der extTable neu und schließe den *succpatEntry* und gebe als Ergebnis succp zurück.

4 Falls o/c-state = open, dann

4.1 Falls safe-rec = FALSE, dann setze succp' := conserv(I_c).

4.2 Falls safe-rec = TRUE und $I_s \neq I_c$, dann setze succp' := **MD-INTERP-GOAL**(p, I_s), sonst succp' := succp.

4.3 Schließe den *succpatEntry* zu goal mit neuem Erfolgspattern succp'. Gebe als Ergebnis succp' zurück.

In Anweisung 2 erkennt man die eigentliche Arbeitsweise von **MD-INTERP-GOAL**, die darin besteht, nacheinander die Klauseln der aufgerufenden Prozedur abzuarbeiten. Hierbei ist die Klauselindexliste so geordnet, daß Verweise auf rekursive Klauseln am Ende der Klauselindexliste stehen. Dies gestattet einen einfachen Test (im Hinblick auf die LISP-Implementierung) auf *safe-rec*. Eine einzelne Klausel wird durch die nachstehend definierte Funktion **MD-INTERP-CLAUSE** abgearbeitet.

Funktion **MD-INTERP-CLAUSE**

Eingabe:	goal = (p I_c)	; Gegeben durch Prozedurnamen und Aufrufpattern.
	C = (*ct* (p T_0) (q_1 T_1) ... (q_n T_n))	; Die zu bearbeitende Klausel.
Ausgabe:	Keine	
Variablen:	extTable	; Globale Variable.
	I_s+val	; Erfolgspattern der abstrakten Aufrufe der Prämissen.
	succp	; Das bisher berechnete Erfolgspattern aus der *succpatEntry* zu goal.
	succp'	; Hilfspattern zur neuen Berechnung von succp.

Bemerkung: Gemäß Def 3.3 wird eine i-state Transformation durchgeführt. Aus dem final-state (i-state nach dem goal (q_n T_n)) wird das Erfolgspattern berechnet und per Seiteneffekt der *succpatEntry* gemäß der LUB-Bildung aktualisiert.

1 Berechne aus I_c, T_0 und dem *aliBit* zur Prozedur p den Instantiierungsstatus ξ_0. Setze I_s+val = $\pi_{T_0}(\xi_0)$+*f*

2 Von i = 1 bis zur Anzahl der Prämissen in C tue

2.1 Setze I_s+val := **MD-INTERP-GOAL**($(q_i, \pi_{T_i}(\xi_{i-1}))$). Führe die i-state Transformation $\xi_i = \delta(T_i, I_S, \xi_{i-1})$ unter Beachtung der aliasing Information durch.

3 Falls *ct* = hn, dann setze val = *t*.

4 Hole das succp aus dem *succpatEntry* zu goal. Bilde succp':= LUB{succp, I_s+val} und aktualisiere den *succpatEntry* (bzw. die extTable) entsprechend.

Mit Hilfe der obigen Funktion kann nun der mode-Interpreter auf einfache Weise beschrieben werden.

Algorithmus: **MODE-INTERPRET**

Eingabe: query = (p I_C) ; Die Anfrage Approximation.
db ; Die Datenbasis (Menge von Prozeduren).

Ausgabe: extTable ; Die mode-Informationen in der *extension table* zusammengefaßt.

Bemerkung: Die extTable wird globalisiert und somit den obigen Funktionen für Seiteneffekte bekanntgemacht.

1 Initialisiere extTable. Berechne hierbei die Klauselindexliste und das aliBit der Prozeduren.

2 Rufe **MD-INTERP-GOAL**(query) auf.

3 Gebe die extTable als Wert zurück.

3.3.2 Beschreibung der LISP-Implementierung

Die Implementierung erfolgte in einem streng funktionalen (*puren*) LISP-Programmierstil (siehe auch Abschnitt 2.3). Im wesentlichen wurden die Ideen und die Algorithmen (Funktionen) aus Abschnitt 3.2 und 3.3.1 implementiert. Die *aliasing*-Information wird zwar nach dem Algorithmus **SET-ALIBIT** aus Abschnitt 3.2 berechnet und anhand des *aliBits* in der extension table repräsentiert, doch für die Praxis ist dieser Algorithmus zu konservativ, da es in RELFUN- (bzw. PROLOG-) Programmen häufig der Fall ist, daß Variablen, die im Kopfliteral vorkommen, auch in den Prämissen auftreten. Für sie besteht also die Gefahr des *call-aliasing* und entsprechend würde hier immer das aliBit = 1 sein. In der vorliegenden LISP-Implementierung wird an dieser Stelle genauer differenziert, so daß call-aliasing für einzelne Klauseln anstatt für ganze Prozeduren betrachtet wird. D. h. call-aliasing wird nur dann beachtet, wenn in dem aufrufendem Literal eine Variable mehrmals vorkommt, wie es beispielsweise in Bild 3.3 (Abschnitt 3.2) der Fall ist. Eine weitere aber eher unbedeutende Änderung ist, daß in der Implementierung nicht mehr von den modes *d, f, c, t* und *e*

gesprochen wird, sondern nun immer ihre Langformen dontknow, free, closed, true und empty benutzt werden.

Der mode-Interpreter wird über die Funktion

mode-interpret : QUERY x DB-NORM -> ExtTable

aufgerufen. QUERY ist eine Anfrageapproximation (q Aufrufpattern) für die (gemäß Abschnitt 3.2) normalisierte Datenbasis DB-NORM. Das Ergebnis ist die in LISP repräsentierte extension table aus Bild 3.5. Innerhalb von *mode-interpret* wird die extension table mittels der Funktion *index-db* und *compute-aliasBit* initialisiert. Die eigentliche abstrakte Interpretation wird von den Funktionen *md-interp-clauseList, md-interp-clause, md-interp-goallist, md-interp-is* und *md-interp-goal* ausgeführt, die innerhalb von *mode-interpret* durch *md-interp-goal* mit dem goal QUERY angestoßen wird. Die 5 Hauptfunktionen sollen im weiteren beschrieben werden.

md-interp-clauseList: PNAME x CALLP x INDEX-LIST x
C-ALI x ExtTable x DATABASE → ExtTable

Abarbeiten aller abstrakten Klauselaufrufe mit dem Aufrufpattern CALLP der Prozedur PNAME. Die Reihenfolge der Abarbeitung ist durch die Klauselindexliste INDEX-LIST vorgegeben, die Verweise auf die jeweiligen Klauseln der DATABASE enthält. Das Flag C-ALI zeigt an, ob in dem Literal, welches diesen Aufruf bewirkte, Variablen mehrmals vorkommen und somit die Gefahr des call-aliasing besteht. Ist die Prozedur PNAME nicht definiert (INDEX-LIST = NIL), wird eine Warnung an den Programmierer ausgegeben und mittels der Funktion *mk-noDef-succpat* eine sichere Approximation des Erfolgspattern berechnet und in die ExtTable eingetragen. Es wird also angenommen, daß es sich um eine *externe Prozedur* handelt, von der man keine genauen Angaben über ihr Verhalten machen kann und deswegen mit einer konservativen (aber sicheren) Abschätzung weitermacht.

md-interp-clause: CALLP x CLAUSE x LASTCL x C-ALI x
ExtTable x DATABASE → ExtTable

Diese Funktion entspricht dem gleichnamigen Algorithmus des vorherigen Abschnitts. Sie berechnet für die Klausel CLAUSE das Erfolgspattern SUCCP anhand der i-state Transformation. Bei der Berechnung des initialen Instantiierungsstatus wird C-ALI für call-aliasing berücksichtigt, sowie das Kopfliteral von CLAUSE auf return-aliasing getestet. Das Argument LASTCL gibt an, ob es sich um die letzte Klausel bezüglich der Klauselindexliste handelt. Später wird diese Information genutzt, um rekursive Aufrufe in *safe-rec* und *unsafe-rec* zu unterscheiden. Der finale Instantiierungsstatus F-STATE wird mittels der Funktion *md-interp-goallist* berechnet. Aus F-STATE und dem CALLP wird der SUCCP berechnet und die ExtTable entsprechend der LUB-Bildung mittels der Funktion *add-succp-to-table* aktualisiert

und als Wert von *md-interp-clause* zurückgegeben. Das Argument DATABASE wird hier nicht direkt gebraucht, sondern wird nur für andere Funktionen durchgereicht.

md-interp-goallist: PNAME x PREMISES x I-STATE x VAL x LASTCL x ExtTable x DATABASE → I-STATE x VAL x ExtTable

Abarbeiten der Prämissen PREMISES einer Klausel der Prozedur PNAME. PREMISES ist hierbei eine Liste von Literalen. Weiterhin ist der aktuelle Instantiierungsstatus der Klauselvariablen (I-STATE) sowie vom Klauselwert (VAL) gegeben. Die erste Prämisse von PREMISES ist das nächste abzuarbeitende abstrakte GOAL. Hierbei werden 4 Fälle unterschieden:

(1) GOAL ist ein denotatives Literal in foot-Position der Klausel. Es ändert sich also nur der Instantiierungsstatus von VAL, der sich nun zum Instantiierungsstatus des foot-Literals GOAL ergibt.

(2) GOAL ist ein RELFUN LISP-builtin. In RELFUN dürfen LISP-Durchgriffe nur mit Grundtermen geschehen und als Wert erhält man ebenfalls nur Grundterme. Entsprechend wird I-STATE durch eine i-state Transformation mit dem Erfolgspattern <closed, ..., closed> neu bestimmt und VAL ebenfalls auf closed gesetzt und beide zusammen als Wert zurückgegeben.

(3) GOAL ist ein is-Literal. In diesem Fall wird das Ergebnis durch die Funktion *md-interp-is* (s. u.) bestimmt.

(4) GOAL ist ein (normaler) Prozeduraufruf. Das SUCCP von GOAL wird durch den Aufruf von *md-interp-goal* berechnet. Dazu wird zuerst eine i-state Projektion von I-STATE auf GOAL vorgenommen, woraus das CALLP des abstrakten Aufrufs gewonnen wird. Handelt es sich um einen rekursiven Aufruf und ist GOAL die letzte Prämisse aus PREMISES sowie LASTCL = T, dann wird der Funktion *md-interp-goal* signalisiert, daß es sich um eine sichere Rekursion handelt. Anhand von SUCCP wird I-STATE und VAL neu berechnet und zurückgegeben.

md-interp-is: PNAME x REST-PREMISES x IS-TERM x I-STATE x LASTCL x ExtTable x DATABASE → I-STATE x VAL x ExtTable

Diese Funktion wird ausschließlich innerhalb von *md-interp-goallist* aufgerufen. Die Idee von *md-interp-is* ist, die rechte Seite von IS-TERM über *md-interp-goallist* wie ein foot-Literal abzuarbeiten und den Wert VAL, den *md-interp-goallist* unter anderem zurückliefert, mittels *s_unify* mit der linken Seite zu unifizieren. Anschließend wird die Kontrolle der Abarbeitung der restlichen Prämissen REST-PREMISES an die Funktion *md-interp-goallist* zurückgegeben.

md-interp-goal: PNAME x CALLP x KOF-REC x C-ALI x ExtTable x
DATABASE → SUCCP x ExtTable

Abarbeiten eines abstrakten Aufrufs der Prozedur PNAME mit dem Aufrufpattern CALLP. Handelt es sich um einen rekursiven Aufruf, gibt KOF-REC an, ob dieser *safe-rec* oder *unsafe-rec* ist. Zuerst wird der procEntry zu PNAME aus der ExtTable selektiert. Dann wird der *o/c-state* des entsprechenden succpatEntry auf *close, open* und *undef* getestet. Falls *o/c-state* = *close* gilt, wird die ExtTable unverändert zurückgegeben und das SUCCP aus dem succpatEntry bestimmt. Gilt *o/c-state* = *open* und KOF-REC = *safe-rec*, dann wird die Funktion *md-interp-goal* rekursiv mit dem neuen Aufrufpattern NEW-CALLP aufgerufen. Hierbei ergibt sich NEW-CALLP aus dem bisherigen Erfolgspattern des succpatEntry. Liegt keine sichere abstrakte Rekursion vor, wird mittels der Funktion *mk-succpat-close* eine konservative Abschätzung für SUCCP berechnet und die ExtTable entsprechend geändert. Falls *o/c-state* = *undef* (in LISP durch NIL repräsentiert) gilt, handelt es sich um den ersten abstrakten Aufruf dieser Art. Es wird die Klauselindexliste für diesen Aufruf aus der ExtTable selektiert sowie ein succpatEntry geöffnet und anschließend die Funktion *md-interp-clauseList* aufgerufen, die die neue ExtTable, also das Ergebnis des abstrakten Aufrufs, zurückliefert. Hiernach ist der abstrakte Aufruf vollständig abgearbeitet und der succpatEntry kann geschlossen werden. Unter Umständen kann zu diesem Zeitpunkt das succpatEntry bereits geschlossen sein, da beispielsweise aufgrund einer abstrakten Rekursion der succpatEntry geschlossen wird, bevor alle Klauseln einer Prozedur betrachtet worden sind. Für diesen Fall bleibt das erneute Schließen ohne Wirkung.

Neben diesen 5 Hauptfunktionen existieren eine ganze Reihe Hilfsfunktionen. Hierunter zählen zum einen die Selektoren, Konstruktoren und Prädikate, die eine abstrakte Syntax für das Managen der extension table realisieren. Sie werden weiter unten beschrieben. Im folgenden werden zuerst die wichtigen Hilfsfunktionen beschrieben, die nicht direkt zu der abstrakten Syntax beitragen. Sie werden teilweise durch kleine Beispiele erläutert. Hierzu betrachten wir erneut die "Spieldatenbasis" aus Abschnitt 3.2 Bild 3.1.

```
(ft (p _u _v) (q _u _v) (r _u) (s _v) )
(hn (q _u _u))
(hn (r a))
(hn (s _v))
```

Weiterhin sei die Datenbasis durch *rfi-database* und die extension table durch die LISP-Variable ExtTable repräsentiert.

index-db: DATABASE x ExtTable → ExtTable

Die Funktion *index-db* erstellt die Klauselindexlisten der einzelnen Prozeduren aus DATABASE. Existiert bereits ein *procEntry* in der ExtTable, so wird die Klauselindexliste erweitert, ansonsten wird zuerst ein *procEntry* erzeugt. Die erste Klausel in DATABASE erhält den Index 0.

Beispiel

```
? ExtTable
NIL
? (index-db *rfi-database* ExtTable)
(((P 2) NIL NIL (0))
 ((Q 2) NIL NIL (1))
 ((R 1) NIL NIL (2))
 ((S 1) NIL NIL (3)))
```

compute-aliasesBit: DATABASE x ExtTable → ExtTable

Diese Funktion berechnet gemäß dem Algorithmus SET-ALI-BIT, ob für eine Prozedur die Gefahr des *aliasing* besteht. Hierzu werden die Funktionen *return-aliasesBit* und *call-aliasesBit* aufgerufen, die die verschiedenen Ausprägungen des aliasing testen. Sei nun ExtTable an das obige Resultat des Aufrufs (index-db ...) gebunden.

Beispiel

```
? (compute-aliasesBit *rfi-database* ExtTable)
(((P 2) t NIL (0))
 ((Q 2) t NIL (1))
 ((R 1) NIL NIL (2))
 ((S 1) NIL NIL (3)))
```

term-state: LIT-ACTIVATION → TERM-STATE

Die Funktion *term-state* realisiert gerade die Erweiterung von ζ (definiert über Programmvariablen) auf beliebige Terme zu ζ'. Hierzu wird einfach der *lub* der Argumente bestimmt.

Beispiel

```
? (term-state 'free)
FREE
? (term-state '(inst (f closed closed true)))
CLOSED
```

lub: TERM-STATE-LIST → TERM-STATE

Berechnung des *lub* (least upper bound) gemäß (Δ, $\sqsubseteq$) von einer Menge (hier in Form einer Liste) von TERM-STATES. Entsprechend dem vorherigen Beispiel gilt:

Beispiel

```
? (lub '(closed closed true))
closed
```

pattern-lub: PATTERN x PATTERN → PATTERN

Berechnung eines kleinsten Pattern gemäß (Δ^*, $\sqsubseteq$), welches größer ist als die beiden Argumente. Die Funktion wird u. a. dazu benutzt, mehrere succpats zu einem speziellen Aufrufpattern durch ein einziges succpat (-> succpatEntry) zu approximieren. Man erkennt die Ähnlichkeit zu der *Antiunifikation*, die aus der Logik bekannt ist.

Beispiel

```
? (pattern-lub '(closed free dontknow true)
               '(true closed true true))
(CLOSED DONTKNOW DONTKNOW TRUE)
```

conser-approx: CALLP → SUCCP

Bei abstrakten Rekursionen, die nicht *safe-rec* sind, muß eine konservative Approximation des Erfolgspattern SUCCP vorgenommen werden. Dies wird durch die Funktion *conser-approx* realisiert.

Beispiel

```
? (conser-approx '(closed free dontknow true))
(CLOSED DONTKNOW DONTKNOW TRUE DONTKNOW)
```

s-unify: TERM-STATE x TERM-STATE → TERM-STATE

Die Funktion *s-unify* realisiert die gleichnamige abstrakte Unifikation, die in Abschnitt 3.2 über eine Wertetabelle eingeführt worden ist

Beispiel

```
? (s-unify 'closed 'true)
TRUE
? (s-unify 'free 'dontknow)
DONTKNOW
```

Der Instantiierungsstatus der Programmvariablen ist durch eine LISP-Liste der Form

((*var1 state*) ... (*varN state*))

gegeben. Die folgenden Funktionen benutzen den Instantiierungsstatus, der durch die LISP-Variable I-STATE repräsentiert ist[51].

```
? > (rf-pprint (setf I-STATE '(( (VARI U) free) ( (VARI V) free))))
((_u free) (_v free))
```

project-state: LITERAL x I-STATE → LIT-ACTIVATION

Durch eine i-state Projektion wird das abstrakte goal bzw. die LIT-ACTIVATION des Literals LITERAL gebildet.

Beispiel

```
? (project-state '(q _u _v) I-STATE)
(Q free free)
```

safe-istate: LITERAL x SUCCP x I-STATE x ALIASING
→ I-STATE

Nachdem mittels *project-state* und dem LITERAL sowie dem aktuellen I-STATE ein abstrakter Aufruf geschehen ist, wird ein Erfolgspattern SUCCP zurückgegeben. Die i-state Transformation δ/3 (siehe Abschnitt 3.2) ergibt hieraus den neuen i-state. Die Transformation wird mit Hilfe der Funktion *safe-istate* realisiert, wobei auch eventuelle aliasing-Effekt (angezeigt durch das Flag ALIASING) berücksichtigt werden.

Beispiel

```
? (safe-istate '(q (vari u) (vari v))        ; LITERAL
               '(free free true)             ; SUCCP vom Aufruf (q free free)!
               I-STATE                       ; = ((_u free) (_v free)) (s. o.)
               t)                            ; Der Aufruf q/2 birgt aliasing
((_u dontknow) (_v dontknow))
```

struc-unify: VARIABLE x STRUCTURE x I-STATE → I-STATE

Ein denotatives is-Literale wird mit Hilfe dieser Funktion abgearbeitet. Da wir im mode-Inter-

[51] In den LISP-Beispielen werden RELFUN-Variablen teils mit führenden Unterstrich (z. B. "_v") und teils als Liste mit dem Funktor *vari* (z. B. "(vari v)") dargestellt. Die erste Darstellung ist die, die der Benutzer sieht und die zweite entspricht der internen Darstellung in RELFUN. Der Benutzer ist aber frei, welche Darstellung er bei der Eingabe von Klauseln wählt.

preter nur Kern-Klauseln betrachten, brauchen wir nur solche Fälle beachten, in denen auf der linken Seite eine Variable oder Konstante steht. Den ersten dieser beiden Fälle wird mittels *struc-unify* modelliert und der neue Instantiierungsstatus I-STATE zurückgegeben.

Beispiel

```
; (is _v `(f b)), I-STATE = ((_u free) (_v free))
? (struc-unify '(vari v)  '(inst (f b)) I-STATE)
((_u free) (_v closed))

; (is _v `(f _u)), I-STATE = ((_u free) (_v free))
? (struc-unify '(vari v)  '(inst (f (vari u))) I-STATE)
((_u free) (_v dontknow))
```

Wie schon weiter oben erwähnt, existieren eine Reihe von Zugriffsfunktionen auf die Datenstrukturen im mode-Interpreter, die zusammen eine abstrakte Syntax bilden. Im folgenden werden sie kurz vorgestellt.

Selektoren und Kollektoren

s-var-state: VARIABLE x I-STATE → VAR-STATE
Die Funktion *s-var-state* selektiert aus dem I-STATE den aktuellen Instantiierungsstatus von VARIABLE. Existiert kein Eintrag in der Liste I-STATE, wird implizit der Instantiierungsstatus *free* angenommen und zurückgegeben.

Beispiel

```
? (s-var-state '(vari u) I-STATE)
free
? (s-var-state '(vari w) I-STATE)
free
```

collect-new-vars: TERM x VARIABLES → VARIABLES
Aufsammeln von Variablen, die in TERM neu vorkommen.

Beispiel

```
? (collect-new-vars '(q (vari u) (vari v)) nil)
((VARI U) (VARI V))
? (collect-new-vars '(q (vari u) '((VARI V))) nil)
((VARI U))
```

Die restlichen Selektoren und Konstruktoren beziehen sich im wesentlichen auf die extension table. Es sind die Funktionen:

s-procEntry : PNAME x ExtTable → procEntry
s-succpat-table : procEntry → succpatTable
s-succpat-entry : CALLP x procEntry → succpatEntry
s-succpat : callpat x procEntry → succpat
s-succpat-state : callpat x procEntry → OC-State
s-indexList : procEntry → CL-INDEXLIST
s-aliasesBit ;: procEntry → aliBit
collect-aliasesBit ExtTable -> PNAME-LIST

Die Semantik der Funktionen ist offensichtlich, trotzdem sollen ein paar Beispiele diese ein wenig illustrieren.

Beispiel

```
? (setf ExtTable (mode-interpret '(p free free) *rfi-database*))
(((P 2) T
  (((FREE FREE) (CLOSED DONTKNOW TRUE) CLOSE))
  (0))
 ((S 1) NIL
  (((DONTKNOW) (DONTKNOW TRUE) CLOSE)) (3))
 ((R 1) NIL (((DONTKNOW) (CLOSED TRUE) CLOSE))
  (2))
 ((Q 2) T
  (((FREE FREE) (DONTKNOW DONTKNOW TRUE)
    CLOSE))
  (1)))

? (setf procEntry (s-procEntry '(r 1) ExtTable ))
((R 1) NIL (((DONTKNOW) (CLOSED TRUE) CLOSE)) (2))

? (s-succpat-Table procEntry)
(((DONTKNOW) (CLOSED TRUE) CLOSE))

? (s-succpat-entry '(dontknow) procEntry)
((DONTKNOW) (CLOSED TRUE) CLOSE)

? (s-succpat '(dontknow) procEntry)
(CLOSED TRUE)

? (s-succpat-state '(dontknow) procEntry)
close

? (s-indexList procEntry)
(2)

? (s-aliasesBit procEntry)
NIL

? (collect-aliasesBit ExtTable)
((P 2) (Q 2))
```

Konstruktoren und Modifizierer

Die nun folgenden Funktionen gestatten es, Strukturen der extension table aufzubauen und zu modifizieren.

create-ExtTable : → ExtTable
add-procEntry : procEntry x ExtTable → ExtTable
rm-procEntry : PNAME x ExtTable → ExtTable
modify-procEntry : PNAME x newEntry x ExtTable → ExtTable
addnew-procEntry : PNAME x ExtTable → ExtTable

mk-procEntry : PNAME x ALI-BIT x SUCCPAT-TABLE
x INDEX-LIST → procEntry

Diese Funktionen erzeugen eine leere extension table[52] (-> *create-ExtTable*), fügen Prozedureinträge zu eine bestehenden ExtTable hinzu (-> *add-procEntry*) bzw. löschen Prozedureinträge (-> *rm-procEntry*). Die häufigste Anwendung der letzten beiden Funktionen erfolgt beim Ändern von Prozedureinträgen (-> *modify-procEntry*), denn hier wird nacheinander *rm-procEntry* und *add-procEntry* aufgerufen. Die Funktion *addnew-procEntry* erzeugt zuerst einen leeren Prozedureintrag mittels (*mk-procEntry* PNAME nil nil nil) und fügt diesen zur ExtTable hinzu.

Beispiel

```
? (setf ExtTable (create-ExtTable))
NIL

? (setf procEntry (mk-procEntry `(p 2) nil nil nil))
((P 2) NIL NIL NIL)

? (setf ExtTable (add-procEntry procEntry ExtTable))
(((P 2) NIL NIL NIL))

? > (modify-procEntry '(p 2) (mk-procEntry '(p 2) t nil nil) ExtTable)
(((P 2) T NIL NIL))
```

mk-succpatEntry : CALLP x SUCCP x O/C-STATE → succpatEntry
modify-succpatEntry : CALLP x SUCCP x O/C-STATE x procEntry → procEntry

Die Funktion *mk-succpatEntry* konstruiert aus dem CALLP, SUCCP und O/C-STATE einen

[52]Die leere extension table entspricht zur Zeit der leeren Liste NIL in LISP

succpatEntry. Die Funktion *modify-succpatEntry* selektiert aus dem procEntry den succpatEntry zu CALLP und ersetzt ihn durch den neuen, der sich aus CALLP, SUCCP und O/C-STATE ergibt. Existiert noch kein succpatEntry zu CALLP im procEntry, wird der neue succpatEntry trotzdem hinzugefügt.

mk-succpat-open : PNAME x CALLP x ExtTable → ExtTable
mk-succpat-close : CALLP x procEntry x WHY x ExtTable
→ succpatEntry x ExtTable

Wird eine Prozedur PNAME mit einem Aufrufpattern CALLP aufgerufen, wird mittels *mk-succpat-open* ein entsprechender succpatEntry eröffnet. Ist der Aufruf abgearbeitet, wird das succpatEntry mittels *mk-succpat-open* wieder geschlossen. Das Argument WHY gibt hierbei den Grund für die Beendigung der Abarbeitung an. Es gibt 3 verschieden Begründungen:

WHY = *goal-finished* Alle Klauseln wurden für den Aufruf beachtet und demnach ist das bisherige Erfolgspattern im succpatEntry zu PNAME und CALLP richtig berechnet. Es wird daher als Ergebnis zurückgeliefert und der o/c-state auf *close* gesetzt.

WHY = *safe-rec* Es erfolgte ein sicherer rekursiver Aufruf auf der abstrakten Ebene und das CALLP und SUCCP sind bis auf den Instantiierungsstatus für *val* identisch. Es kann somit wie im Fall WHY = *goal-finished* verfahren werden.

WHY = *unsafe-rec* Es erfolgte ein nicht-sicherer rekursiver Aufruf in der abstrakten Ebene. Der succpatEntry wird geschlossen und das SUCCP konservativ (s. o.) abgeschätzt.

add-succp-to-table : PNAME x CALLP x SUCCP x ExtTable → ExtTable

Während der Abarbeitung eines abstrakten Aufrufs (p CALLP) liefern die einzelnen Klausel verschiedene Erfolgspattern SUCCP. Sie werden durch *add-succp-to-table* in die ExtTable, unter Beachtung des bisher berechneten Erfolgspattern also der (LUB-Bildung), eingetragen.

Beispiel

```
? (setf pname '(p 3)
        callp '(closed free free))     ; = (p closed free free)
(CLOSED FREE FREE)

? (setf ExtTable (addnew-procEntry PNAME (create-ExtTable)))
(((P 3) NIL NIL NIL))

? (setf ExtTable
        (mk-succpat-open PNAME CALLP ExtTable))
(((P 3) NIL (((CLOSED FREE FREE) NIL OPEN)) NIL))

? (setf ExtTable
        (add-succp-to-table PNAME CALLP
                            '(closed free true true) ; SUCCP
```

```
                                ExtTable))
(((P 3) NIL                    ; ALI-BIT
        (((CLOSED FREE FREE) (CLOSED FREE TRUE TRUE) OPEN)); succpatTable
        NIL))                  ; Klauselindexliste

? (setf ExtTable
        (add-succp-to-table PNAME CALLP
                            '(closed free closed true) ExtTable))
(((P 3) NIL
        (((CLOSED FREE FREE) (CLOSED FREE CLOSED TRUE) OPEN))
        NIL))

? (mk-succpat-close PNAME CALLP 'goal-finished ExtTable)
((CLOSED FREE CLOSED TRUE)
 (((P 3) NIL
         (((CLOSED FREE FREE) (CLOSED FREE CLOSED TRUE) CLOSE))
         NIL)))

? (mk-succpat-close PNAME CALLP 'safe-rec ExtTable)
((CLOSED FREE CLOSED TRUE)
 (((P 3) NIL
         (((CLOSED FREE FREE) (CLOSED FREE CLOSED TRUE) CLOSE))
         NIL)))

? (mk-succpat-close PNAME CALLP 'unsafe-rec ExtTable)
((CLOSED DONTKNOW DONTKNOW DONTKNOW)
 (((P 3) NIL
         (((CLOSED FREE FREE)
           (CLOSED DONTKNOW DONTKNOW DONTKNOW) CLOSE))
         NIL)))
```

Prädikate

inclusion-p: TERM-STATE x TERM-STATE → TRUTH-VALUE

Test, ob der rechte TERM-STATE den linken überdeckt bezüglich (Δ, $\sqsubseteq$).

Beispiel

```
? (inclusion-p 'closed 'true)
T
? (inclusion-p 'true 'closed)
NIL
? (inclusion-p 'free 'free)
T
```

repeated-variable-p: LITERAL → TRUTH-VALUE

Test, ob in LITERAL eine Variable mehrmals vorkommt.

Beispiel

```
? (repeated-variable-p '(foo (vari v) (vari w) (vari y) (vari v)))
T
? (repeated-variable-p '(foo (vari v) (vari w) (vari y) (vari z)))
NIL
```

recursive-call-p: CLAUSE → TRUTH-VALUE

Test, ob in die Klausel rekursiv definiert ist.

Beispiel

```
? (recursive-call-p '(ft (foo a) (bar b) (foo c)))
T
? (recursive-call-p '(ft (foo a) (bar b) `(foo c)))
NIL
```

return-aliases-p: CL-INDEX-LIST x DATABASE → TRUTH-VALUE

Test, ob eine Prozedur, gegeben durch die Klauselindexliste CL-INDEX-LIST der Prozedurklauseln, die Gefahr des return-aliasing birgt. Sei wieder die Spieldatenbasis vom Anfang durch *rfi-database* gegeben und die ExtTable mittels *index-db* erzeugt.

Beispiel

```
? (return-aliases-p (s-indexList (s-procEntry '(q 2) ExtTable))
                    *rfi-database*)
T
? (return-aliases-p (s-indexList (s-procEntry '(s 1) ExtTable))
                    *rfi-database*)
NIL
```

call-to-aliases-p: LITERAL x ALI-LIST → TRUTH-VALUE

Test, ob das Literal LITERAL einer Prozedur angehört, für die bereits die Gefahr des aliasing festgestellt wurde und deshalb in der ALI-LIST vorhanden ist. Die Funktionen *call-to-aliases-p* und *return-aliases-p* bilden den Kern des Algorithmus SET-ALI-BIT aus Abschnitt 3.2.

Beispiel

```
? (call-to-aliases-p '(q _u _v) '((q 2)))
T
```

exist-procEntry-p:	PNAME x ExtTable	→ TRUTH-VALUE
exist-succpatEntry-p:	CALLP x procEntry	→ TRUTH-VALUE
aliases-p:	procEntry	→ TRUTH-VALUE
succpat-close-p:	CALLP x procEntry	→ TRUTH-VALUE
succpat-open-p:	CALLP x procEntry	→ TRUTH-VALUE

Prädikate, die die Eigenschaften (Slotinhalte) der extension table ExtTable und ihrer Komponenten testen. Die Semantik ist für diese Funktionen mehr als offensichtlich und es wird deswegen auf Beispiele verzichtet.

3.3.4 Einbindung des mode-Interpreters in das RFM-System

In diesem letzten Abschnitt wird die Schnittstelle des mode-Interpreters mit dem RFM-System vorgestellt. Wie im vorherigen Abschnitt erläutert, wird der mode-Interpreter über die Funktion

mode-interpret : QUERY x DB-NORM -> ExtTable

aufgerufen. Als Ergebnis wird die extension table zurückgeliefert, in der die modes der Prozeduren repräsentiert ist. Die im weiteren beschriebene Schnittstelle zwischen RFM-System und mode-Interpreter gibt zum einen die extension table in einer informativeren Darstellung für den Benutzer wieder (siehe hierzu das Beispiel weiter unten) und erweitert zum anderen die Aufrufmöglichkeiten des mode-Interpreters. Der mode-Interpreter wird im RFM-System durch das Kommando *modes* aktiviert. Es kann auf 4 verschiedene Art und Weise benutzt werden:

(1) modes *query*

(2) modes *procedure*

(3) modes *query-list*

(4) modes *procedure-list*

Unter einer *query* ist eine Anfrageapproximation der Form (pred-name Aufrufpattern) zu verstehen etwa (foo free closed). Eine *procedure* ist durch das Paar (pred-name pred-arity) beschrieben etwa (foo 2). Es wird interpretiert, wie die Anfrageapproximation (pred-name $\delta_1, \ldots, \delta_{\text{pred-arity}}$) mit δ_i = dontknow. Die *query-list* (*procedure-list*) ist eine Liste von *query* (*procedure*). Mit ihnen hat man die Möglichkeit eine Menge von Anfragen zu betrachten.

Sind für eine konkrete RELFUN-Datenbasis die möglichen Anfragen genau bekannt, verwendet man sinvoller Weise die Aufrufe der Form (1) und (3). Sind jedeglich die (erlaubten) Prozeduren bekannt, welche als Benutzer-goals zur Laufzeit möglich sind, so findet die abkürzende Schreibweise aus (2) und (4) Anwendung. Man hat in diesem Fall nur Informationen darüber, daß die Prozedur *procedure* aufgerufen wird, aber keine näheren Informationen, wie die Instantiierungen beim Aufruf sind. Die Schnittstellenfunktionen sind in der Datei "mode-rfi-interface.lsp" zu finden. Das RFM-System ruft dieses Inferface mittels der Funktion

rfi-modes : USERLINE -> ExtTable

auf. USERLINE hat hierbei eine der Darstellungen (1) bis (4). Für (1) und (2) wird die Funktion *md-interpret* direkt aufgerufen und für die Fälle (3) und (4) wird zuerst eine Pseudo-Klausel *md-query* generiert, die die *query-list* (*procedure-list*) als Prämissen enthält. Diese Vorgehensweise geschieht transparent für den Benutzer. Beispielsweise wird für

modes (foo free closed) (bar true closed dontknow)

die Pseudo-Klausel

(ft (md-query _1 _2 _3 _4 _5) (foo _1 _2) (bar _3 _4 _5))

erzeugt und zusammen mit der eigentlichen RELFUN-Datenbasis dem mode-Interpreter übergeben, wobei die Anfrageapproximation die Form

(md-query free closed true closed dontknow)

hat. Entsprechen erhalten wir für die Verwendungsmöglichkeit (4):

modes (foo 2) (bar 3)
(ft (md-query _1 _2 _3 _4 _5) (foo _1 _2) (bar _3 _4 _5))
(md-query dontknow dontknow dontknow dontknow dontknow)

Die Funktion *rfi-modes* liefert, genau wie *md-interpret*, die extension table zurück. Dies geschieht aber unsichtbar für den Benutzer. Der Benutzer sieht vielmehr eine informativere Darstellung der extension table. Diese Darstellung soll nun an einem Beispielprotokoll einer möglichen RELFUN-Sitzung illustriert werden. In dem Beispiel betrachten wir zwei kleine Datenbasen. Die erste Datenbasis ("mini-db.rf") besteht nur aus zwei Klauseln, die aber Prädikate/Funktionen aus der zweiten Datenbasis ("fac.rf") aufrufen. Anhand einer ersten mode-Analyse von "mini-db.rf" wird festgestellt, wie die (importierten) Prädikate/Funktionen aus "fac.rf" aufgerufen werden. Mit dieser Information kann dann für diesen speziellen Gebrauch von "fac.rf" eine mode-Analyse für "fac.rf" gestartet werden.

```
rfi> ; Load the database "mini-db.rf"
rfi> consult "mini-db"
rfi> ; List the database
rfi> l
(ft (testfun _n)
    (numberp _n)
    (facfun _n) )
(ft (testrel _n _val)
    (numberp _n)
    (facrel _n _val) )
rfi>
rfi> ; There exist only two functions in the database.
rfi> ; Assume we know nothing about the functions.
rfi> ; So, we compute the modes with no strong calling patterns and
rfi> ; get the extension table in a suited manner
```

```
rfi> ; Note, the function facfun/1 und the relation facrel/2 are defined in
rfi> ; the database "fac.rf" and have no known definition yet. Nevertheless
rfi> ; the mode-interpreter computes nice calling pattern for facfun/1 and
rfi> ; facrel/2.
rfi> ; Remember, a call to LISP-builtins like numberp/1 are only allowed with
rfi> ; ground terms!
rfi>
rfi> modes (testfun 1) (testrel 2)
*** Warning ***there is no clause definition for >>FACFUN/1<<
*** Warning ***there is no clause definition for >>FACREL/2<<

Procedure: TESTREL/2  Aliasing: yes
calling: <DONTKNOW DONTKNOW > returning: <CLOSED DONTKNOW DONTKNOW >

Procedure: FACREL/2  Aliasing: no
calling: <CLOSED DONTKNOW > returning: <CLOSED DONTKNOW DONTKNOW >

Procedure: TESTFUN/1  Aliasing: yes
calling: <DONTKNOW > returning: <CLOSED DONTKNOW >

Procedure: FACFUN/1  Aliasing: no
calling: <CLOSED > returning: <CLOSED DONTKNOW >
rfi>
rfi> ; Now we can look at the database "fac.rf".
rfi> replace "fac.rf"
rfi> l
(hn (facrel 0 1))
(hn (facrel _n _r)
    (sub1rel _n _p)
    (facrel _p _v)
    (multrel _n _v _r) )
(ft (facfun 0)
    1 )
(ft (facfun _n)
    (multfun _n (facfun (sub1fun _n))) )
(hn (sub1rel _n _p)
    (is _p (sub1fun _n)) )
(hn (multrel _n _v _r)
    (is _r (multfun _n _v)) )
(ft (sub1fun _n)
    (1- _n) )
(ft (multfun _n _v)
    (* _n _v) )
rfi>
rfi> ; Normalizing the database!
rfi> flatter
rfi> horizon
rfi>
rfi> ; And now computing the modes for all the predicates in the database
rfi> ; with respect to the call information of facfun/1 and facrel/2
rfi>
rfi> modes (facfun closed) (facrel closed dontknow)

Procedure: FACREL/2  Aliasing: yes
calling: <CLOSED DONTKNOW > returning: <CLOSED CLOSED TRUE >
calling: <CLOSED FREE > returning: <CLOSED DONTKNOW DONTKNOW >

Procedure: MULTREL/3  Aliasing: yes
calling: <CLOSED DONTKNOW DONTKNOW > returning: <CLOSED DONTKNOW CLOSED TRUE
>
calling: <CLOSED DONTKNOW FREE > returning: <CLOSED DONTKNOW CLOSED TRUE >

Procedure: SUB1REL/2  Aliasing: yes
```

```
calling: <CLOSED FREE > returning: <CLOSED CLOSED TRUE >

Procedure: FACFUN/1  Aliasing: yes
calling: <CLOSED > returning: <CLOSED DONTKNOW >

Procedure: MULTFUN/2  Aliasing: yes
calling: <CLOSED DONTKNOW > returning: <CLOSED CLOSED CLOSED >

Procedure: SUB1FUN/1  Aliasing: yes
calling: <CLOSED > returning: <CLOSED CLOSED >
rfi>
rfi> ; Maybe there exist another database using "fac.rf",  calling only
rfi> ; facfun/1 with calling pattern <closed> then we get additional
rfi> ; information about dead code, indicated by prodedures with no
rfi> ; calling and returning pattern.
rfi>
rfi> modes (facfun closed)

Procedure: FACFUN/1  Aliasing: yes
calling: <CLOSED > returning: <CLOSED DONTKNOW >

Procedure: MULTFUN/2  Aliasing: yes
calling: <CLOSED DONTKNOW > returning: <CLOSED CLOSED CLOSED >

Procedure: SUB1FUN/1  Aliasing: yes
calling: <CLOSED > returning: <CLOSED CLOSED >

Procedure: FACREL/2  Aliasing: yes

Procedure: SUB1REL/2  Aliasing: yes

Procedure: MULTREL/3  Aliasing: yes
```

4 Zusammenfassung und Ausblick

In dieser Arbeit habe ich kurz die Compilationsphilosophie des *Complabs* im C-Teil von ARC-TEC beschrieben. Im Sinne dieser Philosophie habe ich 6 Regelsysteme vorgestellt, die Klausel-basierte Source-to-Source-Transformationen von RELFUN beschreiben und zusammen mit dem Kontrollalgorithmus die Kern-Klauseln definieren. Die Kern-Klauseln ihrerseits bilden den RELFUN-Kern. Als *normalizer* wurden diese Transformationen in Form eines LISP-Programms implementiert.

Zwei weitere Transformationen wurden nur zum Teil bzw. überhaupt nicht beschrieben, sie sind aber im RFM-System integriert. Zum einen ist dies die Umwandlung von hn-Klauseln in ft-Klauseln. Es wurde zwar in Abschnitt 1.2 erklärt, wie diese Transformation zu realisieren ist, doch wurde sie nicht direkt in den Regelsystemen repräsentiert. Zum anderen wurden Listen, in RELFUN durch tup-Strukturen mit beliebiger Stelligkeit dargestellt, nicht direkt betrachtet. Ich habe für die Regelsysteme die Umwandlung der tup-Strukturen (Listen) in binäre cns-Strukturen (ähnlich den *dotted pairs* in LISP) bereits vorausgesetzt.

Ein Kritikpunkt des *normalizers* ist sicherlich die ausschließlich klauselorientierte Sichtweise. Die Regelsysteme haben zwar in ihrem Bedingungsteil prozedurorientierte Elemente und auch der *normalizer* ist für eine derartige Erweiterung teilweise vorbereitet; allerdings sind in der jetzigen Version beispielsweise die prozedurorientierten Bedingungen wie etwa Determinismus und Seiteneffektfreiheit nur für LISP-Durchgriffe positiv entscheidbar. In die prozedurorientierte Richtung geht dann der zweite Teil meiner Arbeit, der zugleich die erste Anwendung der Kern-Klauseln darstellt.

Anhand einer Datenflußanalyse, realisiert durch den *mode-Interpreter*, werden automatisch die möglichen Argumentinstantiierungen der Prädikate einer vorgegebenen Datenbasis inferiert. Sie werden als die modes der Prädikate (bzw. Prozeduren) bezeichnet. Die mode-Informationen sind im strengen Sinne bereits oberhalb der Prozedur- in die Datenbasisebene einzuordnen. Im Motivationsteil zu Abschnitt 3 wurden Anwendungen der mode-Informationen erläutert (etwa die Determinismus-Analyse). Zukünftige Arbeiten können hier anknüpfen und die modes für die Inferenz von weiterem Prozedurwissen nutzen. Möglicherweise ist es dann auch sinnvoll, den *normalizer* mit den zusätzlichen Prozedurinformationen erneut aufzurufen (also auf dieser Ebene weiter zu optimieren) und anschließend wieder den *mode-Interpreter* aufzurufen usw. Der wechselseitige Aufruf der Komponenten (möglicherweise auch weiteren) geschieht dann bis zum Erreichen eines "Fixpunkts", wenn sich keine oder nur noch sehr geringe Änderungen in den Ergebnissen einstellen.

Anhang A: Programmlisting des *normalizers*

A.1 Datei "normalizer.lisp"

```
;;;;;
;;;;;   The normalizer computes the RELFUN kernel for a
;;;;;   RELFUN program
;;;;;
;;;;;  Thomas Krause, DFKI Kaiserslautern
;;;;;  22. March 1991


(defun normalize-database (db &optional (count nil)) "DATABASE -> DATABASE"
;;;;
;;;; REMARK: COUNT is for debugging only
;;;;
   (declare (special *error-level* *error-stream*)); for debugging only
   (setq *error-stream* nil)
   (cond ((null db) nil)
         (t
          (let ((clause (first db))
                (proc-name (mk-proc-name (first db))))
            (case  *error-level*
              ((1 2 3) (setf *error-stream* (list (list proc-name clause))))
              (4 (setf *error-stream* (cons (list proc-name clause) *error-stream*)))
              (otherwise (setq *error-stream* nil)))

            (if count (progn (princ count) (princ " ")))

            (cons (first (normalize-clause clause nil))
                  (normalize-database (rest db) (if count (1+ count) nil)))))))

(defun normalize-clause (clause proc-info)
  "CLAUSE x PROC-INFO -> CLAUSE x PROC-INFO"
  (normalize-flat-clause clause proc-info)) ; clause must be allready flatten
                                            ; by using flatten-struc-clause

(defun normalize-flat-clause (cl proc-info)
  (let ((funM nil) ;holds values from multiple values functions
        (head-chunk nil) (cl-info nil) (guards nil)(prem-rest nil)
        (kind (s-kind cl))
        (head (s-conclusion cl))
        (premises (s-premises cl)))

    ;;; collecting the guards of the first chunk (namely the head chunk)
    (setf funM (collect-guards premises)) ; funM = <guard-list x <call-literal x premises>>
    (setf guards (first funM)
          prem-rest (second funM))


    ;;; computing the normalized head chunk
    (setf funM                                ;;;  status x head chunk x clause info
          (norm-head-chunk kind head guards prem-rest))
    (setf head-chunk (second funM)
          cl-info (third funM))

    (case (first funM) ; whats happened by normalizing?
      (unknown  ; detecting unknown
       (list (mk-clause kind head '(unknown))
             proc-info))

      (lisp-evaluate  ;  LISP expression was evaluated, as a result there are new guards!
       (normalize-flat-clause   ; with the computed lisp result
        (mk-clause kind (first head-chunk)
                   (append (rest head-chunk)
                           (rewrite (rest prem-rest)
                                    (s-simple-bindings cl-info))))
        proc-info))

      (lisp-error
```

```
        (error "Lisp error: NOT YET IMPLEMENTED") (terpri))

       (ok
        (norm-premises (cons kind head-chunk) (rest prem-rest) cl-info proc-info))

       (otherwise
        (error "normalize-clause")))))


(defun norm-premises (cl-new cl-rest cl-info proc-info)
"CL-NEW x CL-REST x CL-INFO x PROC-INFO -> clause x proc-info"
   (cond ((null cl-rest)
          (list cl-new proc-info))
         (t
          (let ((funM nil)
                (guards nil) (prem-rest nil)
                (chunk nil) (cl-info-new nil))

            ;;; collecting the guards of the next chunk
            (setf funM (collect-guards cl-rest))
            (setf guards (first funM)
                  prem-rest (second funM))

            (setf funM                      ;;; status x head chunk x clause info
                  (norm-chunk guards prem-rest cl-info))
            (setf chunk (second funM)
                  cl-info-new (third funM))


            (case (first funM) ; whats happened by normalizing?
              (unknown  ; detecting unknown
               (list (mk-clause (s-kind cl-new)
                                (s-conclusion cl-new)
                                '(unknown))
                     proc-info))

              (lisp-evaluate  ; LISP expression was evaluated, as a result there are new guards!
                (norm-premises cl-new
                               (append chunk
                                       (rewrite (rest prem-rest)
                                                (s-simple-bindings cl-info-new)))
                               cl-info proc-info))

              (lisp-error
               (error "Lisp error: NOT YET IMPLEMENTED") (terpri))

              (ok
               (norm-premises (append cl-new chunk)
                              (rest prem-rest)
                              cl-info-new proc-info))

              (otherwise
               (error "normalize-clause")))))
         ))


(defun norm-head-chunk (kind head guards prem-rest)
"KIND x HEAD x GUARDS x PREM-REST -> STATUS x HEAD-CHUNK x CL-INFO"
  (let ((funM nil)
        (new-guards nil) (chunk-vars nil)
        (bindings-s nil) (bindings-c nil) (new-head nil)
        (chunk-foot (s-chunk-foot guards)))

    (setf funM
          (norm-guards guards nil bindings-s bindings-c))

    (setf new-guards (first funM)
          bindings-s (second funM)
          bindings-c (third funM)
          chunk-vars (fourth funM))

    (cond ((eq new-guards 'unknown)
           (list 'unknown))

          ((null prem-rest)
           (setf new-head (rewrite head bindings-s))
           (list 'ok
```

```
                (append (cons new-head
                              (sort-head-guards new-guards
                                                (collect-new-vars new-head nil)))
                        (cond ((eq kind 'ft)
                               (list
                                (norm-foot (rewrite chunk-foot bindings-s)
                                           bindings-c)))
                              (t ; hn-clause
                               nil)
                              ))
                (mk-cl-info kind (collect-vars head chunk-vars)
                            bindings-s bindings-c)))

          (t
           (let ((prem (rewrite (first prem-rest) bindings-s)))
             (cond ((lispcall-inst-p prem bindings-c) ; full instantiate call to lisp
                    (list 'lisp-evaluate
                          (append (cons (rewrite head bindings-s)
                                        new-guards)
                                  (list (cond ((is-t prem)
                                               (mk-is (s-patt-is prem)
                                                      (un-inst (lisp-exec
                                                                (rewrite (s-expr-is prem)
                                                                         bindings-c) nil))))
                                              (t
                                               (lisp-exec (rewrite prem bindings-c) nil)))))
                          (mk-cl-info kind (collect-vars head chunk-vars)
                                      bindings-s bindings-c)))

                   (t
                    (setf chunk-vars (collect-vars prem chunk-vars)
                          new-head (rewrite head bindings-s))
                    (list 'ok
                          (append (cons new-head
                                        (sort-head-guards new-guards
                                                          (collect-new-vars new-head nil)))
                                  (list (rewrite (first prem-rest)
                                                 bindings-s)))
                          (mk-cl-info kind (collect-vars head chunk-vars)
                                      bindings-s bindings-c)))
                   )))
          )))


(defun norm-chunk (guards prem-rest cl-info)
"GUARDS x PREM-REST x CL-INFO -> STATUS x BODY-CHUNKN x CL-INFO"
  (let ((funM nil)
        (new-guards nil) (chunk-vars nil)
        (bindings-s (s-simple-bindings cl-info))
        (bindings-c (s-complex-bindings cl-info))
        (chunk-foot (s-chunk-foot guards)))

    (setf funM
          (norm-guards guards (s-cl-info-variables cl-info)
                       bindings-s bindings-c))
    (setf new-guards (first funM)
          bindings-s (second funM)
          bindings-c (third funM)
          chunk-vars (fourth funM))

    (cond ((eq new-guards 'unknown)
           (list 'unknown))

          ((null prem-rest)
           (list 'ok
                 (append  new-guards
                          (cond ((eq (s-cl-info-kind cl-info)'ft)
                                 (list
                                  (norm-foot (rewrite chunk-foot bindings-s)
                                             bindings-c)))
                                (t ; hn-clause
                                 nil)
                                ))
                 (mk-cl-info (s-cl-info-kind cl-info)
                             chunk-vars
                             bindings-s bindings-c)))
```

```
        (t
         (let ((prem (rewrite (first prem-rest) bindings-s)))
           (cond ((lispcall-inst-p prem bindings-c) ; full instantiate call to lisp
                  (list 'lisp-evaluate
                        (append guards
                                (list (cond ((is-t prem)
                                             (mk-is (s-patt-is prem)
                                                    (un-inst (lisp-exec
                                                              (rewrite (s-expr-is prem)
                                                                       bindings-c) nil))))
                                            (t
                                             (lisp-exec (rewrite prem bindings-c) nil)))))
                        (mk-cl-info (s-cl-info-kind cl-info)
                                    chunk-vars
                                    (s-simple-bindings cl-info)
                                    (s-complex-bindings cl-info) )))
                 (t
                  (setf chunk-vars (collect-vars prem chunk-vars))
                  (list 'ok
                        (append new-guards (list prem))
                        (mk-cl-info (s-cl-info-kind cl-info)
                                    chunk-vars
                                    bindings-s bindings-c))
                 ))))
        )))


(defun norm-guards (guards var-before bindings-s bindings-c)
  "GUARDS x VAR-BEFORE x BINDINGS-S x BINDINGS-C
          -> NEW-GUARDS BINDINGS-S x BINDINGS-C x VAR-AFTER"
  (norm-guards2 guards nil nil var-before nil bindings-s bindings-c nil))

(defun norm-guards2 (guard-rest new-guards conflict-guards
                           var-before var-add
                           bind-s
                           bind-c nbind-c) ; complex- and new-complex-bindings

  "computes the normalized guards called new-guards. The computation looks like a fixpoint
   computation. This means that a guard is added to the list CONFLICCT-GUARDS if it
   causes a meaningfull change to the complex bindings. As a result, maybe two rhs of
   BINDINGS-C become identity and structure sharing is possible. This is indicated by the
   function COLLECT-IS-SET-FROM-NEW-BINDINGS and a new iteration is started with the new
   guards, gaining by COLLECT-IS-SET-FROM-NEW-BINDINGS and  the guards of conflict-guards"

  (cond ((null guard-rest)
         (let ((more-is-set (collect-is-set-from-new-bindings bind-c nbind-c)))
           (cond (more-is-set   ;;; one more iteration
                  (norm-guards2 (append more-is-set conflict-guards)
                                new-guards nil
                                var-before nil
                                bind-s
                                bind-c nil))
                 (t
                  (list (append new-guards conflict-guards)
                        bind-s
                        (append bind-c nbind-c)
                        (append var-before var-add))))))
        (t
         (let ((term**tag nil)
               (term (rewrite (first guard-rest) bind-s))
               (all-bind-c (append nbind-c bind-c)))
           (setf term**tag (get-guardClas term (append var-before var-add) all-bind-c))
           (setf term (first term**tag))


           (case (second term**tag)
             (unknown ;;; apply UNP
              (list 'unknown bind-s bind-c var-before))

             (deno-prem   ;;; can be deleted because a foot is handled in function norm-chunk
                          ;;; apply DBE
              (norm-guards2 (rest guard-rest) new-guards conflict-guards
                            var-before var-add
                            bind-s bind-c nbind-c))

             (new-simple ;;; e.g. (is _x a) or (is _x _v) _x occurs first
                         ;;; apply DIP
```

```
 (norm-guards2 (rest guard-rest) new-guards conflict-guards
               var-before
               (cond ((vari-t (s-expr-is term))
                      (append var-add
                              (collect-new-vars (s-expr-is term)
                                                (append var-before var-add))))
                     (t var-add))
               (mk-bindings (s-patt-is term)
                            (s-expr-is term)
                            bind-s)
               bind-c nbind-c))

(new-complex ;;; e.g. (is _x `(f a)) _x occurs first (_y `(f a)) is in complex-bindings
             ;;;  apply DIP or CSE
 (let ((rhs-inst nil) (common-expr nil)
       (lhs-is (s-patt-is term))
       (rhs-is (un-inst (s-expr-is term))))
   (setf rhs-inst (rewrite rhs-is all-bind-c))
   (setf common-expr
         (rhs-match-p rhs-inst all-bind-c)) ;nil indicates no match, otherwise
                                            ;the variable (bind to rhs-inst) is returned

   (cond ((occur-check-p lhs-is rhs-inst) ; occur-check (-> apply SIU)
          (progn
            (error-handler 20 ; occur-check
                           term)
            (list 'unknown bind-s bind-c var-before)))

         (common-expr    ;;; applx CSE
          (norm-guards2 (rest guard-rest) new-guards conflict-guards
                        var-before
                        var-add
                        (mk-bindings lhs-is
                                     (first common-expr)
                                     bind-s)
                        bind-c nbind-c))

         (t ;;; a new complex structure  (apply DIP)
          (norm-guards2 (rest guard-rest) new-guards
                        (cons term conflict-guards)
                        var-before
                        (append var-add
                                (collect-new-vars term
                                                  (append var-add var-before)))
                        bind-s bind-c
                        (mk-bindings lhs-is
                                     rhs-inst
                                     nbind-c))))))

(unbound-simple ; e.g. (is _v _w); _v have no complex bindings and both do not occur
first
                ;;; apply DIP
 (let ((rhs-inst nil)
       (lhs-is (s-patt-is term))
       (rhs-is (s-expr-is term))
       (reduction (list (rest term)))) ; builts the reduction lhs -> rhs
   (setf rhs-inst (rewrite rhs-is all-bind-c))
   (cond ((occur-check-p lhs-is (rewrite rhs-is all-bind-c)) ; occur-check
          (progn
          (error-handler 20 ; occur-check
                         term)
          (list 'unknown bind-s bind-c var-before)))
         (t
          (norm-guards2 (rest guard-rest)
                        (cond ((member lhs-is var-before :test #'equal)
                               (cons term (rewrite new-guards reduction)))
                              (t (rewrite new-guards reduction)))
                        (rewrite conflict-guards reduction)
                        var-before var-add
                        (mk-bindings (s-patt-is term) (s-expr-is term)
                                     (rhs-rewrite lhs-is rhs-is bind-s))
                        (rhs-rewrite lhs-is rhs-inst bind-c)
                        (rhs-rewrite lhs-is rhs-inst nbind-c))))))

(unbound-complex ; e.g.  (is _v `(f a)) (apply DIP, CSE or SIU)
 (let ((rhs-inst nil)(common-expr nil)
       (lhs-is (s-patt-is term))
```

```
                (rhs-is (un-inst (s-expr-is term))))
              (setf rhs-inst (rewrite rhs-is all-bind-c))
              (cond ((occur-check-p lhs-is rhs-inst) ; occur-check (apply SIU and UNP)
                     (progn
                       (error-handler 20 ; occur-check
                                      term)
                       (list 'unknown bind-s bind-c var-before)))
                    (t
                     (setf common-expr (rhs-match-p rhs-inst all-bind-c))
                     (cond (common-expr ;(-> CSE)
                            (norm-guards2 (cons (mk-is lhs-is (first common-expr))
                                                (rest guard-rest))
                                          new-guards conflict-guards
                                          var-before var-add
                                          bind-s bind-c  nbind-c))
                           (t  ;;; add a new structure to nbind-c
                               ;;; -> DIP
                            (norm-guards2 (rest guard-rest)
                                          new-guards
                                          (cons term conflict-guards)
                                          var-before
                                          (append var-add
                                                  (collect-new-vars rhs-is
                                                                    (append var-add var-before)))
                                          bind-s
                                          (rhs-rewrite lhs-is rhs-inst bind-c)
                                          (mk-bindings lhs-is rhs-inst
                                                       (rhs-rewrite lhs-is rhs-inst nbind-c))))
                           ))
                    )))

            (complex-complex ;;; (rewrite term all-bind-c ==> (is `(..) `(..))
                             ;;; -> SIU
             (let ((error**is-set nil)  ;;; error=unkonwn, if unification fail; mgu defined by IS-SET
                   (lhs-inst (rewrite (s-patt-is term) all-bind-c))
                   (rhs-inst (rewrite (un-inst (s-expr-is term)) all-bind-c)))
               (setf error**is-set (mk-is-set lhs-inst rhs-inst))
               (norm-guards2 (cons (first error**is-set)
                                   (append (second error**is-set) (rest guard-rest)))
                             new-guards conflict-guards
                             var-before var-add
                             bind-s bind-c  nbind-c)))
            (otherwise
             (error "norm-guards"))
            )))
       ))



(defun norm-foot (term bindings-c)
  (let ((common-expr (rhs-match-p term bindings-c)))
    (cond (common-expr
           (first common-expr))
          (t term))
    ))


(defun get-guardClas (guard variables bindings-c)
"guard x VARIABLES x BINDINGS-C -> GUARD x TAG
       Returns a guard in a well manner with a description-tag
       A trivial guard likes (is a a) returns (a 'deno-prem) "

  (trace-in-handler 'get-guardClas (list (list 'guard guard) (list 'variables variables)
                                         (list 'bindings-c bindings-c)) nil)
  (cond ((eq 'unknown guard) ;;; -> UNP
         (error-handler 21 ; unknown-detection
                        guard)
         (trace-out-handler 'get-guardClas '(unknown unknown)))

        ((final-p guard)
        (trace-out-handler 'guardClass (list guard 'deno-prem)))

        (t ; the rhs-is is a denotative term!
         (let* ((lhs-is (s-patt-is guard))
                (rhs-is (un-inst (s-expr-is guard)))
                (lhs-var (vari-t lhs-is))
```

```
                (rhs-var (vari-t rhs-is))
                (lhs-inst (if lhs-var (rewrite lhs-is bindings-c)
                              lhs-is))
                (rhs-inst (if rhs-var (rewrite rhs-is bindings-c)
                              rhs-is)))

          (cond ((equal lhs-inst rhs-inst)  ; Trivial Is-Term (-> SIU)
                 (trace-out-handler 'get-guardClas (list (s-expr-is guard)
                                                         ;(if (eq 'unknown lhs-is)
                                                          ; 'unknown
                                                           'deno-prem;)
                                                          )))

                ((and (or (con-t lhs-is)(con-t rhs-is))  ; Trivial unknown dedection
                      (and (not (eq lhs-is rhs-is))
                           (not (vari-t lhs-inst))
                           (not (vari-t rhs-inst))))
                 (error-handler 22 ; unknown-generation
                                guard)
                 (trace-out-handler 'get-isClas '(unknown unknown)))

                ; Example (is _x a) + _x not in variables  -> (is _x a) + new-simple
                ((and lhs-var (not (member lhs-is variables :test #'equal))) ; the variable occurs
first
                 (trace-out-handler 'get-guardClass (list guard (if (convar-p rhs-is)
                                                                    'new-simple ; then
                                                                    'new-complex)))) ; else

                ; Example (is (f a) _x) + _x not in variables -> (is _x `(f a)) + new-complex
                ((and rhs-var (not (member rhs-is variables :test #'equal)))
                 (trace-out-handler 'get-guardClass (list (mk-is rhs-is (mk-inst lhs-is))
                                                          (if (convar-p lhs-is)
                                                            'new-simple ; then
                                                            'new-complex)))) ; else

                ; Is primitive without new variables and structures
                ((and (convar-p lhs-inst)   ; _v, _w in variables (or constants)
                      (convar-p rhs-is))    ; + no structure
                 (trace-out-handler 'get-guardClass (list (if lhs-var
                                                              guard
                                                              (mk-is rhs-is lhs-is))
                                                          'unbound-simple)))

                ((and (convar-p lhs-is)   ; _v, _w in variables (or constants)
                      (convar-p rhs-inst))    ; + no structure
                 (trace-out-handler 'get-guardClass (list (if rhs-var
                                                              (mk-is rhs-is lhs-is)
                                                              guard)
                                                          'unbound-simple)))

                ((convar-p lhs-inst)
                 (trace-out-handler 'get-guardClass (list guard 'unbound-complex)))

                ((convar-p rhs-inst)
                 (trace-out-handler 'get-guardClass (list (mk-is rhs-is (mk-inst lhs-is))
                                                          'unbound-complex)))


                (t
                 (trace-out-handler 'get-guardClass (list guard 'complex-complex)))
                )))
        ))



(defun rewrite (term bindings) "TERM x BINDINGS -> TERM"
  (cond ((null term ) nil)
        ((con-t term)
         term)
        ((vari-t term )
         (s-binding term bindings))
        (t
         (cons (rewrite (first term) bindings)
               (rewrite (rest term) bindings)))))
```

```
;;;      rhs-rewrite

(defun rhs-rewrite (lhs rhs bindings) "VAR x TERM x BINDINGS -> BINDINGS
      Replacing all occurrence of 'lhs' on the right-side in 'bindings'
      to the term denoted by 'rhs'"
   (trace-in-handler 'rhs-rewrite (list (list 'lhs lhs) (list 'rhs rhs) (list 'bindings bindings))
nil)
  (cond ((null bindings)
         nil)
        (t
         (let ((binding-first (first bindings))
               (binding-rest (rest bindings)))
           (cons (list (first binding-first)
                       (rewrite (second binding-first) (list (list lhs rhs))))
                 (rhs-rewrite lhs rhs binding-rest))))))

(defun rewrite-with-isset (term is-set)
  (rewrite term (remove-is-from-isset is-set)))

(defun remove-is-from-isset (is-set)
  (mapcar #'remove-is is-set))

(defun remove-is (is-term) (remove 'is is-term))

(defun sort-head-guards (guards head-vars)
   (sort-guards guards  nil head-vars))

(defun sort-guards (guards guards-new variables)
  (cond ((null guards) guards-new)
        (t (let ((gl1 nil)
                 (gl2 nil)
                 (gl1**gl2 (split-guards guards variables)))
             (setf gl1 (first gl1**gl2)
                   gl2 (second gl1**gl2))
           (cond ((null gl1)
                  (sort-guards (rest gl2) (append guards-new
                                                  (list (first gl2)))
                               (collect-new-vars (first gl2) variables)))
                 (t
                  (sort-guards gl2 (append guards-new gl1)
                               (collect-new-vars gl1 variables))))
          ))))

(defun split-guards (guards variables)
  (split-guards2 guards nil nil variables))

(defun split-guards2 (guards gl1 gl2 variables)
  (cond ((null guards) (list gl1 gl2))
        (t (let ((guard (first guards)))
             (cond ((and (is-t guard)
                         (member (s-patt-is guard) variables :test #'equal))
                    (split-guards2 (rest guards)
                                   (cons guard gl1)
                                   gl2 variables))
                   (t
                    (split-guards2 (rest guards)
                                   gl1
                                   (cons guard gl2)
                                   variables)))))
        ))


;;;;
;;;;
;;;;     SELECTORS and COLLECTORS
;;;;
;;;;

(defun collect-guards (premises)
  "function collect-guards: premises -> guards x premises
   collects the guards for the next chunk.
   The guards are returned in reverse order."
  (collect-guards2 nil premises))

(defun collect-guards2 (guard-list premises)
  (cond ((null premises)
         (list guard-list premises))
```

```
        (t
         (let ((prem (first premises)))
           (cond ((final-p prem)
                  (collect-guards2 (cons prem guard-list)
                                   (rest premises)))
                 ((and (is-t prem)
                       (final-p (s-expr-is prem)))
                  (collect-guards2 (cons prem guard-list)
                                   (rest premises)))
                 (t
                  (list guard-list premises)))))
        ))

(defun collect-deno-is (premises)
"PREMISES -> IS-SET x PREMISES
      The returned IS-SET is in the reverse order
      First terms in IS-SET are simple-is-primitives as (is _x a), where both sides
      are Variables or constants"

  (let ((isList**isList**premises (collect-deno-is2 nil nil premises)))
   (list (append (first isList**isList**premises)
                 (second isList**isList**premises))
         (third isList**isList**premises))))

(defun collect-deno-is2 (cv-is ot-is prem)
;;;
;;; collects the next denotative is-literals. 'cv-is' means that only constants or variables
;;; occur in the is-literal
;;;
  (let ((term (first prem))
        (term-rest (rest prem)))

    (cond ((null term-rest)
           (list cv-is ot-is prem))

          (t
                  ;;; denotative literal
           (cond ((final-p term)
                  (collect-deno-is2 cv-is ot-is term-rest))

                 ;;; denotative is-literal
                 ((and (is-t term)
                       (final-p (s-expr-is term)))
                  (let* ((lhs-is (s-patt-is term))
                         (rhs-is (s-expr-is term))
                         (lhs-cv (convar-p lhs-is))
                         (rhs-cv (convar-p rhs-is)))

                    (cond ((and lhs-cv rhs-cv)
                           (collect-deno-is2 (cons term cv-is) ot-is term-rest))

                          (t (collect-deno-is2 cv-is (cons term ot-is) term-rest)))))

                 ;;; evaluative literal
                 (t
                  (list cv-is ot-is prem)))))))


(defun collect-vars (term variables)
  "TERM x VARS -> VARS
  collects all variables of term and returns the union of the new variables and VARIABLES"

   (cond ((null term) variables)
         ((atom term) variables)
         ((and (vari-t term) (not (member term variables :test #'equal)))
          (cons term variables))
         (t (collect-vars (rest term) (collect-vars (first term) variables)))))


(defun collect-new-vars (term variables)
  "TERM x VARS -> VARS
  collects all variables of term and returns only the new variables"

   (cond ((null term) nil)
         ((atom term) nil)
         ((and (vari-t term) (not (member term variables :test #'equal)))
```

```
          (cons term nil))
         (t
          (let ((new-vars (collect-new-vars (first term) variables)))
            (append new-vars
                    (collect-new-vars (rest term) (append new-vars variables)))))
        ))


(defun s-body-foot (term-list) " LIST -> LIST x TERM"
  (let ((rev-list (reverse term-list)))
    (list (reverse (rest rev-list))
          (first rev-list))))


(defun s-clause-kind (cl-info) "CL-INFO -> CLAUSE-TAG"
   (first cl-info))

(defun s-cl-info-kind (cl-info)
    (first cl-info))

(defun s-clause-variables (cl-info) "CL-INFO -> VAR-LIST"
   (second cl-info))

(defun s-cl-info-variables (cl-info)
  (second cl-info))

(defun s-simple-bindings (cl-info) "CL-INFO -> BINDINGS-S"
   (third cl-info))


(defun s-complex-bindings (cl-info) "CL-INFO -> BINDINGS-C"
   (fourth cl-info))


(defun s-binding (variable bindings) "VAR x BINDINGS -> TERM"
  (let ((var-binding (assoc variable bindings :test #'equal)))
    (cond (var-binding
           (second var-binding))
          (t
           variable))))

(defun s-chunk-foot (guards)
  (let ((ft-guard (first guards)))
    (cond ((is-t ft-guard)
           (s-patt-is ft-guard))
          (t ft-guard))))

;;;;
;;;;
;;;;    CONSTRUCTORS and MODIFIERS
;;;;
;;;;

(defun mk-variables-cl-info (var-list cl-info) "VARS x CL-INFO -> CL-INFO"
   (mk-cl-info (s-clause-kind cl-info)
               var-list
               (s-simple-bindings cl-info)
               (s-complex-bindings cl-info)))

(defun mk-proc-name (clause) "CLAUSE -> PROCEDURE-NAME"
   (list (first (s-conclusion clause))
         (length (rest (s-conclusion clause)))))

(defun mk-clause (kind head body-list &optional (foot 'no-foot))
  (cons kind
        (append (cons head body-list)
                (cond ((eq foot 'no-foot) nil)
                      (t (list foot))))))

(defun mk-bindings (lhs rhs bindings)
            (cons (list lhs rhs) bindings))

(defun mk-is-set (arg-list1 arg-list2) "ARG-LIST x ARG-LIST -> IS-SET
      IS-SET = ( (is ...) ... (is ...)) "
  (cond ((null arg-list1) '(true nil))
        ((eql (length arg-list1) (length arg-list2))
         (list 'true (mk-is-set2 arg-list1  arg-list2)))
```

```
        (t '(unknown nil))))

(defun mk-is-set2 (l1 l2)
  (cond ((null l1) nil)
        (t (cons (mk-is (first l1) (mk-inst (first l2)))
              (mk-is-set2 (rest l1) (rest l2))))))

(defun mk-clause-info (kind variables simple-bindings complex-bindings)
  (list kind variables simple-bindings complex-bindings))

(defun mk-inst (term)
  (cond ((convar-p term) term)
        (t (cons 'inst (cons term nil)))))

(defun mk-cl-info (kind vars bindings-s binding-c)
  (list kind vars bindings-s binding-c))

(defun mk-isset-from-complex-bindings (bindings)
;;; Die komplexe Bindungsumgebung wird durch die simple Bindung lhs -> rhs aktualisiert
;;; Hierbei können rechte Seiten gleich werden, dies führt zu neuen is-Primitiven und zur
;;; Löschung der entsprechenden komplexen Regel aus der Datenbasis

(mk-isset-from-complex-bindings2 bindings nil nil))

(defun collect-is-set-from-new-bindings (bind-c nbind-c)
  (second (mk-isset-from-complex-bindings2 nbind-c bind-c nil)))


(defun mk-isset-from-complex-bindings2 (old-bindings new-bindings is-set)
  (cond ((null old-bindings)
         (list new-bindings is-set))
        (t
         (let* ((rest-bindings (rest old-bindings))
                (next-rule (first old-bindings))
                (lhs-rule (first next-rule))
                (rhs-rule (second next-rule))
                (common-expr (rhs-match-p rhs-rule new-bindings)))
           (cond (common-expr ;; there is the same rigth size
                  (mk-isset-from-complex-bindings2 rest-bindings
                                                   new-bindings
                                                   (cons (mk-is lhs-rule
                                                                (first common-expr))
                                                         is-set)))
                 (t
                  (mk-isset-from-complex-bindings2 rest-bindings
                                                   (cons next-rule new-bindings)
                                                   is-set)))))))


(defun add-conflict-guards  (guard guard-list)
  (reverse (add-conflict-guards2 guard (reverse guard-list))))

(defun add-conflict-guards2  (guard guard-list)
  (cond ((null guard-list)
         (list guard))
        ((is-t guard)
         (let ((lhs (s-patt-is guard))
               (first-guard (first guard-list)))
           (cond ((member lhs (collect-variables first-guard) :test #'equal)
                  (cons guard guard-list))
                 (t (cons first-guard
                          (add-conflict-guards2  guard (rest guard-list))))
                 )))
        (t (cons guard guard-list))
        ))


;;;;
;;;;
;;;;    PREDICATES
;;;;
;;;;

(defun con-t (term) (atom term))

(defun rhs-match-p (rhs-term bindings) "TERM x BINDINGS -> BINDING"
   (cond ((null bindings) nil)
```

```
         (t
          (let ((bind-rule (first bindings)))
            (cond ((equal (second bind-rule) rhs-term)
                   bind-rule)
                  (t
                   (rhs-match-p rhs-term (rest bindings)))))))))


(defun occur-check-p (x term)
  (cond ((null term) nil)
        ((equal x term) t)
        ((atom term) nil)
        ((occur-check-p x (first term))
         t)
        (t
         (occur-check-p x (rest term)))))

(defun lispcall-inst-p (guard bindings-c)
  (cond ((is-t guard)
         (let ((functor (first (s-expr-is guard))))
           (and (or (lisp-function-p functor)
                    (lisp-predicate-p functor))
                (null (collect-variables (rewrite (s-expr-is guard) bindings-c))))))
        (t
         (let ((functor (first guard)))
           (and (or (lisp-function-p functor)
                    (lisp-predicate-p functor))
                (null (collect-variables (rewrite guard bindings-c))))))
       ))
```

A.2 Datei "debug.lisp"

```
(defvar *tracer-level* 0 "default tracer-level is: no trace")
(defvar *error-level* 0 "default error-level is: ignore errors")
(defvar *error-stream* nil "post error-messsage to this stream if *error-level* = 4.
                            Otherwise the current clause stays here")

(defun set-trace (level) (declare (specials *tracer-level*)) (setq *tracer-level* level))

;;;
;;;     ******** trace-in-handler ********
;;;

(defun set-err (level) (declare (specials *error-level*))
       (setq *error-level* level))

(defun trace-in-handler (func-name args environment &optional (level *tracer-level*))
  (cond ((or (not (numberp level))
             (<= level 0)
             (> level 2))
         level)
        (t
         (progn (terpri) (princ "Observe Function:")
                (princ func-name) (terpri)
                (if (eql level 2)
                  (progn (princ (documentation func-name 'function)) (terpri)))
                (trace-args args)
                (trace-env environment)))))

;;;
;;;     ******** trace-out-handler ********
;;;

(defun trace-out-handler (func-name value &optional (level *tracer-level*))
  (cond ((or (not (numberp level))
             (<= level 0)
             (> level 2))
         value)
        (t (progn (terpri) (princ "*** ") (princ func-name) (princ " *** returns:")
                  (rf-pprint value)
                  value))))


(defun trace-args (argvalue-list)
  (cond ((null argvalue-list) nil)
        (t
         (progn (princ "   *** Arguments ***") (terpri)
                (mapcar #'trace-name-value argvalue-list)
                nil))))


(defun trace-env (argvalue-list)
  (cond ((null argvalue-list) nil)
        (t
         (progn (princ "    *** Evironment ***") (terpri)
                (mapcar #'trace-name-value argvalue-list)
                nil))))

(defun trace-name-value (name-value)
  (rf-pprint (first name-value)) (princ " :")
  (rf-pprint (second name-value))
  (terpri))



;;;
;;;       ********   error-handler  *********
;;;

(defun error-handler (err-no error-args) ; hanldes warnings also
  (declare (specials *error-level* *error-stream*))
```

```
  (cond ((or (<= *error-level* 0) (> *error-level* 4))
         t)
        (t
         (let* ((proc-name**clause (first *error-stream*))
                (proc-name (first proc-name**clause))
                (clause (second proc-name**clause)))
           (case err-no ;
             (20 ; occur-check
              (case *error-level*
                ((1 2) ; print on screen
                 (print-error "Occur-check has failed"
                              proc-name clause error-args t)
                 (if (eql *error-level* 2) (break)))
                (4 ; print on *error-stream*
                 (print-error "Occur-check has failed"
                              proc-name clause error-args nil))))

             (21 ; unknown-detection
              (case *error-level*
                ((1 2) ; print on screen
                 (print-error "Unknown-detection"
                              proc-name clause error-args t)
                 (if (eql *error-level* 2) (break)))
                (4 ; print on *error-stream*
                 (print-error "Unknown-detection"
                              proc-name clause error-args nil))))
             (22 ; unknown-generation
              (case *error-level*
                ((1 2) ; print on screen
                 (print-error "Unknown-generation"
                              proc-name clause error-args t)
                 (if (eql *error-level* 2) (break)))
                (4 ; print on *error-stream*
                 (print-error "Unknown-generation"
                              proc-name clause error-args nil)))))))))

(defun print-error (err-mess proc-name clause err-args on-screen)
  (declare (specials *error-level* *error-stream*))
  (cond (on-screen
         (terpri) (princ "***** WARNING *****") (terpri)
         (princ err-mess) (terpri)
         (princ "In procedure: " ) (princ proc-name) (terpri)
         (princ "In clause: ") (terpri) (rf-pprint clause)
         (princ "While normalizing: ") (rf-pprint err-args))
        (t
         (setq *error-stream*
               (cons
                (list err-mess
                      "In procedure:"  proc-name
                      "In clause: " clause
                      "While normalizing: " err-args)
                (s-rest *error-stream*))))))
```

Anhang B: Programmlisting des *mode-Interpreters*

B.1 Datei "mode-interpreter.lisp"

```
;;;;;
;;;;;   A mode interpreter for automatically generating procedure modes
;;;;;   in a RELFUN program
;;;;;
;;;;;  Thomas Krause, DFKI Kaiserslautern
;;;;;  20. March 1991


(defun mode-interpret (query db)
"QUERY x DATABASE -> ExtTable
Computes an approximation of the mode for all predicates in the database db
relative to the abstract query. The database has to be normalized by using
the command HORIZON on the RELFUN toplevel loop. The function mode-interpret
gives back the extension table, which describes the runtime dataflow."

  (let ((ExtTable (create-ExtTable))
        (fun2 nil))
    (setf ExtTable (index-db db ExtTable))
    (setf ExtTable (compute-aliasesBit db ExtTable))
    (setf fun2 (md-interp-goal (mk-litname query)
                               (rest query)
                               nil ;No recursion in the initial query
                               nil ;No call-aliasing in the initial query
                               ExtTable
                               db))
     (second fun2)))


(defun md-interp-clauseList (pname callp index-list c-ali ExtTable db)
"PNAME x CALLP x INDEX-LIST x C-ALI x ExtTable x DB
        -> ExtTable"
  (cond ((null index-list)
         (let ((procEntry (s-procEntry pname ExtTable)))
           (cond ((null procEntry)
                  (progn
                    (princ "*** Warning ***")
                    (princ "there is no clause definition for >>")
                    (princ (first pname)) (princ "/") (princ (second pname))
                    (princ "<<") (terpri))
                  (setf procEntry (mk-procEntry pname nil nil nil))
                  (setf ExtTable (add-procEntry procEntry  ExtTable))))

                 ;;; repeated call (with CALLP) to a procedure with no definition in DB
           (cond ((exist-succpatEntry-p callp procEntry)
                  ExtTable)

                 ;;; first call (with CALLP) to a procedure with no definition in DB
                 (t
                  (modify-procEntry
                     pname
                     (modify-succpEntry callp
                                        (mk-noDef-succpat callp); compute safe approximation
                                        'close procEntry)
                     ExtTable))
                 )))

         ;;; call to a procedure which is defined in DB
        (t
         (md-interp-clauseList2 callp index-list c-ali ExtTable db))
        ))

(defun md-interp-clauseList2 (callp index-list c-ali ExtTable db)
  "CALLP x INDEX-LIST x C-ALI x ExtTable x DB
        -> ExtTable"
  (cond ((null index-list)
         ExtTable)
```

```
        (t
         (let ((rest-list (rest index-list)))
           (md-interp-clauseList2 callp rest-list c-ali
                                  (md-interp-clause callp
                                                    (nth (first index-list) db)
                                                    (null rest-list)
                                                    c-ali
                                                    ExtTable db)
                                  db)))
        ))

(defun md-interp-clause (callp clause lastcl c-ali ExtTable db)
  "CALLP x CLAUSE x LASTCLx C-ALI x ExtTable x DB
        -> ExtTable"
  (let ((i-state nil) (aliasing nil)
        (f-state**val**ET nil) ;final state, return value, extension table
        (succp nil)
        (cl-kind (s-kind clause))
        (cl-head (s-conclusion clause))
        (cl-premises (s-premises clause)))


    (setf aliasing (or c-ali                              ; call-aliasing
                       (repeated-variable-p cl-head))) ; return-aliasing
    ;;; computing the initial instantion state
    (setf i-state (safe-istate cl-head callp nil aliasing))

    ;;; computing the final instantiation state and the return value state
    (setf f-state**val**ET (md-interp-goallist (mk-pname clause)
                                               cl-premises
                                               i-state 'free
                                               lastcl ExtTable db))

    ;;; computing the success pattern for CLAUSE with respect to CALLP
    (setf succp (get-succp (rest cl-head) (first f-state**val**ET)
                           (cond ((eq cl-kind 'ft)
                                  (second f-state**val**ET))
                                 (t 'true))))

    ;;; computing the new extension table and give back as the result
    (add-succp-to-table (mk-pname clause) callp succp
                        (third f-state**val**ET))))



(defun md-interp-goallist (pname premises i-state val lastcl ExtTable db)
"PNAME x PREMISES x I-STATE x VAL x LASTCL x ExtTable x DATABASE
                          -> I-STATE x VAL x ExtTable"
  (cond ((null premises)
         (list i-state val ExtTable))
         (t
          (let ((goal (first premises)))

            ;;; the goal is a denotative literal and by definition in foot position
            (cond ((final-p goal)
                   (list i-state
                         (term-state (project-state goal i-state))
                         ExtTable))

                  ;;; the goal is a kind of LISP builtin
                  ((lispcall-p goal)
                   (md-interp-goallist pname
                                       (rest premises)
                                       (safe-istate goal
                                                    (mk-lsp-succpat (s-termargs
                                                                     (project-state goal i-state)))
                                                    i-state
                                                    nil) ; calling LISP cause no aliasing effects
                                       'closed  ;;; LISP builtin default return value
                                       lastcl
                                       ExtTable db))

                  ;;; the goal is a kind of an is primitive
                  ((is-t goal)
```

```
                (md-interp-is pname (rest premises)
                              goal i-state lastcl ExtTable db))

              ;;; an evaluative (normal) goal
              (t
               (let ((kof-rec ; kind of allowed recursion
                      (cond ((and lastcl                    ; Only the last clause is recursiv?
                                  (equal pname (mk-litname goal)) ; It is not a indiect recursion?
                                         (null (rest premises))) ; It is a tail recursion?
                             'safe-rec)                          ; then it is a safe recursion!
                            (t 'unsafe-rec)))
                     (succp**newET nil) ;;; succp, new extension table
                     (lit-activation (rest (project-state goal i-state))))

                 (setf succp**newET (md-interp-goal (mk-litname goal)
                                                    lit-activation
                                                    kof-rec
                                                    (call-aliasing-p goal)
                                                    ExtTable
                                                    db))
                 (md-interp-goallist pname  (rest premises)
                                     (safe-istate goal (first succp**newET)
                                                  i-state
                                                  nil);No use for c-aliasing information here
                                     (first (last (first succp**newET)))
                                     lastcl
                                     (second succp**newET)
                                     db)))
              )))
        ))


(defun md-interp-is (pname rest-premises is-term i-state lastcl ExtTable db)
"PNAME x REST-PREMISES x IS-TERM x I-STATE x LASTCL x ExtTable x DATABASE
                        -> I-STATE x VAL x ExtTable"
  (let ((new-istate nil)
        (val nil)
        (istate**retset**ExtTable nil)
        (lhs-term (s-patt-is is-term))
        (rhs-term (s-expr-is is-term)))

          ;;; Denotative is-Literal
    (cond ((final-p rhs-term)
           (cond ((inst-t rhs-term)
                  (setf new-istate (struc-unify lhs-term rhs-term i-state))
                  (setf val (s-var-state lhs-term new-istate)))
                 (t
                  (setf val (s-unify (project-state lhs-term i-state)
                                     (project-state rhs-term i-state)))
                  (setf new-istate (add-istate lhs-term val i-state)))
                 ))

          ;;; evaluative is-Literal
          (t
           (setf istate**retset**ExtTable
                 (md-interp-goallist pname (list rhs-term)
                                     i-state
                                     'empty    ;; No information about the return value available
                                     (cond (lastcl 'lcl-is); last clause and is primitive
                                           (t lastcl))
                                     ExtTable db))

           ;;; handling the VALue of the abstract goal given by RHS-TERM
           (cond ((vari-t lhs-term)
                  (setf val (s-unify (second istate**retset**ExtTable)
                                     (s-var-state lhs-term (first istate**retset**ExtTable))))
                  (setf new-istate (add-istate lhs-term
                                               val i-state)))

                 (t; lhs-term is a constant
                  (setf val (s-unify (second istate**retset**ExtTable)
                                     (project-state lhs-term i-state))
                        new-istate (first istate**retset**ExtTable)))
                 ))
          )
```

```
    (md-interp-goallist pname  rest-premises new-istate
                        val lastcl
                        (cond ((final-p rhs-term) ExtTable)
                              (t (third istate**retset**ExtTable)))
                        db)))



(defun md-interp-goal (pname callp kof-rec c-ali ExtTable db)
  "PNAME  x  CALLP x KOF-REC x C-ALI x ExtTable x DATABASE
                        -> succp x ExtTable"
  (let ((newExtTable nil)
        (procEntry (s-procEntry pname ExtTable)))
    (cond ((succpat-close-p callp procEntry) ; succes pattern is allready computed
           (list (s-succpat callp procEntry) ExtTable))

          ((succpat-open-p callp procEntry) ;a recursive call with the same call pattern
           (cond ((eq kof-rec 'safe-rec)
                  (let ((new-callp nil)
                        (current-succp (s-succpat callp procEntry)))
                    (setf new-callp (reverse (rest (reverse current-succp))))

                    (cond ((or (not new-callp)           ; there exist no succp
                               (equal new-callp callp))  ; a recursive call gives no more information
                           (mk-succpat-close pname callp kof-rec ExtTable))

                          (t
                           (md-interp-goal pname new-callp nil c-ali ExtTable db))
                          )))

                 (t
                  (mk-succpat-close pname callp kof-rec ExtTable))))

          (t; the first call to procedure PNAME with call pattern CALLP

           ;;; try all possible clauses with these call pattern
           (setf newExtTable
                 (md-interp-clauseList pname callp
                                       (s-indexList procEntry) c-ali
                                       (cond ((null procEntry); call to extern procedure
                                              ExtTable)
                                             (t
                                              (mk-succpat-open pname callp ExtTable)))
                                       db))

           ;;; No further need to compute these goal with the same call pattern!
           (mk-succpat-close pname
                             callp
                             'goal-finished
                             newExtTable))
          )))


(defun index-db (db ExtTable)
"DATABASE ExtTable -> ExtTable"
  (index-db2 (reverse db)
             (1- (length db)) ; the first clause has the index 0!
             ExtTable))


(defun index-db2 (db clause-index ExtTable)
"DATABASE x CLAUSE-INDEX x ExtTable  ->  ExtTable"
  (cond ((null db) ExtTable)
        (t
         (let ((clause (first db))
               (db-rest (rest db))
               (pname nil)(procEntry nil))
           (setf pname (mk-pname clause)
                 procEntry (s-procEntry pname ExtTable))
           (setf ExtTable
                 (cond ((null procEntry) ; there is no entry for the procedure yet
                        (add-procEntry (mk-procEntry pname nil nil
                                                     (list clause-index))
                                       ExtTable))

                       ((recursive-call-p clause)
                        (modify-procEntry pname
```

```
                                                (mk-procEntry pname nil nil
                                                              (append (s-indexList procEntry)
                                                                      (list clause-index)))
                                                ExtTable))

                        (t
                         (modify-procEntry pname
                                           (mk-procEntry pname nil nil
                                                         (cons  clause-index
                                                                (s-indexList procEntry)))
                                           ExtTable))
                        ))
            (index-db2 db-rest (1- clause-index) ExtTable)))
        ))



(defun compute-aliasesBit (db ExtTable)
"DATABASE x ExtTable  ->  ExtTable"
  (let ((aliases-list nil)
        (new-ExtTable (return-aliasesBit db ExtTable)))
    (setf aliases-list (collect-aliasesBit new-ExtTable))
    (call-aliasesBit db aliases-list (null aliases-list) new-ExtTable)))


(defun return-aliasesBit (db ExtTable)
  (cond ((null ExtTable) nil)
        (t
         (let ((pname nil) (i-list nil)
               (procEntry (first ExtTable)))
           (setf pname (first procEntry)
                 i-list (s-indexList procEntry))
           (cond ((return-aliases-p i-list db)
                  (cons (mk-procEntry pname
                                      t            ; ALI-BIT!
                                      nil          ; SUCCPAT-TABLE
                                      i-list)
                        (return-aliasesBit db (rest ExtTable))))
                 (t
                  (cons procEntry
                        (return-aliasesBit db (rest ExtTable))))
                 )))
        ))


(defun call-aliasesBit (db ali-list endloop ExtTable)
;;; ENDLOOP indicates no chnage of the ALI-BIT while the last iteration
  (cond (endloop ExtTable)
        (t (let ((more-ali-list nil)
                 (copy-ExtTable (copy-list ExtTable)))
             (dolist (procEntry ExtTable) ; for all predicates do
               (cond ((or (aliases-p procEntry)
                          (member (first procEntry) more-ali-list :test #'equal))
                      t); nothing to do becuse the predicates is allready tagged

                     ((call-aliases-p (s-indexList procEntry)
                                      (append more-ali-list ali-list)
                                      db)
                      (setf more-ali-list (cons (first procEntry) more-ali-list)
                            copy-ExtTable (modify-procEntry (first procEntry)
                                                            (set-aliasesBit procEntry)
                                                            copy-ExtTable)))
                     ))
             (call-aliasesBit db (append more-ali-list ali-list)
                              (null more-ali-list) ; Was there a change in the ALI-BIT?
                              copy-ExtTable)
             ))
        ))


(defun term-state (lit-activation)
"LIT-ACTIVATION ->  TERM-STATE"
  (cond ((atom lit-activation) lit-activation)
        ((inclusion-p 'closed (lub (s-termargs lit-activation)))
         'closed)
        (t 'dontknow)
```

```
        ))

(defun safe-istate (literal succp i-state aliasing)
"LITERAL x SUCCP x I-STATE x ALIASING
                                                                    ->  I-STATE"
  (safe-istate2 (rest literal) succp
                      aliasing
                      i-state))

(defun safe-istate2 (arglist succp aliBit i-state)
  (cond ((null arglist)  i-state)
        (t
        (let ((var-state nil)
              (arg (first arglist)))
          (cond ((vari-t arg)
                 (setf var-state
                       (s-unify (s-var-state arg i-state)
                                (first succp)))
                 (cond ((and
                          aliBit ; indicates return-aliasing
                          (eq var-state 'free))
                         (setf var-state 'dontknow)))
                 (safe-istate2 (rest arglist)
                               (rest succp) aliBit
                               (add-istate arg var-state i-state)))
                (t
                 (safe-istate2 (rest arglist) (rest succp) aliBit i-state))
                )))
        ))


(defun lub (term-state-list)
"TERM-STATE-LIST  ->  TERM-STATE"
  (lub2 term-state-list 'empty))

(defun lub2 (tsl ts)
  (cond ((null tsl) ts)
        (t (lub2 (rest tsl)
                 (get-lub (first tsl) ts)))
        ))

(defun pattern-lub (pat1 pat2)
"PATTERN x PATTERN  ->  PATTERN"
  (cond ((and (null pat1)
              (null pat2))
         nil)
        (t (cons (get-lub (first pat1) (first pat2))
                 (pattern-lub (rest pat1) (rest pat2))))
        ))


(defun get-lub (ts1 ts2)
  (cond ((eq ts1 ts2) ts1)

        ((or (eq ts1 'dontknow)
             (eq ts2 'dontknow))
         'dontknow)

        ((or (and (eq ts1 'closed) (eq ts2 'true))
             (and (eq ts1 'true) (eq ts2 'closed)))
         'closed)

        ((or (eq ts1 'empty)
             (null ts1))
         ts2)

        ((or (eq ts2 'empty)
             (null ts2))
         ts1)

        ((or (eq ts1 'free)
            (eq ts2 'free))
           'dontknow)

        (t (error "calling get-lub with wrong arguments"))
        ))
```

```
(defun conser-approx (callp)
  " CALLP -> SUCCP"
  ;;; computes a conservative approximation for the callp, for instance because
  ;;; of a recursive call
  (cond ((null callp)
         '(dontknow)) ;;; Approximation of the return term state VAL

        ((eq 'free (first callp))
         (cons 'dontknow (conser-approx (rest callp))))

        (t
         (cons (first callp) (conser-approx (rest callp))))
        ))

(defun consvalue-approx (callp succp)
  (cond ((eq (length callp) (length succp))
         (reverse (cons 'dontknow (reverse succp))))
        (t
         (reverse (cons 'dontknow (rest (reverse succp)))))))


(defun s-unify (ts1 ts2)
"TERM-STATE x TERM-STATE  ->  TERM-STATE "
  (cond ((or (eq ts1 'empty)
             (eq ts2 'empty))
         'empty)
        ((or (eq ts1 'true)
             (eq ts2 'true))
         'true)
        ((or (eq ts1 'closed)
             (eq ts2 'closed))
         'closed)
        ((or (eq ts1 'dontknow)
             (eq ts2 'dontknow))
         'dontknow)
        (t 'free)
        ))

(defun struc-unify (var struc i-state)
"VARIABLE x STRUCTURE x i-state  ->  i-state"
  (let ((new-istate i-state)
        (var-state (s-var-state var i-state))
        (term-state (term-state (project-state struc i-state))))

    (cond ((eq 'closed term-state)
           (add-istate var 'closed i-state))

          ((or (eq 'closed var-state)
               (eq 'true var-state)  ;;; maybe, an error could generated!!!
               )
           (dolist (v (collect-new-vars struc nil))
             (setf new-istate (add-istate v 'closed new-istate)))
           new-istate)

          ((eq var-state 'free)
           (add-istate var 'dontknow i-state))

          (t
           (dolist (v (collect-new-vars struc (list var)))
             (setf new-istate (add-istate v 'dontknow new-istate)))
           new-istate)
          )))



(defun project-state (literal i-state)
"LITERAL x I-STATE -> LIT-ACTIVATION
   Computes the activation of the literal (instantiation pattern) by
   projecting each i-state of an argument into the literal"

  (cond ((eq literal 'true) 'true)
        ((atom literal) 'closed)
        ((vari-t literal) (s-var-state literal i-state))
        (t
         (cons (s-functor literal)
               (project-state2 (s-termargs literal) i-state)))
```

```
      ))

(defun project-state2 (arglist i-state)
;;; Remember, only flat clauses belong to the relfun kernel. This means that
;;; the arglist is always flat
  (cond ((null arglist) nil)
        (t
         (let ((arg (first arglist)))
           (cons
            (cond ((eq arg 'true) 'true)
                  ((atom arg) 'closed)
                  ((vari-t arg)
                   (let ((var-state (s-var-state arg i-state)))
                     (cond ((and nil (eq var-state 'free)
                                 (member arg
                                         (rest arglist)
                                         :test #'equal)); testing call-aliasing
                            (setf i-state (add-istate arg 'dontknow i-state))
                           'dontknow)
                           (t var-state))))
                  (t (error "No flat arglist!!!")))
            (project-state2 (rest arglist) i-state))))
      ))



;;;;;
;;;;;
;;;;;            SELECTORS and COLLECTORS
;;;;;
;;;;;

(defun s-procEntry (pname ExtTable)
  (assoc pname ExtTable :test #'equal))

(defun s-succpat-table (procEntry)
  (third procEntry))

(defun s-succpat-entry (callp procEntry)
  (assoc callp (s-succpat-table procEntry) :test #'equal))

(defun s-succpat (callp procEntry)
  (second (s-succpat-entry callp procEntry)))

(defun s-succpat-state (callp procEntry)
  (third (s-succpat-entry callp procEntry)))

(defun s-indexList (procEntry)
  (fourth procEntry))

(defun s-aliasesBit (procEntry)
  (second procEntry))

(defun collect-aliasesBit (ExtTable)
  (cond ((null ExtTable) nil)
        ((aliases-p (first ExtTable))
         (cons (first (first ExtTable))
               (collect-aliasesBit (rest ExtTable))))
        (t (collect-aliasesBit (rest ExtTable)))))

(defun collect-new-vars (term variables)
  "TERM x VARS -> VARS
  collects all variables of term and returns only the new variables"

   (cond ((null term) nil)
         ((atom term) nil)
         ((and (vari-t term) (not (member term variables :test #'equal)))
          (cons term nil))
         (t
          (let ((new-vars (collect-new-vars (first term) variables)))
            (append new-vars
                    (collect-new-vars (rest term) (append new-vars variables)))))
         ))


(defun s-var-state (variable i-state)
"VARIABLE  x I-STATE  ->  VAR-STATE"
```

```
  (let ((var-state (assoc variable i-state :test #'equal)))
    (cond (var-state
           (second var-state))
          (t
           'free))))

(defun s-functor (term)
  (cond ((final-p term)
         (first (second term)))
        (t (first term))))


(defun s-termargs (term)
  (cond ((final-p term)
         (rest (second term)))
        (t (rest term))))



;;;;;
;;;;;
;;;;;            CONSTRUCTORS and MODIFIERS
;;;;;
;;;;;


(defun create-ExtTable ()
" -> ExtTable
   creates an empty extension table"
   nil)

(defun add-procEntry (procEntry ExtTable)
"procEntry x ExtTable -> ExtTable"
  (cons procEntry ExtTable))

(defun rm-procEntry (pname ExtTable)
"PNAME x ExtTable -> ExtTable"
  (cond ((null ExtTable) nil)
        (t (let ((p-entry (first ExtTable)))
             (cond ((equal pname (first p-entry)) ; pname matched the procEntry
                    (rest ExtTable))
                   (t (cons p-entry (rm-procEntry pname (rest ExtTable)))))))))


(defun modify-procEntry (pname new-entry ExtTable)
"PNAME x newEntry x ExtTable -> ExtTable"
  (add-procEntry new-entry
                 (rm-procEntry pname ExtTable)))

(defun addnew-procEntry (pname ExtTable)
"PNAME x ExtTable -> ExtTable
   creates a new procedure entry for pname and adds the entry to the
   extension table"

   (add-procEntry (mk-procEntry pname nil nil nil) ; the procEntry skeleton
                  ExtTable))


(defun mk-succpatEntry (callp succp oc-bit)
  (list callp succp oc-bit))

(defun modify-succpEntry (callp succp oc-bit procEntry)
"function modify-succpEntry: callp x succp x oc-bit procEntry
                             -> procEntry"
  (mk-procEntry (first procEntry)
                (s-aliasesBit procEntry)
                (modify-succpEntry2 callp succp oc-bit
                                    (s-succpat-table procEntry) nil)
                (s-indexList procEntry)))

(defun modify-succpEntry2 (callp succp oc-bit succpat-table new-succpat-table)
  (cond ((null succpat-table)
         (cons (mk-succpatEntry callp succp oc-bit) new-succpat-table))
        (t
         (let ((succpat-entry (first succpat-table)))
           (cond ((equal callp (first succpat-entry))
```

```
                (cons (mk-succpatEntry callp succp oc-bit)
                      (append new-succpat-table
                              (rest succpat-table))))
               (t
                (modify-succpEntry2 callp succp oc-bit
                                    (rest succpat-table)
                                    (cons succpat-entry new-succpat-table)))
               )))
       ))

(defun add-succp-to-table (pname callp succp ExtTable)
"PNAME x CALLP x SUCCP x ExtTable
                           -> ExtTable
   computes the least upper bound from the actual success pattern and succp
   and puts the result in the correspondending slot in the returning ExtTable"

   (let ((procEntry (s-procEntry pname ExtTable)))

         ;;; the succpat entry is allready closed
     (cond ((succpat-close-p callp procEntry)
            ExtTable)

           ;;; the succpat entry is open
           ((succpat-open-p callp procEntry)
            (let ((old-succp (s-succpat callp procEntry)))
              (modify-procEntry pname
                                (modify-succpEntry callp
                                                   (pattern-lub succp old-succp)
                                                   'open procEntry)
                                ExtTable)))

           (t " error because the entry is neither close nor open"
             (error "close/open error in the succpat table"))
           )))


(defun get-succp (arglist i-state ret-value)
  (cond ((null arglist) (list ret-value))
        (t
         (cons (term-state (project-state (first arglist) i-state))
               (get-succp (rest arglist) i-state ret-value)))
        ))


(defun mk-succpat-open (pname callp ExtTable)
"PNAME x CALLP x ExtTable -> ExtTable "
  (let ((procEntry (s-procEntry pname ExtTable)))
    (modify-procEntry pname (modify-succpEntry callp
                                               (s-succpat  callp procEntry)
                                               'open
                                               procEntry)
                      ExtTable)))


(defun mk-succpat-close (pname callp why ExtTable)
"PNAME x CALLP x WHY x ExtTable
                         -> succpatEntry x ExtTable"
  (let ((procEntry (s-procEntry pname ExtTable))
        (new-procEntry nil))
    (case why
      ((goal-finished  safe-rec)  ;;; correct succes patter is always computed!
       (setf new-procEntry (modify-succpEntry callp
                                              (s-succpat callp procEntry)
                                              'close procEntry)))
      (unsafe-rec   ;;; compute a conservative (but safe) approximation
       (setf new-procEntry (modify-succpEntry callp
                                              (conser-approx callp)
                                              'close procEntry)))
      (otherwise
       (error "Unknown justification for closing the succpat entry")))

    (list (s-succpat callp new-procEntry)
          (modify-procEntry (first procEntry) new-procEntry
                            ExtTable))))
```

```
(defun mk-procEntry (pname ali-bit succpat-table index-table)
"PNAME x ALI-BIT x  SUCCPAT-TABLE x INDEX-TABLE
                        -> procEntry
   builts a procedure entry"
   (list pname ali-bit succpat-table index-table))


(defun set-aliasesBit (procEntry)
  (setf (second procEntry) t)
  procEntry)


(defun mk-pname (clause)
"CALUSE -> PNAME
  computes the procedure name, to which the clause belongs to"
  (let ((head (s-conclusion clause)))
    (list (first head)
          (length (rest head)))))


(defun mk-litname (literal)
  (list (first literal)
          (length (rest literal))))


(defun mk-lsp-succpat (callp)
  ;;; Note, there are only closed arguments allowed for a lispcall
  (cond ((null callp) '(closed))
        (t (cons (s-unify (first callp) 'closed)
                 (mk-lsp-succpat (rest callp))))
        ))

(defun mk-noDef-succpat (callp)
  ;;; Note, you have to safe in your approximation a
  (cond ((null callp) '(dontknow))
        (t (cons (s-unify (first callp) 'dontknow)
                 (mk-noDef-succpat (rest callp))))
        ))

(defun add-istate (var var-state state)
  (cond ((null state)
         (cons (list var var-state) state))
        (t (let ((state-entry (first state)))
             (cond ((equal var (first state-entry))
                    (cons (list var var-state)
                          (rest state)))
                   (t
                    (cons state-entry
                          (add-istate var var-state (rest state))))
                   )))
        ))




;;;;;
;;;;;
;;;;;            PREDICATES
;;;;;              -
;;;;;


(defun exist-procEntry-p (pname ExtTable)
  (s-procEntry pname ExtTable))

(defun exist-succpatEntry-p (callp procEntry)
  (s-succpat-entry callp procEntry))

(defun aliases-p (procEntry)
  (s-aliasesBit procEntry))

(defun return-aliases-p (i-list db)
"CL-INDEX-LIST x DATABASE -> TRUTH-VALUE"
  (cond ((null i-list) nil)
        (t
```

```
        (let ((clause (nth (first i-list) db))) ;Select the next clause
          (cond ((repeated-variable-p (s-conclusion clause)) ; Does a variable occur more than once
                 t)                                          ; in the head?
                ; does a head variable occur in the body
                ((intersection (collect-variables (s-conclusion clause))
                               (collect-variables (s-premises clause))
                               :test #'equal)
                 t)

                (t ; no, than testing the next clause
                 (return-aliases-p (rest i-list) db))
                )))
       ))


(defun call-aliases-p (i-list ali-list db)
  (cond ((null i-list) nil)
        ((call-to-aliases-p (s-premises (nth (first i-list) db))
                            ali-list)
         t);
        (t
         (call-aliases-p (rest i-list) ali-list db))
       ))


(defun call-to-aliases-p (literals ali-list)
"LITERAL x ALI-LIST -> TRUTH-VALUE"
  (cond ((null literals) nil)
        (t
         (let ((lit (first literals)))
           (cond ((final-p lit)
                  (call-to-aliases-p (rest literals) ali-list))
                 ((is-t lit)
                  (call-to-aliases-p (cons (third lit) (rest literals))
                                     ali-list))
                 ((member (list (first lit) (length (rest lit)))
                          ali-list :test #'equal)
                  t)
                 (t
                  (call-to-aliases-p (rest literals) ali-list))
                )))
       ))

(defun call-aliasing-p (literal)
  (repeated-variable-p literal))

(defun repeated-variable-p (literal)
"LITERAL -> TRUTH-VALUE"
  (rep-var-p literal nil))

(defun rep-var-p (lit vars)
  (cond ((null lit) nil)
        (t
         (let ((arg (first lit)))
           (cond ((vari-t arg)
                  (cond ((member (second arg) vars)
                         t); repeated variable detected
                        (t (rep-var-p (rest lit) (cons (second arg) vars)))
                        ))
                 (t (rep-var-p (rest lit) vars)))))
       ))


(defun succpat-close-p (callp procEntry)
  (eq 'close (s-succpat-state callp procEntry)))

(defun succpat-open-p (callp procEntry)
  (eq 'open (s-succpat-state callp procEntry)))


(defun inclusion-p (ts1 ts2)
"TERM-STATE x TERM-STATE -> TRUTH-VALUE"
  (case ts1
    (empty (null ts2))
    (true (cond ((eq ts2 'true) t)
                (t (null ts2))))
    (closed (cond ((or (eq ts2 'closed)
```

```
                    (eq ts2 'true))
                t)
               (t (null ts2))))
    (free (cond ((eq ts2 'free))
                (t (null ts2))))
    (dontknow t)))


(defun recursive-call-p (clause)
  (let ((head (s-conclusion clause)))
    (recursive-call2-p (first head)
                       (length (rest head))
                       (s-premises clause))))

(defun recursive-call2-p (cl-name arity premises)
  (cond ((null premises) nil)
        (t
         (let ((prem (first premises)))
           (cond ((is-t prem)
                  (recursive-call2-p cl-name arity
                                     (cons (s-expr-is prem)
                                           (rest premises))))
                 ((final-p prem)
                  (recursive-call2-p cl-name arity (rest premises)))
                 ((and (eq cl-name (s-functor prem))
                       (eql arity (length (s-termargs prem))))
                  t)
                 (t
                  (recursive-call2-p cl-name arity (rest premises)))
                 )))
        ))
```

B.2 Datei "mode-rfi-interface.lisp"

```
(defun rfi-modes (userline)
  (let ((mode-query (second userline))
        (query-list (rest userline))
        (query-length (length (rest userline)))
        (database *rfi-database*))
    (cond ((eql query-length 1)
           (cond ((one-goal-p mode-query)
                  (compute-modes mode-query database))
                 ((predicate-p mode-query)
                  (rfi-modes (list 'modes (mk-goal mode-query))))
                 (t
                  (progn
                    (princ "Use syntax: modes (cl-name model ... modeN)") (terpri)))))
          (t
           (cond
            ((predicates-p query-list)
             (rfi-modes (cons 'modes (mk-conjunction query-list))))
            ((conjunction-p query-list)
             (let ((varList (mk-new-var-list query-list 0 nil))
                   (md-query (mk-query query-list))
                   (md-clause nil))
               (setf md-clause (mk-md-Clause query-list varList))
               (compute-modes md-query (cons md-clause database))))
            (t
             (progn
               (princ "Use syntax: modes (cl-name model ... modeN)") (terpri))))
          ))))

(defun compute-modes (md-query database)
  (let ((ExtTable (mode-interpret md-query database)))
    (pp-extTable ExtTable)
    ExtTable))

(defun one-goal-p (query)
  (cond ((atom query)
         nil)
        ((not (numberp (second query)))
         t)
        (t nil)))

(defun predicate-p (query)
  (cond ((atom query)
         nil)
        ((numberp (second query))
         t)
        (t nil)))

(defun conjunction-p (query-list)
  (list-and (mapcar #'one-goal-p query-list)))

(defun predicates-p (query-list)
  (list-and (mapcar #'predicate-p query-list)))

(defun list-and (l)
  (cond ((null l) t)
        (t (and (first l)
                (list-and (rest l))))))


(defun mk-conjunction (predicates)
  (mapcar #'mk-goal predicates))

(defun mk-goal (predicate)
  (cons (first predicate)
        (make-list  (second predicate) :initial-element 'dontknow)))

(defun mk-new-var-list (conjunction varnum varlist)
  (cond ((null conjunction) varlist)
```

```
        (t
         (let ((goal (first conjunction)))
           (mk-new-var-list (rest conjunction)
                            (+ varnum (list-length (rest goal)))
                            (append varlist (mk-varList varnum (list-length (rest goal))))))
        )))

(defun mk-varList (start count)
  (cond ((= count 0) nil)
        (t
         (cons (list 'vari (intern (princ-to-string start)))
               (mk-varList (1+ start) (1- count))))))



(defun mk-query (conjunction)
  (cons 'md-query (apply #'append (mapcar #'cdr conjunction))))

(defun mk-md-Clause (conjunction varList)
  (let ((cl-tag 'ft)
        (cl-head (cons 'md-query varList))
        (cl-body (project-var conjunction varList)))
  (append (list cl-tag cl-head)
          cl-body)))

(defun project-var (conjunction varList)
  (cond ((null conjunction) nil)
        (t
         (let* ((goal (first conjunction))
                (arglist (project-goal-var (rest goal)
                                           varList)))
           (cons (cons (first goal)
                       argList)
                 (project-var (rest conjunction)
                              (nthcdr (length arglist) varList)))))))

(defun project-goal-var (arglist varList)
  (cond ((null arglist) nil)
        (t
         (cons (first varList)
               (project-goal-var (rest arglist) (rest varList))))))



(defun pp-extTable (extTable)
  (cond ((null extTable) nil)
        (t
         (let ((proc-name nil);  predicate-name and arity
               (proc-entry (first extTable)))
           (setf proc-name (first proc-entry))
           (cond ((eq (first proc-name) 'md-query)
                  ; the help predicate for handling multi query approximation
                  nil)
                 (t (terpri)
                  (princ "Procedure: ") (princ (first proc-name)) (princ "/") (princ (second proc-
name))
                  (princ "  Aliasing: ") (princ (if (s-aliasesBit proc-entry) "yes" "no")) (terpri)
                  (pp-succpat-Table (s-succpat-Table proc-entry)))
                 ))
         (pp-extTable (rest extTable)))
        ))

(defun pp-succpat-Table (succpat-list)
  (cond ((null succpat-list) nil)
        (t
         (princ "calling: ")
         (pp-pattern (first (first succpat-list)))
         (princ " returning: ")
         (pp-pattern (second (first succpat-list)))
         (terpri)
         (pp-succpat-Table (rest succpat-list)))))

(defun pp-pattern (pattern)
  (princ "<")
  (dolist (x pattern)
    (princ x) (princ " ")))
```

```
  (princ ">"))

(defun lispcall-p (goal)
  (lisp-builtin-p (s-functor goal)))
```

Literaturnachweis

[All87] Coral Software Corp. Cambridge. *Allegro Common Lisp for the Macintosh*, Franz, Inc. Berkeley, CA, 1987.

[Ave89] Jürgen Avenhaus. *Reduktionssysteme*. Universität Kaiserslautern Fachbereich Informatik, 1989.

[Ben90] H. Benker, M. Dorochevsky, J Noye, and A. Sexton. *A Knowledge Crunching System*, ECRC, ITG 90, 1990, .

[Bol90a] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. Tech. Rept. SR-90-05, SEKI Report, Deutsches Forschungszentrum für Künstliche Intelligenz Universität Kaiserslautern, April, 1990.

[Bol90b] Harold Boley *Partial Deduction/Evaluation in a Relational/Functional Language.* Vortragskurzfassung für ICLP *Workshop on Partial Deduction and Partial Evaluation*, Israel, June, 1990.

[Bol91] Harold Boley, Klaus Elsbernd, Hans-Günther Hein, and Thomas Krause. *RFM Manual: Compiling RELFUN into RelationallFunctional Machine*, DFKI Kaiserslautern ARC-TEC Projekt, February, 1991.

[Bru86] M. Bruynooghe. Compile Time Garbage Collection. In *Proceedings of the IFIP Working Conference on Programming Transformation and Verification*, Elsevier-North Holland, 1986.

[Bue88] W. Büttner, R. Enders, K. Estenfeld, H. Leiß, M. Rükert, R. Schmid, H.-A. Schneider, and E. Tidén. *PROLOG-XT*, Siemens AG, Zentralbereich Forschung u. Entwicklung, December, 1988.

[Cha85] J. Chang, A.M. Despain, and D. DeGroot. *AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis*, Compcon 85, IEEE Computer Society, February 1985, .

[Deb86] Saumya K. Debray. Register Allocation in a Prolog Machine. In *International Symposium on Logic Programming IEEE*, Department of Computer Scince State University of New York at Stony Brook, 1986.

[Deb86a] Saumya K. Debray *Global Optimization of Logic Programs*, Ph.D. dissertation, SUNY at Stony Brook, NY 11794, 1986.

[Deb88] Saumya K. Debray. Efficient Dataflow Analysis of Logical Programs. In *Proceedings of the Fifteenth Annual ACM SIGACTSIGPLAN Symbosium on Principles of Programming Languages,* 1988.

[Hei89] Hans-Günther Hein. Adding WAM instructions to support Valued Clauses for the Relational/Functional Integration Language RELFUN. Tech. Rept. SWP-90-02, SEKI Working Paper, Universität Kaiserslautern Fachbereich Informatik, December, 1989.

[Hei91] Hans-Günther Hein. Der Codegenerator und die NyWAM: Register und andere Optimierungen. In *Arbeitstreffen über WAM-Erweiterungen am DFKI Kaiserslautern,* Harold Boley [Hrsg.], (in German), March 1991.

[Hic89] Timothy Hickey and Shyam Mudambi. *Global Compilation of Prolog,* The Journal of Logic Programming Vol 7, 1989, .

[Kah83] Kenneth M. Kahn. Partial Evaluation as an Example of the Relationships betwen Programming Methodology and Artificial Intelligence. Tech. Rept. 17 University of Uppsala, Department of Computer Science, 1983.

[Kow71] R.A. Kowalski and D. Kuehner Linear Resolution with Selection Function. *Artificial Intelligence 2*(1971), 227-260.

[Kra90] Thomas Krause. Klassifizierte relational/funktionale Klauseln: Eine deklarative Zwischensprache zur Generierung von Register-optimierten WAM-Instruktionen. Tech. Rept. SWP-90-04, SEKI Working Paper, Fachbereich Informatik Universität Kaiserslautern, (in German), May, 1990.

[Kyo??] Taiichi Yuasa and Masami Hagiya. *Kyoto Common Lisp Report,* Research Institute for Mathematical Sciences, Kyoto University.

[Man87] Heikki Mannila and Esko Ukkonen. Flow analysis of PROLOG programs. In *In International Symposium on Logic Programming IEEE,* 1987.

[Mel85] C.S. Mellisch Some Global Optimization for a Prolog compiler. *Logic Programming* , 1 (1985), 43-66.

[Nei87] Burkkard Neidecker. *KAP-Maschine: Maschinenmodell und Instruktionssatz,* Universität Karlsruhe, Institut für Informatik, 1989, (in German).

[NGC88] *Selected Papers from the Workshop on Partial Evaluation and Mixed Computation 1987,* OHMSHA , LTD. and Springer-Varlag , Vol. 6, New Generation Computing(1988).

[Qui85] Quintus Computer System. *Quintus Prolog User's Guide and Reference Manual, Versin 4*, Quintus Computer System ICR, Palo Altor, September, 1985.

[Ric89] M. M. Richter. *Prinzipien der Künstlichen Intelligenz*, B. G. Teubner Stuttgart (1989).

[Sch88] Daniel De Schreyer and Maurice Bruynooghe. An application of Abstract Interpretation in source level Program Transformation. In *Programming Languages Implementation and Logic Programming*, 1988.

[Sin91] Michael Sintek and Werner Stein. WAM Indexing Techniques. In *Arbeitstreffen über WAM-Erweiterungen am DFKI Kaiserslautern*, Harold Boley [Hrsg.], March 1991.

[Som86] Zoltan Somogyi. A system of precise modes for logic programs. In *Proceedings of 4th International Conference of Logic Programming*, Department of Computer Science University of Melbourne, 1986.

[Tay90] Andrew Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of the Seventh International Conference of Logic Programming*, David H. D. Warren and Peter Szeredi, 1990, pp. 174-185.

[Tho88] J. Thom and J. Zobel. NU-Prolog Reference Manual, Version 1.3. Tech. Rept. 86-10Department of Computer Science, University of Melbourne, 1988.

[Wae89] Annika Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Proceedings 5th International Conference of Logic Programming*, 1989.

[War83] David H.D Warren. An Abstract Prolog Instruction Set. Tech. Rept. 309, SEKI Report, SRI International, Menlo, Park CA, October, 1983.

Register

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:

1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-90-17
Stephan Busemann
Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka :
On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt:
The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost
X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents
A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist
WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta
RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke
A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel
Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-13
Gert Smolka
Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-15
Bernhard Nebel, Gert Smolka
Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann
Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber: Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay: Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler
Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann
Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel and Rainer Bleisinger
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

DFKI Documents

D-89-01
Michael H. Malburg, Rainer Bleisinger:
HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

D-90-02
Georg Seul: Logisches Programmieren mit Feature -Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

D-91-01
Werner Stein , Michael Sintek
Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause
RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause
Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 pages

Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN.

Thomas Krause

D-91-08
Document