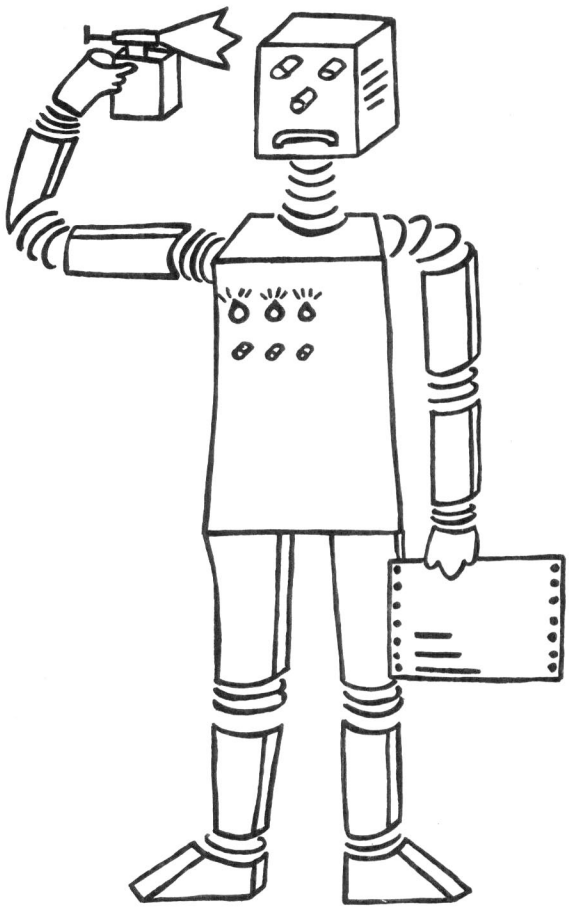
SEKI-Working Paper

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



# LISPLOG im Kontext anderer LISP/PROLOG-Vereinheitlichungen

Franz Kammermeier

Dezember 1986 SWP-86-09

# LISPLOG

## im Kontext anderer LISP/PROLOG-Vereinheitlichungen

Franz Kammermeier

Fachbereich Informatik
Universitaet Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1
W. Germany

uucp: unido!uklirb!kamei
- oder -
kamei@uklirb.UUCP

Projektarbeit
(Betreuer : Harold Boley)

Dezember 1986

## Abstract

Since 1979 a lot of functional/relational languages integrating LISP and PROLOG have been described in the literature.
In this paper it is tried to find the position of LISPLOG in the context of ten important other hybrids, which are mainly LISP based, e.g. LOGLISP, LM-Prolog and Symbolics Prolog.

After a short introduction and a general view of these hybrids, firstly the integration of programming styles is compared, namely the correspondence of LISP/PROLOG data structures and the facilities to access one formalism from the other.
Implementation techniques of PROLOG interpreters, particularly for implementation of control, representation of term instances and database indexing, are examined in the second part, with the emphasis being on LISP as the implementation language.
The comparison is supplemented by a short description of compilation techniques for PROLOG (in LISP) used by compilers of the treated languages.

Finally, some conclusions for present and future improvements of LISPLOG are outlined.


---

This research was partially supported by the
Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314,
"Kuenstliche Intelligenz - Wissensbasierte Systeme".

# Inhaltsverzeichnis

1. Zielsetzung ..... 4

2. Warum LISP/PROLOG-Vereinheitlichungen ..... 6

3. Die Sprachen ..... 7

3.1 LOGLISP ..... 7
3.2 HCPRVR ..... 7
3.3 Horne ..... 8
3.4 QLOG ..... 8
3.5 LM-Prolog ..... 9
3.6 Symbolics Prolog ..... 9
3.7 Babylon-Prolog ..... 10
3.8 Salford LISP/PROLOG ..... 10
3.9 LISLOG ..... 10
3.10 ConProlog ..... 11
3.11 LISPLOG ..... 11

4. Vergleich ..... 13

4.1. Die Integration der Programmierstile ..... 13
4.1.1 Datenstrukturen und externe Syntax ..... 13
4.1.2 Durchgriffe von der PROLOG-Ebene auf LISP ..... 16
4.1.3 Durchgriffe von der LISP-Ebene auf PROLOG ..... 24

4.2 Implementation der Interpreter ..... 29
4.2.1 Implementationstechniken fuer die Kontrolle .... 29
4.2.2 Repraesentation von Terminstanzen und Bindungsumgebungen ..... 35
4.2.3 Indexierung der Datenbasis ..... 40

4.3 Compilationstechniken ..... 44

5. LISPLOG im Kontext ..... 47

Literatur ..... 49

Anhang A: Wissenswertes ueber LISPLOG ..... 54

## 1. Zielsetzung

Seit 1979 wurde eine grosse Anzahl von funktional/relationalen Sprachen in der Literatur beschrieben, die LISP und PROLOG vereinheitlichen.
Im Rahmen dieser Arbeit wird versucht, die Stellung von LISPLOG im Kontext von zehn wichtigen anderen Hybriden, die ueberwiegend auf LISP basieren, herauszuarbeiten, beispielsweise LOGLISP, LM-Prolog und Symbolics Prolog.

Nach einer kurzen Einfuehrung und einem Ueberblick ueber die Vergleichssprachen werden als erstes die Konzepte zur Vereinheitlichung der Programmierstile verglichen - genauer: die Korrespondenz der LISP/PROLOG-Datenstrukturen und die Moeglichkeiten der beiden Formalismen aufeinander zuzugreifen.
Implementationstechniken fuer PROLOG-Interpreter, mit Betonung auf der Implementationssprache LISP, insbesondere fuer die Implementation der Kontrolle, die Repraesentation von Terminstanzen und die Indexierung der Datenbasis werden im zweiten Teil behandelt.
Der Vergleich wird durch eine kurze Beschreibung von Compilationstechniken fuer PROLOG (besonders mit Zielsprache LISP), wie sie bei den Vergleichssprachen gefunden wurden, ergaenzt.

Am Ende der einzelnen Kapitel des Vergleichs und am Schluss der ganzen Arbeit werden einige Folgerungen fuer gegenwaertige (seit Juni 1986 bearbeitete) und zukuenftige Verbesserungen von LISPLOG eroertert.

Natuerlich werden dadurch nicht alle moeglichen Vergleichsaspekte abgedeckt, es sei hier nur der grosse Bereich der Programmierumgebungen und der grundlegende Sprachumfang genannt, ganz zu schweigen von der Vielzahl der Besonderheiten und experimentellen Spracheigenschaften der einzelnen Hybride. Zudem wurden hauptsaechlich praxisorientierte Argumente beim Vergleich beruecksichtigt, die durch theoretisch fundierte Ueberlegungen ergaenzt werden sollten.
Zum einen sprechen vergleichstechnische Gruende, wie starke Abhaengigkeit vom zugrundeliegenden System, zu grosse Vielfalt der Vergleichspunkte und fehlende Vergleichspartner gegen die Aufnahme dieser Aspekte, zum anderen haette dies den Rahmen (als Projektarbeit) gesprengt.

Tabelle 1 (vgl. [Boley, Kammermeier et al. 1985]) enthaelt eine alphabetische Aufzaehlung einiger uns bekannter LISP/PROLOG-Systeme samt Literaturangaben. Sie erhebt jedoch Angesichts der staendig wachsenden Anzahl solcher hybriden Systeme keinen Anspruch auf Vollstaendigkeit.
Die im Vergleich beruecksichtigten Systeme sind durch einen Stern gekennzeichnet.

Tabelle 1: LISP/PROLOG-Systeme

| Systeme | Literatur |
|---|---|
| APPLOG | [Cohen 1986] |
| *Babylon-Prolog | [Gross 1985] |
| *ConProlog | [Knopfler & Hotop 1985] |
| Exeter Prolog | [Fogelholm 1984] |
| FOOLOG | [Nilsson 1984] |
| *HCPRVR | [Chester 1979], [Chester 1980] |
| *Horne | [Frisch et al. 1982], [Allen et al. 1983] |
| *LISLOG | [Bourgault et al. 1985], [Sansonnet 1986] |
| *LISPLOG | [Bernardi 1986], [Bernardi et al. 1987], [Boley & Kammermeier et al. 1985], [Boley & Meyer 1986], [Boley 1986b], [Dahmen, Herr, Hinkelmann, Morgenstern 1985], [Dahmen 1986a], [Herr 1986], [Hinkelmann 1986], [Lessel 1986], [Meyer 1987] |
| *LM-Prolog | [Kahn 1983], [Kahn & Carlsson 1983], [Kahn & Carlsson 1984], [Carlsson 1985] |
| *LOGLISP | [Robinson & Sibert 1981], [Robinson & Sibert 1982a], [Robinson & Sibert 1982b], [Schrag 1984] |
| LOVLISP | [Greussay 1983] |
| PiL | [Wallace 1983] |
| *QLOG | [Komorowski 1982] |
| QUTE | [Sato & Sakurai 1983], [Sato & Sakurai 1984] |
| *Salford LISP/PROLOG | [SALFORD 1984], [Bailey 1985] |
| *Symbolics Prolog | [Combuechen 1985], [SYMBOLICS 1985] |
| TAO | [Takeuchi et al. 1983], [Okuno et al. 1984] |
| TLOG | [Read & Dyer 1985] |
| UNIFORM | [Kahn 1986] |
| YAQ | [Carlsson 1981] |

## 2. Warum LISP/PROLOG-Vereinheitlichungen?

Die Frage erscheint berechtigt, da solch ein hybrides Werkzeug vom Anwender verlangt, zwei unterschiedliche Programmierstile zu beherrschen. Das gilt im Besonderen, wenn Software in Teams erstellt wird und der einzelne Benutzer sich nicht auf den ihm vertrauten Stil beschraenken kann.

Offensichtlich eignen sich jedoch LISP und PROLOG fuer bestimmte Aufgabengebiete in besonderem Masse. In PROLOG laesst sich z.B. aufgrund seiner deklarativen und nondeterministischen Natur und der Faehigkeit zum Patternmatching in knapper Form Wissen repraesentieren und durch die eingebauten Suchverfahren zurueckgewinnen und verwerten. In LISP dagegen lassen sich deterministische Algorithmen klarer und effizienter implementieren. Mit den prozeduralen Kontrollstrukturen und Anweisungen, die sich in LISP recht gut eingliedern, koennen auch Low-Level-Probleme, z.B. Betriebssystemaufgaben gut geloest werden (vgl. LISP-Maschinen).

Fuer viele LISP-Implementationen wurden inzwischen ausgefeilte und aufwendige Programmierumgebungen geschaffen (Extremfall: LISP-Maschinen). Da sich LISP- und PROLOG-Datenstrukturen zwanglos aufeinander abbilden lassen, liegt es nahe, diese Programmierumgebungen auch zur Entwicklung von PROLOG-Programmen zu verwenden.
Das erleichtert zudem den LISP-Anhaengern der KI-Gemeinde den (partiellen) Umstieg auf PROLOG (und umgekehrt).

Auch fuer die Implementation eines PROLOG-Systems ist LISP eine attraktive Basis. Bei Beibehaltung eines moeglichst puren Programmierstils bleibt der Interpreter (oder Compiler) uebersichtlich, funktionelle Einheiten (z.B. der Unifikationsalgorithmus) lassen sich aufgrund der klaren Schnittstelle (Argumentliste/Wert) leicht auswechseln.
So kann experimentiert, z.B. die Implementation der Kontrolle oder die Semantik des PROLOGs veraendert werden.

Schliesslich sind die Programme, falls nicht spezielle Eigenschaften des LISP-Systems oder der Hardware ausgenutzt werden, ausgesprochen portabel.

## 3. Die Sprachen

In den Kurzbeschreibungen der elf Hybride wird, soweit bekannt, dargelegt, was die Hauptmotivation zur Entwicklung der jeweiligen Sprache war, wo Schwerpunkte gesetzt wurden und welche Besonderheiten in Konzept und Implementation liegen.
Um einen ungefaehren Eindruck von der Geschwindigkeit der jeweiligen Systeme zu geben, wurden an einigen Stellen LIPS-Angaben gemacht, die sich im allg. auf das uebliche Programm zur Umkehrung einer dreissigelementigen Liste beziehen. Diese Angaben duerfen keinesfalls als exakter Massstab fuer die Laufzeiteffizienz der Sprachen gelten.
Schliesslich wird auf die Literatur verwiesen, die diesem Vergleich zugrunde liegt.

### 3.1 LOGLISP

Der Implementation von LOGLISP lag der Wunsch zugrunde, LISP um einen Logik-Teil zu erweitern. Dabei wurde besonderer Wert auf die Reinheit der Logik-Sprache gelegt, die nur denotativ und nicht mit der Steuerung der Kontrolle vermischt sein sollte.
Der Interpreter verwendet im Gegensatz zu ueblichen PROLOG-Implementationen kein Tiefensuchverfahren mit Backtracking, sondern ein Breiten-/Bestensuchverfahren. Die Kontrolle des Beweisverlaufs erfolgt durch spezielle LISP-Funktionen und kann durch Veraenderung von Parametern beeinflusst werden. So konnte auf imperative Spracheigenschaften wie den Cut-Operator und die Beruecksichtigung der Reihenfolge von Klauseln verzichtet werden. Fuer spezielle Anwendungen kann jedoch auf einen PROLOG-Modus zurueckgegriffen werden.
Eine weitere Besonderheit ist die weitgehende Einbettung von LISP in die Logik-Sprache, die die Reduktion von Logik-Termen in Zielen allgemein gestattet.
Die Effizienz des Systems ist, wohl auch aufgrund des Breitensuchverfahrens, nicht sehr gross (ca. 150 Knoten/s (LIPS?)). Am Rome Air Development Center wurden inzwischen Ueberlegungen fuer einen Compiler vorgenommen.
Als Literatur diente vor allem [Robinson & Sibert 1982b]; [Robinson & Sibert 1982a], [Robinson & Sibert 1981] und Teile von [Schrag 1984] wurden zur Abrundung herangezogen.

### 3.2 HCPRVR

HCPRVR (Horn Clause PRoVeR) ist wahrscheinlich die aelteste der hier betrachteten Vereinheitlichungen [Chester 1979]. Sie wurde an der Universitaet von Texas in Austin zur Implementation von Sprachverarbeitungsprogrammen entwickelt, da noch kein konventionelles PROLOG-System zur Verfuegung stand. Spaeter wurde sie wegen der Verfuegbarkeit der gewohnten und flexiblen

LISP-Programmierumgebung weiterverwendet.
Fuer den Vergleich wurde [Chester 1980] herangezogen. Die LISP/PROLOG-Schnittstellen sind hier noch recht rudimentaer. Die Implementation des Resolutionsbeweisers ist eine etwas eigenwillige Kombination von funktionalem und prozeduralem Programmierstil.
Die Angaben ueber die Geschwindigkeit gehen in [Chester 1980] und [Komorowski 1982] etwas auseinander, sie ist jedoch auf jeden Fall nicht groesser, als die von interpretiertem DECsystem-10 Prolog auf einer KI10 (nach den Angaben von [Kahn & Carlsson 1983] im Bereich von 0,5 kLIPS).

## 3.3 Horne

Der HCPRVR diente den Entwicklern von Horne als Ausgangsbasis fuer eine eigene LISP/PROLOG-Vereinheitlichung. In der in [Frisch et al. 1982] beschriebenen Version weist Horne gegenueber HCPRVR einige Verbesserungen auf. So wurden zum Beispiel die LISP/PROLOG-Schnittstelle erweitert und die Indexierung der Datenbasis verfeinert. Die Implementierung eines Compilers brachte eine weitere Geschwindigkeitssteigerung um das Zwei- bis Vierfache.
Der Logik-Sprachumfang wurde durch die Einfuehrung von Variablentypisierung erweitert, die durch besondere Klauseln in der Datenbasis erfolgt.
[Allen et al. 1983] beschreibt die Ergaenzung um einen Mechanismus, der den Aufschub des Beweises einer Anfrage mit noch ungebundenen Variablen ermoeglicht, sodass der Wertebereich dieser Variablen im weiteren Verlauf des Beweises einer Beschraenkung unterliegt.
Das System ist relativ portabel, es existierten 1982 Implementationen in FRANZ LISP und, als abgemagerte Version, in UCI LISP.

## 3.4 QLOG

[Komorowski 1982] beschreibt die Implementation von QLOG im Interlisp-System.
Der Schwerpunkt lag auf einer weitgehenden Ausnutzung der Interlisp-Programmierumgebung fuer den PROLOG-Teil von QLOG. Das erstreckt sich bis hin zu Break Package, History Package, Spelling Corrector und File Librarian. Dafuer musste der PROLOG-Interpreter so gestaltet werden, dass PROLOG-Prozeduren als LISP-Funktionen dargestellt und aufgerufen werden koennen.
Die Portabilitaet des Systems ist gut, es lief 1982 auch unter FRANZ LISP, CDC LISP und LISP Machine LISP.
Die Geschwindigkeit ist vergleichbar mit interpretiertem DECsystem-10 Prolog.

## 3.5 LM-Prolog

LM-Prolog ist ein ausgesprochen farbiges System.
Urspruenglich wurde es zur Entwicklung eines partiellen Evaluators fuer PROLOG implementiert.
Es verfuegt ueber eine ganze Reihe innovativer Sprachkonzepte, wie effiziente Conditionale, Primitive zur Berechnung von 'lazy' Mengen und Multimengen, Kontrollprimitive zur Verzoegerung von Beweisen (a la 'lazy evaluation') und zur Einschraenkung des Wertebereichs von Variablen (vgl. Horne), u.v.a.m.
Der Interpreter wurde moeglichst modular und seiteneffektfrei gestaltet und arbeitet ueberwiegend funktional unter Verwendung von Stroemen.
Spezielle Konzepte von LISP Machine LISP, wie unsichtbare Zeiger, cdr-codierte Listen und 'stack groups' wurden hier, wie fuer den vom Compiler erzeugten LISP-Code, ausgiebig verwendet, sodass die Implementation trotzdem eine gute Effizienz aufweist.
Unter Verwendung von Microcode ist LM-Prolog auf CADR LISP-Maschinen interpretiert dreimal so schnell, wie interpretiertes DECsystem-10 Prolog auf einer KI10 und compiliert etwas langsamer als compiliertes DECsystem-10 Prolog (ca. 6 kLIPS).
Als Quelle diente vor allem [Kahn 1983] und [Kahn & Carlsson 1984], bei der Betrachtung der Integration der Programmierstile auch [Kahn & Carlsson 1983]. Als Ergaenzung wurden [Kahn 1982] und [Carlsson 1985] verwendet.

## 3.6 Symbolics Prolog

Ebenfalls auf einer LISP-Maschine laeuft Symbolics Prolog [SYMBOLICS 1985] (Version 12). Es ist ein hochgradig auf Geschwindigkeit optimiertes Produkt, dass eine Inferenzrate von ca. 50 kLIPS aufweist. Es arbeitet nur als Compiler und verwendet ausgiebig Microcode.
Das System wurde gut in die Programmierumgebung der LISP-Maschine integriert, allerdings laesst die Unterstuetzung fuer Laufzeittests im Vergleich dazu etwas zu wuenschen uebrig (einfachster Tracer, keine Break-Unterstuetzung, kein Stepper, kein korrekter Backtrace).
Eine Besonderheit stellt der zweite Syntaxmodus dar, der es erlaubt, Programme in der traditionellen Edinburgh PROLOG-Syntax darzustellen. Einem geuebten PROLOG-Programmierer bleibt so die Umstellung auf die in vielen Vereinheitlichungen verwendete LISP-Syntax erspart.
Leider steht fuer Symbolics Prolog als rein kommerziellem Produkt kein Quelltext zur Verfuegung. Die Flexibilitaet der Sprache, wie sie z.B. LM-Prolog aufweist, geht so weitgehend verloren. Ueber Implementationsdetails liegen ebenfalls kaum Informationen vor.

## 3.7 Babylon-Prolog

Babylon-Prolog [Gross 1985] ist neben Produktionsregeln und Frames nur ein Wissensrepraesentationsformalismus von Babylon, einem groesseren hybriden Werkzeug zur Erstellung von Expertensystemen.
Daher wurde auf innovative Sprachkonzepte, eine Vielzahl von Primitiven oder eine hohe Effizienz (ca. 0,1 kLIPS) kein grosser Wert gelegt.
Babylon laeuft auf Symbolics und TI LISP-Maschinen; die Implementation von PROLOG basiert, wie die des ganzen Systems, hauptsaechlich auf der Verwendung von Flavors, ist also eher objektorientiert als funktional.

## 3.8 Salford LISP/PROLOG

Fuer die Implementatoren von Salford LISP/PROLOG [Bailey 1985], [SALFORD 1984] lag der Vorteil einer LISP/PROLOG-Vereinheitlichung in der Ergaenzung von PROLOG um eine Sprache, in der prozedurale Algorithmen gut implementiert werden koennen und deren Strukturen sich leicht auf PROLOG-Strukturen abbilden lassen.
Die Implementation weicht etwas von den anderen Systemen ab, denn hier wurde PROLOG nicht in LISP eingebettet, sondern beide Sprachen wurden miteinander in FORTRAN-77 geschrieben.
Die Syntax des PROLOG-Teils wurde nicht an LISP angepasst, sondern blieb in der konventionellen Edinburgh PROLOG-Form.
Ausser dem PROLOG-Interpreter steht ein Compiler zur Verfuegung, der PROLOG-Prozeduren dynamisch, abhaengig vom Typ aktueller Ziele, uebersetzt.
Die Geschwindigkeit von compiliertem PROLOG liegt im Bereich von ca. 2 kLIPS. Durch die Verwendung von nichtstandardisierten FORTRAN-77-Erweiterungen und Inline-Assemblercode ist die Portabilitaet des Systems vermutlich sehr eingeschraenkt.

## 3.9 LISLOG

LISLOG wurde in Frankreich vom Centre National d'Etudes des Telecommunications entwickelt. Im Rahmen des Projekts MAIA soll es bei der Implementation einer LISP/PROLOG-Maschine [Sansonnet 1986] verwendet werden.
Der Interpreter laeuft momentan [Bourgault et al. 1985] unter MACLISP, FRANZ LISP, LELISP und NIL, ist also sehr portabel.
Eine an PROLOG angepasste und recht umfangreiche Programmierumgebung gehoert mit zum System und steht unabhaengig von der Implementationssprache zur Verfuegung.
Die Implementation weist einige bemerkenswerte Eigenschaften auf, z.B. ist die LISP/PROLOG-Schnittstelle umfangreicher als ueblich, sie erlaubt speziell gekennzeichnete evaluierbare Terme ueberall

in Kopf und Rumpf von Klauseln.
Ein weiteres interessantes Konzept sind die sogenannten 'terms under constraints', Terme, deren Typ durch beigefuegte LISP-Praedikate festgelegt wird.

## 3.10 ConProlog

Die hier beschriebene Version ConProlog2 [Knoepfler & Hotop 1985] laeuft auf XEROX LISP-Maschinen unter Interlisp/D und entstand aufgrund der mangelnden Verfuegbarkeit von PROLOG auf diesen Maschinen.
Der Interpreter, frueher rekursiv (funktional) programmiert, wurde zur Effizienzsteigerung iterativ (prozedural) gestaltet.
Die Laufzeitdebugginghilfen sind recht mager, es existiert nur ein unvollstaendiger Box-Modell-Tracer.
Eine Version ConProlog3 soll hierbei Abhilfe schaffen und stand Ende 1985 in der Entwicklung.

## 3.11 LISPLOG

LISPLOG wurde seit April 1985 mit Unterstuetzung des SFB 314 der Deutschen Forschungsgesellschaft an der Universitaet Kaiserslautern entwickelt.
Das Hauptziel liegt in der Implementation einer moeglichst portablen, flexiblen und puren LISP/PROLOG-Vereinheitlichung, die schrittweise, an die praktische Erfahrung angelehnt, zu einem praktikablen Werkzeug weiterentwickelt werden soll.
Der Vergleich bezieht sich auf die Version LISPLOG.2 von Juni 1986, die in [Dahmen 1986a] und [Bernardi 1986] beschrieben wird. Hier wurde vor allem die Effizienz des Interpreters verbessert, was zum Teil auf Kosten der vormals weitgehenden Funktionalitaet der Implementation ging.
Bei der Betrachtung der Implementationstechniken wurde zur Vervollstaendigung der Konzepte auch LISPLOG.1 [Boley, Kammermeier et al. 1985] und in den Bewertungen und Verbesserungsvorschlaegen auch die neueste Version von LISPLOG.2 von Dezember 1986 beruecksichtigt ([Bernardi et al. 1987], [Dahmen 1986b]).
Ein erster Ansatz eines Compilers fuer LISPLOG.1 wird in [Herr 1985] und [Herr 1986] vorgestellt.
Besonderheiten von LISPLOG liegen in der Beschraenkung auf den initialen Cut, die einen transparenteren Programmierstil foerdert, in der Bereitstellung einer portablen Programmierumgebung mit einem erweiterten Box-Modell-Tracer fuer Funktionen und Praedikate, einem umfangreichen Break-Paket und Laufzeitunterstuetzung des initialen Cuts, sowie in der Bereitstellung eines experimentellen Verfahrens zur automatischen Quotierung von LISP-Ausdruecken im PROLOG-Teil (Trennung: Daten/Programm).

LISPLOG laeuft momentan in FRANZ LISP unter UNIX mit ca. 0,1 kLIPS auf einer VAX11/750 und, etwas abgemagert, auf Symbolics 3600 LISP-Maschinen in COMMON LISP mit ca. 0,3 kLIPS, was etwas langsamer als interpretiertes LM-Prolog ohne Microcode-Unterstuetzung auf LMI LISP-Maschinen ist.

# 4. Vergleich

## 4.1 Die Integration der Programmierstile

### 4.1.1 Datenstrukturen und externe Syntax

Bis auf eine Vereinheitlichung (Salford LISP/PROLOG) bauen alle Systeme auf einer bereits existierenden LISP-Implementation auf, in die der PROLOG-Teil eingebettet wurde. Bis auf Salford LISP/PROLOG verwenden auch alle Vereinheitlichungen fuer den PROLOG-Teil eine an LISP angepasste Syntax. So wird statt der herkoemmlichen Praefix-Notation die fuer LISP typische Cambridge-Polnische Praefix-Notation verwendet. Infix-Operatoren werden ebenfalls nicht mehr verwendet.
In Symbolics Prolog kann alternativ dazu allerdings noch in einem zu DECsystem-10 Prolog (Edinburgh Prolog) kompatiblen Modus verwendet werden, der die herkoemmliche Syntax benutzt.

Auch die interne Repraesentation von PROLOG-Datenstrukturen wurde auf LISP-Datenstrukturen abgebildet.
So koennen vorhandene Elemente der LISP-Programmierumgebung z.B. Editoren und Pretty-Printer meist ohne Probleme auch fuer PROLOG-Programme verwendet werden.

Bei der Abbildung der Datenstrukturen zeigten sich keine nennenswerten konzeptuellen Unterschiede; daher wird im folgenden nur ein kurzer Ueberblick ueber die grundlegenden Korrespondenzen gegeben.

PROLOG-Terme werden durch LISP-S-Ausdruecke dargestellt.
PROLOG-Literale werden durch LISP-Atome repraesentiert; in einigen Systemen sind jedoch nicht alle symbolischen LISP-Atome auch gueltige PROLOG-Literale (vor allem bei Verwendung von Sonderzeichen).
PROLOG-Strukturen entsprechen komplexen S-Ausdruecken; zum Beispiel wird der folgende, in Edinburgh Prolog Syntax geschriebene Term

```
book(author(lewis),title(perelandra))
```

durch die LISP-Liste

```
(book (author lewis) (title perelandra))
```

dargestellt. Die Repraesentation durch allgemeine S-Ausdruecke erlaubt es, Funktionen variabler Aritaet zu verarbeiten; die Struktur

```
(longstructure 2 3 4 . ?x)
```

wobei ?x eine Variable ist, kann z.B. mit Strukturen unbegrenzter Laenge >= 4 unifiziert werden. Auch der Funktor muss im allg. kein Symbol mehr sein, sondern kann die Form eines beliebigen S-Ausdrucks haben.
Spezielle PROLOG-Listen sind daher nicht mehr notwendig, eine solche Datenstruktur findet sich in diesen Implementationen nicht mehr.
Praedikationen werden wie Terme als S-Ausdruecke repraesentiert; allerdings, soweit Information darueber vorhanden war, mit der Einschraenkung, dass der 'car' des Ausdrucks (das Praedikatssymbol) ein symbolisches Atom oder eine PROLOG-Variable ist. So sind auch Praedikationen variabler Aritaet zulaessig.
Eine andere Sichtweise dieses Sachverhalts ist folgende: als Funktor bzw. Praedikatssymbol ist nur noch der Punkt (".") erlaubt, der LISPs 'cons' entspricht.
PROLOG-Klauseln werden durch LISP-Listen dargestellt.

Die Repraesentation von PROLOG-Variablen ist uneinheitlich. Zudem muss noch zwischen der textuellen Repraesentation auf Bildschirm und Dateien, der Repraesentation in der Datenbasis und der internen Repraesentation waehrend des Verlaufs eines Beweises unterschieden werden. Etwas mehr Information hierzu findet sich in Kapitel 4.2.2 ueber die Repraesentation von Terminstanzen und Bindungsumgebungen.

In vielen LISP-Systemen gibt es inzwischen ueber normale S-Ausdruecke hinaus spezielle Datenstrukturen, wie ganze Zahlen beliebiger Groesse, komplexe Zahlen, Strings, Streams, Flavors usw. Ueber die Moeglichkeit, solche Objekte im PROLOG-Teil zu verwenden, ist in den meisten Quellen nichts ausgesagt. Einer solchen Verwendung sollte jedoch nichts im Wege stehen. Ausdruecklich bestaetigt wird diese Moeglichkeit fuer LM-Prolog, LISLOG, und Symbolics Prolog; in LISPLOG besteht sie ebenfalls.

Fuer den Interpreter von LM-Prolog wurde fuer experimentelle Zwecke eine Reihe weiterer Datentypen definiert, die beim Durchgriff von PROLOG auf LISP (und umgekehrt) konvertiert werden muessen. In der compilativen Version wurde auf solche Datentypen verzichtet.

In Salford LISP/PROLOG ist im Gegensatz zu den anderen Vereinheitlichungen der PROLOG-Teil nicht in LISP implementiert worden. Vielmehr bauen sowohl LISP als auch PROLOG auf einer FORTRAN-77-Variante auf. Der PROLOG-Teil verwendet ausserdem die traditionelle Edinburgh Prolog-Syntax.
Trotzdem besteht die Moeglichkeit, LISP-Datenstrukturen in PROLOG zu verwenden und umgekehrt. Dazu wurde eine Abbildung aequivalenter Datenstrukturen festgelegt:
PROLOG-Grundterme entsprechen, wie oben beschrieben, LISP-Listen, jedoch sind keine Terme variabler Aritaet erlaubt. Dafuer existieren weiterhin spezielle PROLOG-Listen. In den aequivalenten LISP-Listen steht vor jedem Element der PROLOG-Liste der String "|".

Der PROLOG-Liste

```
[dies ist eine liste]
```

entspricht

```
(":" dies ":" ist ":" eine ":" liste)
```

Ueber die Konversion von Head/Tail-Konstrukten liegt keine Information vor.
Terme mit Infix-Operatoren werden bei der Eingabe in Praefix-Notation umgewandelt, sodass auch sie eine LISP-Entsprechung besitzen.

## LISPLOG im Kontext:

Die Abbildung von PROLOG-Termen auf LISP-Listen und -S-Ausdruecke mit Verwendung von Cambridge-Polnischer Praefix-Notation und der Verzicht auf spezielle PROLOG-Listen wird in fast allen Vereinheitlichungen vorgenommen.
Dadurch sind einige Verallgemeinerungen gegenueber traditionellem PROLOG, wie Terme und Praedikationen variabler Laenge, moeglich. Vor allem wird der Durchgriff von LISP nach PROLOG und umgekehrt sehr erleichtert.
Sie bringt jedoch auch einige Nachteile mit sich, so zum Beispiel bei der Kompatibilitaet zur traditionellen Syntax (nach der Uebersetzung werden Terme mit Listen unifiziert), bei der Indexierung von komplexen Termen (Funktor nicht notwendigerweise konstant) und im Speicherungsverhalten (Records lassen sich kompakter repraesentieren als Listen).
Eine ausfuehrlichere Diskussion hierzu findet sich z.B. in [Campbell & Hardy 1984].

Solange LISP der Partner von PROLOG bleibt, sollte die veraenderte Syntax jedoch beibehalten werden. Wuenschenswert ist ein zusaetzlicher Syntaxmodus fuer den konventionellen PROLOG-Benutzer. Fuer LISPLOG (wie auch fuer LM-Prolog) existiert bereits ein Translator von Edinburgh PROLOG-Syntax in die an LISP angepasste Form [Dahmen 1985]. Darueber hinaus gibt es fuer LISPLOG auch einen Translator fuer die umgekehrte Richtung, der unter bestimmten Vorraussetzungen sogar den LISP-Teil eines LISPLOG-Programms mit nach Edinburgh PROLOG uebersetzen kann ([Hinkelmann 1986], [Hinkelmann & Morgenstern 1985] und [Boley & Kammermeier et al. 1985]). Diese Uebersetzer koennten in die Benutzeroberflaeche von LISPLOG eingebaut werden.

### 4.1.2 Durchgriffe von der PROLOG-Ebene auf LISP

In allen behandelten Sprachen besteht eine Moeglichkeit, zur Bearbeitung von Teilproblemen die Kontrolle an den LISP-Interpreter zu uebergeben. So koennen deterministische Algorithmen und 'low level'-Programme als LISP-Funktionen bzw. -Prozeduren implementiert werden.
Der Vergleich gliedert sich in diesem Kapitel nach dem Zweck, fuer den der LISP-Interpreter von PROLOG aus verwendet werden soll. Innerhalb jedes Abschnitts werden verschiedene Formen des Durchgriffs nebeneinandergestellt, die von den verschiedenen Vereinheitlichungen zur Verfuegung gestellt werden.
Der letzte Vergleichspunkt behandelt Hilfsmittel, die dazu dienen sollen, die kritischen Punkte beim Durchgriff fuer den Anwender zu entschaerfen.

a) Berechnung des Werts von Ausdruecken mit LISP

Schon in konventionellen PROLOG-Systemen kann auf der rechten Seite des is-Primitivs ein arithmetischer Ausdruck stehen, der beim Beweis berechnet wird, und dessen Resultat mit dem Term auf der linken Seite unifiziert wird.

1. Allgemeines is-Primitiv:
In den meisten Vereinheitlichungen wurde dieses Primitiv so erweitert, dass beliebige evaluierbare LISP-Ausdruecke auf der rechten Seite stehen duerfen und vom LISP-Interpreter berechnet werden. In einigen Implementationen erhielt es einen anderen Namen, in Horne heisst es setv*, in LM-Prolog LISP-VALUE und in LISLOG LISP-EVAL.

Symbolics Prolog bietet darueber hinaus die Moeglichkeit mehrwertige Funktionen zu verwenden. Das are-Primitiv arbeitet prinzipiell wie das is-Primitiv, nur dass statt einer Variablen auf der linken Seite mehrere Variablen in Form einer Liste stehen, die mit den Werten der Funktion unifiziert werden.

2. LISP-Simplifikation:
Durch die Einschraenkung der Evaluation von Termen auf die rechte Seite des is-Primitivs ist der Anwender sehr eingeengt. In LOGLISP wird ein bedeutend allgemeinerer Ansatz verfolgt. Hier werden Terme in Zielen vor der Unifikation grundsaetzlich 'simplifiziert'. Simplifikation zeichnet sich gegenueber der herkoemmlichen Evaluation vor allem in zwei Weisen aus:
- Auch Terme, in denen freie Variablen vorkommen, koennen sinnvoll vereinfacht werden. Dabei werden die Subterme des Terms soweit wie moeglich vereinfacht. So kann der Term (* (+ 2 a) (+ 3 3)) durch den Term (* (+ 2 a) 6) ersetzt werden.
- Das Ergebnis einer Simplifikation kann nicht weiter reduziert werden. Entweder ist es keine Funktionsapplikation (ein LISP-Atom, ein komplexer Term ohne symbolisches Atom an erster Stelle oder ein quotierter Ausdruck), oder es enthaelt noch freie

Variablen und keine reduzierbaren Subterme mehr. In allen anderen Faellen ist es am Schluss der Simplifikation nachtraeglich quotiert worden und kann daher nicht mehr reduziert werden.

Dieser Ansatz ist sehr weitgehend und organisch. Einige Eigenschaften erscheinen mir jedoch stoerend:
- Zur Simplifikation eines Ausdrucks kann nicht mehr das einfache LISP-EVAL verwendet werden, da Subterme aufgrund ungebundender Variablen unter Umstaenden nicht reduziert werden koennen. So ist ein neuer (zeitaufwendiger?) Simplifikationsalgorithmus notwendig und der Anwender muss sich mit der Semantik dieses neuen Verfahrens vertraut machen.
- Da auch nur teilweise reduzierte Terme mit entsprechenden Partnern in Klauselkoepfen unifiziert werden duerfen, kann das System aufgrund verschiedenen Instanziierungsgrads ueberraschend zu unterschiedlichen Ergebnissen kommen. So kann das Ziel (Teilbar_durch_2 (* a x)) mit dem Kopf (Teilbar_durch_2 (* 2 x)) unifiziert werden, nicht jedoch die Simplifikation von (Teilbar_durch_2 (* 2 4)), (Teilbar_durch_2 8).
- Die Vielzahl gueltiger LISP-Funktionsnamen muss beruecksichtigt werden, wenn ein Funktor wie in Standard-PROLOG nur als Konstruktor verwendet werden soll; reine Datenlisten muessen explizit quotiert werden.

3. Evaluierbare Terme:
Eine Zwischenform stellt der Ansatz von LISLOG dar. Hier koennen Terme durch Voranstellen des Symbols '&' als evaluierbar gekennzeichnet werden. Dadurch ist es moeglich, ausserhalb der evaluierbaren Terme Funktoren weiterhin als Konstruktoren zu verwenden und laengliche Konjunktionen von is-Zielen zu vermeiden.

Beispiel: Berechnung der Fibonacci-Zahlen
([Bourgault et al. 1985], S.16)

Ohne evaluierbare Terme:

```
((FIB *n *m) (LISPEVAL (- *n 1) *n1)
             (FIB *n1 *m1)
             (LISPEVAL (- *n 2) *n2)
             (FIB *n2 *m2)
             (LISPEVAL (+ *m1 *m2) *m))
```

Mit evaluierbaren Termen im Klauselrumpf:

```
((FIB *n *m) (FIB (& - *n 1) *m1)
             (FIB (& - *n 2) *m2)
             (LISPEVAL (+ *m1 *m2) *m))
```

Evaluierbare Terme werden waehrend der Unifikation ausgewertet, falls sie einen evaluierbaren LISP-Ausdruck enthalten (d.h. u.a. ohne freie logische Variablen ausserhalb von quotierten

Ausdruecken). Sonst werden sie als komplexer Term behandelt und unter Umstaenden bei einer spaeteren Unifikation ausgewertet. Problematisch ist ihre Verwendung in Klauselkoepfen. Zum Beispiel kann mit der Klausel

```
((FIB *m (& + *m1 *m2)) (FIB (& - *n 1) *m1)
                        (FIB (& - *n 2) *m2))
```

nicht mehr das Ziel

```
(FIB 2 2)
```

bewiesen werden, da das numerische Atom '2' nicht mit dem komplexen Term (& + *m1 *m2) unifiziert werden kann.

b) Verwendung des LISP-Interpreters zum Beweis von Zielen

Im allgemeinen werden Ziele mit Hilfe einer Regel/Faktenbasis durch Resolution bewiesen. Wenn der Praedikatsname eines Ziels ein LISP-Funktionsname ist, kann das Ziel in einigen Implementationen mit Hilfe des LISP-Interpreters evaluiert und das Resultat als Failure/Success gewertet werden.

Ueblicherweise werden Non-nil-Werte nach LISP-Manier als Success und nil als Failure gewertet. Eine Ausnahme bildet LOGLISP: Hier erstreckt sich die Simplifikation auch auf die Auswertung von Praedikationen; das Resultat 'T' wird als Success gewertet, andernfalls wird versucht, das Resultat mit dem Resolutionsbeweiser weiterzuverarbeiten.

Der Konflikt, ob der Resolutionsbeweiser oder der LISP-Interpreter zum Beweis verwendet wird, wird verschieden geloest: LOGLISP simplifiziert grundsaetzlich vor der Unifikation und versucht das Resultat der Simplifikation zu beweisen.
ConProlog und LISPLOG verwenden den LISP-Interpreter immer, wenn ein Praedikat nicht in Klauselform definiert ist.
In HCPRVR, Horne und LM-Prolog muss das Praedikat entsprechend deklariert sein. In LM-Prolog wird dazu eine Option in der Datenbasis verwendet, samt einem Muster des Funktionsaufrufes, das (vermutlich) zum Test verwendet werden kann, ob ein Ziel die geeignete Form hat. Bei HCPRVR muss ein Eintrag in die Property-Liste des Praedikatsnamens erfolgen, wenn der LISP-Interpreter fuer den Beweis verwendet werden soll. Hier gilt ausserdem die Einschraenkung, dass die LISP-Funktion als FEXPR definiert sein muss. So werden eingeschachtelte Terme auch beim Beweis von Zielen mit dem LISP-Interpreter in der Regel nicht reduziert und das Problem der Quotierung von Termen entfaellt ganz (vgl. d)).

In einigen Implementationen gibt es ein Primitiv, dass den LISP-Interpreter zur Reduktion von Termen verwendet, das Resultat jedoch nur als Wahrheitswert interpretiert. Wenn nil zurueckgeben

wird, 'failed' das Primitiv, bei Non-nil-Werten ist es erfolgreich. Dies erscheint nicht ganz so natuerlich wie die direkte Interpretation eines Ziels als berechenbarerer LISP-Ausdruck. (Vergleiche Argumentation in [Boley, Kammermeier et al. 1985], S.21)

c) Erzielung von Seiteneffekten durch LISP-Programme

Obwohl die Verwendung von Seiteneffekten sowohl im funktionalen, als auch im logischen Programmierstil sehr unsauber ist, erscheint sie jedoch im Hinblick auf 'low level'-Aufgaben unvermeidbar. Im allgemeinen duerften LISP-Programme fuer solche Aufgaben verwendet werden. LM-Prolog besitzt ein Primitiv, LISP-COMMAND, das klar anzeigt, wo solche Seiteneffekte ausgeloest werden. ([Kahn & Carlsson 1983]) Es wertet den LISP-Ausdruck, den es als Parameter hat, aus, ist aber unabhaengig vom Ergebnis immer erfolgreich.
Symbolics Prolog's funcall-Primitiv erfuellt die gleiche Funktion. Erwaehnenswert sind auch die hier verfuegbaren Primitive send und sendp, die Nachrichten an Objekte (Flavors und Flavorinstanzen) versenden koennen.
In LISP/PROLOG-Vereinheitlichung mit LISP-Praedikaten als PROLOG-Zielen koennen die hier aufgefuehrten Primitive leicht mit Hilfe entsprechender LISP-Funktionen realisiert werden.

d) Behandlung von logischen Variablen und automatische Quotierung

Logische Variablen werden in allen Implementationen vor Aufruf des LISP-Interpreters instanziiert. Der Fall, dass in der instanziierten Anfrage noch freie Variablen vorhanden sind, wird unterschiedlich behandelt.

In LOGLISP werden Variablen in quotierten Ausdruecken nicht instanziiert, Terme, in denen Variablen ausserhalb solcher Ausdruecke auftauchen, werden nur teilweise reduziert.
LISLOG verzoegert die Auswertung von evaluierbaren Termen, falls noch freie Variablen ausserhalb von quotierten Ausdruecken existieren. Eine Besonderheit stellt die Behandlung von Lambda-Applikationen dar. Hier wird angenommen, dass im Rumpf einer Applikation keine logischen Variablen existieren, sodass die kostspielige Instanziierung und Suche nach freien Variablen entfaellt.
LISPLOG testet auf freie Variablen ausserhalb von quotierten Kontexten, d.h. explizit quotierten Ausdruecken, Nlambda-Applikationen und Funktoren von Funktionsapplikationen (also auch nicht in Ruempfen von Lambda-Applikationen).
Die anderen Implementationen erlauben entweder grundsaetzlich die Uebergabe von freien Variablen, oder es wurde keine Angabe dazu gemacht.
Die Verwendung dieser freien Variablen ist im allg. problematisch, da sie meist in einer internen Repraesentation

vorliegen und in LISP-Programmen entweder gar nicht oder nur mit genauer Kenntnis der Implementation sinnvoll behandelt werden koennen.

Ein Problem bei allen Durchgriffen auf den LISP-Interpreter stellt das 'call by value'-Prinzip von LISP dar. Hierdurch sind zwar geschachtelte Funktionsapplikationen moeglich (bei HCPRVR im allg. nicht mehr!), aber dafuer muessen hier Daten von Funktionsapplikationen unterscheidbar gemacht werden. Das ist besonders umstaendlich, wenn die Argumente einer Applikation erst im Verlauf des Beweises konstruiert und an eine logische Variable gebunden werden. Eine explizite Quotierung faellt hier sehr schwer.
In LOGLISP werden zumindest symbolische Atome als Konstanten behandelt, sodass der impure Zugriff auf LISP-Variablen mit einem eval-Aufruf gekennzeichnet werden muss.
In LISPLOG wurde der Versuch gemacht, geschachtelte Funktionsapplikationen nach der Instanziierung automatisch zu quotieren ([Boley, Kammermeier et al. 1985], S.37). Die Definitionsart der Funktionen (lambda/nlambda) wurde beruecksichtigt und Macro-Applikationen, soweit moeglich, expandiert. Special Forms werden nicht besonders behandelt.
Im Gegensatz zu einer Aussage in [Herr 1986] stellt die Reihenfolge von Instanziierung und anschliessender automatischer Quotierung bei der Auswertung von LISP-Ausdruecken keinen Fehler dar. Es sollte dem Anwender bewusst die Moeglichkeit offengelassen werden, Teile eines auszuwertenden LISP-Ausdrucks, die nicht grundsaetzlich als Daten zu behandeln sind, erst im Verlauf der Abarbeitung eines LISPLOG-Programms zu konstruieren. Wie oben schon erwaehnt, ist gerade hier die Konstruktion von expliziten Quotierungen ausgesprochen umstaendlich. Der Aufwand fuer die explizite Quotierung der zu instanziierenden Variablen, die zur ausdruecklichen Deklaration als Datum notwendig ist, erscheint im Vergleich dazu gering.

## LISPLOG im Kontext:

LISPLOG weist mit dem verallgemeinerten is-Primitiv und der Verwendung von LISP-Praedikaten die grundlegenden Schnittstellen von PROLOG nach LISP auf. Durch die Bereitstellung einer Quotierungsautomatik und die explizite Behandlung freier Variablen ist es besser ausgestattet, als die meisten der anderen Vereinheitlichungen.
Die Quotierungsautomatik ist allerdings besonders bei Anwendern nicht unumstritten, sie kostet Zeit und wird in ihrer Wirkung nicht immer richtig eingeschaetzt. Sie wurde daher in der neuesten Version von LISPLOG.2 als Option zur Verfuegung gestellt. Eine besondere Behandlung von speziellen Funktionen, wie COND und AND, wuerde die Quotierungsautomatik zu einem angemesseneren Verhalten fuehren.

Tabelle 2: (PROLOG -> LISP) - Durchgriff

| | RESULTAT | | BEWEIS-MIT-LISP | | | SEIT | SERVICE | |
|---|---|---|---|---|---|---|---|---|
| | IS | SIMP EVT | ZIEL | DECL | PRIM | EFKT | VARS | QUOT |
| LOGLISP | | x | x | | | | x | (x) |
| HCPRVR | | | (x) | x | | | | |
| Horne | x | | x | x | | | | |
| QLOG | | | | | ? | | | |
| LM-Prolog | x | | x | x | x | x | | |
| Symbolics Prolog | x | | | | x | x | | |
| Babylon-Prolog | x | | | | x | | | |
| Salford LISP/PROLOG | x | | | | x | | | |
| LISLOG | x | x | | | | | x | |
| ConProlog | x | | x | | | | | |
| LISPLOG | x | | x | | | | x | x |

RESULT:
IS = is- oder semantisch gleichwertiges Primitiv
SIMP = allg. LISP-Simplifikation vor Unifikation
EVT = speziell gekennzeichnete evaluierbare Terme

BEWEIS-MIT-LISP:
ZIEL = Verwendung des LISP-Interpreters zum Beweis von Zielen
DECL = Deklaration von LISP-Funktionen, die zum Beweis von Zielen verwendet werden duerfen
PRIM = Primitiv, das das Ergebnis der Auswertung eines

LISP-Ausdrucks als Success/Failure interpretiert

SEIT EFKT = Primitiv, das die Verwendung von LISP-Funktionen mit Seiteneffekten kennzeichnet

SERVICE:
VARS = Test auf freie Variablen vor dem Durchgriff
QUOT = Quotierungsautomatik

---

Ein allgemeinerer Ansatz bei der Reduktion (eingeschachtelter) Terme wie bei LOGLISP und LISLOG erscheint wuenschenswert.
Da ich annehme, dass Funktoren ueberwiegend als Konstruktoren verwendet werden, erscheint mir die explizite Kennzeichnung evaluierbarer Terme, wie in LISLOG, in der Praxis angemessener als der (harmonischere) Ansatz von LOGLISP, Konstanten explizit zu quotieren (fuer eine solchen weitgehenden Ansatz erscheint die Quotierungsautomatik zu unsicher). Die spezielle Deklaration von erlaubten LISP-Funktionsnamen, vergleichbar mit der Deklaration von LISP-Praedikaten, die beim Beweis verwendet werden duerfen (s. oben), koennte eventuell ein Ausweg sein.

Solange LISP als Partner von PROLOG verwendet wird, halte ich auch die Verwendung eines speziellen Simplifikationsalgorithmus fuer weniger geeignet als das simple LISP-eval.

Evaluierbare Terme sollten nicht in Klauselkoepfen vorkommen (vgl. Argumentation in 4.1.2a)3. ) und nicht mit 'normalen' Termen ausser Variablen unifizierbar sein.

Durch mehrfache Verwendung von Variablen im Klauselkopf ist es dennoch moeglich, dass zwei evaluierbare Terme miteinander unifiziert werden sollen. Hier halte ich es fuer angemessen, dass die Unifikation nur erfolgreich ist, wenn beide Terme textuell gleich sind.
Die Tatsache, dass der Beweis aufgrund unvollstaendiger Instanziierung fehlschlagen kann, scheint weniger verwirrend, als die Moeglichkeit, dass er mit wenig Information zunaechst glueckt, mit etwas mehr fehlschlaegt, und mit vollstaendiger wieder glueckt, wie es im folgenden Fall passiert:

Datenbasis:

```
((unify *x *x))
```

1. Anfrage:

   (unify (& times *a *b) (& times *c *d)) mit
   leerer Bindungsumgebung ist erfolgreich

2. Anfrage:

wie oben, mit der Bindungsumgebung
((*a <- 4) (*b <- 3)) schlaegt fehl

3. Anfrage:
wie oben, mit der Bindungsumgebung
((*a <- 4) (*b <- 3) (*c <- 2) (*d <- 6)) ist
wieder erfolgreich.

### 4.1.3 Durchgriffe von der LISP-Ebene auf PROLOG

Neben der konventionellen Verwendung des PROLOG-Beweisers als geschlossenes interaktives System mit eigener Benutzeroberflaeche (Anfrage lesen - beweisen - Ausgabe der Antwortsubstitution - Erfolgs-/Misserfolgsmeldung - Anforderung von Alternativen), bietet sich in LISP/PROLOG-Vereinheitlichungen die Moeglichkeit an, ihn zur Loesung von Teilproblemen, die sich gut mit Horn-Klausel-Mengen beschreiben lassen, in LISP-Programmen zu verwenden.

In LOGLISP ist das sogar die urspruengliche Verwendungsform; die Benutzeroberflaeche ist der LISP-Toplevel.

Der Vergleich gliedert sich hier nach verschiedenen Aspekten des Durchgriffs, naemlich den zugrundeliegenden Schnittstellenkonzepten, der Form des zurueckgegebenen Werts und der Behandlung des Nondeterminismus.

a) Schnittstellenkonzepte

Fuer die Schnittstelle lassen sich drei verschiedene Konzepte erkennen:

1. Verwendung einer speziellen Beweiserfunktion, die als Parameter eine Konjunktion von Zielen erhaelt und deren Wert von LISP-Funktionen weiterverarbeitet werden kann.

2. Implementation von PROLOG-Prozeduren als LISP-Funktionen, die direkt aufgerufen werden koennen.

3. Ein objektorientierter Ansatz; der Beweiser wird als LISP-Flavor implementiert. Fuer eine Konjunktion von Zielen wird eine Instanz dieses Flavors erzeugt, an die eine Nachricht geschickt werden kann, um Loesungen fuer die Ziele zu erhalten.

b) Form des zurueckgegebenen Werts

Die einfachste Auskunft, die der Beweiser dem LISP-System geben kann, ist

1. ein Wahrheitswert (t oder nil), der angibt, ob ein Beweis von Zielen erfolgreich war. Das ist allerdings den Faehigkeiten eines solchen Beweisers, Substitutionen fuer die freien PROLOG-Variablen in den Zielen zu ermitteln, nicht angemessen. Darum gibt es in einigen Systemen die Moeglichkeit,

2. ein (oder mehrere) Antwortsubstitutionen fuer die PROLOG-Variablen, oder etwas komfortabler,

3. ein (oder mehrere) instanziierte Antwortmuster zurueckzugeben.

Das Antwortmuster ist ein beliebiger LISP-Ausdruck, der als Parameter beim Aufruf des Beweisers mitgegeben wird und PROLOG-Variablen aus den Zielen enthalten kann.

c) Behandlung des Nondeterminismus

Eine wesentliche Faehigkeit eines PROLOG-Beweisers ist es, nondeterministische Programme verarbeiten zu koennen. In den einzelnen Implementationen wurde dem, in Abhaengigkeit vom grundlegenden Schnittstellenkonzept, in unterschiedlichem Masse Rechnung getragen.
Falls vom PROLOG-Beweiser nur Wahrheitswerte zurueckgegeben werden, ist die Behandlung des Nondeterminismus natuerlich uninteressant.

Fuer Implementationen, die den Beweiser ueber eine Funktion aufrufen und den Wert dieser Funktionsapplikation weiterverarbeiten, gibt es folgende Loesungen:

1. One-Solution
Es wird nur die erste gefundene Loesung zurueckgegeben. Das ist im allg. nicht ausreichend.

2. All-Solutions mit Listen
Alle Loesungen werden im Resultat als Liste zurueckgegeben. Wenn mehr Loesungen existieren, als spaeter bearbeitet werden sollen, koennen Effizienzprobleme auftreten. Fuer Ziele mit unendlich vielen Loesungen ist dieser Ansatz ungeeignet.

3. N-Solutions mit Listen
Es wird eine beim Aufruf festgelegte Anzahl von Loesungen zurueckgegeben. Das ist sinnvoll, wenn annaehernd bekannt ist, wieviele Loesungen gebraucht werden. Es koennen allerdings auch Faelle auftreten, bei denen die Anzahl der benoetigten Loesungen a priori unbeschraenkt ist. In diesem Fall muss der Beweiser mehrfach mit schrittweise erhoehter Anzahl N aufgerufen werden. Dann muessen die bereits gefundenen Loesungen allerdings jedesmal neu berechnet werden.

4. Stroeme von Loesungen
Wenn der LISP-Teil der Vereinheitlichung Lazy Evaluation-Faehigkeiten hat oder Funktionen zur verzoegerten Berechnung spezieller Datenstrukturen geschrieben wurden, koennen auch unbegrenzt viele Loesungen als Strom zurueckgegeben werden. Die Kernfunktion von LM-Prolog arbeitet auf diese Weise ([Kahn 1982] und [Kahn & Carlsson 1984]). Das erscheint mir ein angemessener Ansatz zu sein.

Eine andere Moeglichkeit beliebig viele Antwortsubstitutionen mit LISP-Programmen zu verarbeiten bieten

5. Continuations

Tabelle 3: (LISP -> PROLOG) - Durchgriff

| | SCHNITTSTLE | | | RESULTAT | | | NONDETERMINISMUS | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BF | DIR | FLV | BOL | SUB | MUS | ONE | ALL | N | STR | CON | NACH |
| LOGLISP | x | | | | | x | x | x | x | | | |
| HCPRVR *) | x | | | nicht | | | bekannt | | | | | |
| Horne | x | | | nicht bekannt | | | | x | x | | | |
| QLOG *) | | x | | nicht | | | bekannt | | | | | |
| LM-Prolog | x | | x | | x | x | | | | x | | x |
| Symbolics Prolog | x | | | x | | | | | | | x | |
| Babylon-Prolog | | | x | | | x | | | | | | x |
| Salford LISP/PROLOG | x | | | x | | | | | | | | |
| LISLOG | x | | | | | x | | x | | | | |
| ConProlog | x | | | | x | | x | x | | | | |
| LISPLOG | x | | | | x | | | | x | | | |

*) Fuer QLOG und HCPRVR ist mir nicht bekannt, ob der Beweiser unabhaengig von der interaktiven Benutzeroberflaeche von LISP aus aufgerufen werden kann.

SCHNITTSTELLE:
BF = spez. Funktion zum Aufruf des Beweisers
DIR = Direktaufruf von PROLOG-Proz. als Funktionen
FLV = Implementation des Beweiser als LISP-Flavor

RESULTAT:
BOL = Wahrheitswert (t/nil)

SUB = eine oder mehrere Antwortsubstitutionen
MUS = ein oder mehrere instanziierte Antwortmuster

NONDETERMINISMUS:
ONE = Rueckgabe einer Loesung
ALL = - " - aller Loesungen
N = - " - einer festen Anzahl von Loesungen
STR = - " - eines Stroms von Loesungen
CON = Continuation als Parameter
NACH = Versand von Nachrichten zur Anforderung von Loesungen

---

Dieser Loesungsansatz faellt etwas aus dem Rahmen, da nicht der Wert der Beweiserfunktion fuer die Weiterverarbeitung in LISP-Programmen entscheidend ist. Statt dessen wird ein LISP-Ausdruck, die Continuation, der auf die Bindungen von PROLOG-Variablen zugreift, der Beweiserfunktion als Parameter uebergeben und am Ende des Beweises in der ermittelten Antwortsubstitution evaluiert. Das Resultat der Evaluation (nil/non-nil) bestimmt, ob weitere Loesungen benoetigt werden. Gegebenenfalls wird Backtracking ausgeloest.
Wenn die Verwendung von LISP-Praedikaten als Ziel erlaubt ist, kann eine solche Continuation durch eine entsprechende Ergaenzung der Liste der zu beweisenden Ziele leicht simuliert werden.

Falls die Schnittstelle mit Flavors implementiert wurde, bietet sich eine weitere Moeglichkeit an:

6. Versand von Nachrichten
Da hier der aufgerufene Beweiser als globales Objekt (Flavorinstanz) existiert, koennen von ihm nacheinander beliebig viele Loesungen mit Hilfe von Nachrichten angefordert werden.

LISPLOG im Kontext:

Mit der Verfuegbarkeit von N-Solutions und der Rueckgabe des Ergebnisses als Antwortsubstitution kann in LISPLOG ohne allzu grosse Umwege alles berechnet werden, was mit den anderen Schnittstellen moeglich ist. Fuer den Anwender ist es allerdings angenehmer, wenn das Resultat eines Beweises in Form instanziierter Antwortmuster zurueckgegeben wird.
Auch die Moeglichkeit, unbeschraenkt viele Loesungen vom Beweiser direkt anzuforden, ist aus Effizienzgruenden wuenschenswert und darueber hinaus benutzerfreundlich.

In der neuesten LISPLOG.2-Version [Dahmen 1986b] wurde beides realisiert. Um einen Strom von Loesungen zu erhalten, wird zunaechst ein Generator erzeugt. Ueber eine Zugriffsfunktion, die den Generator als Argument benoetigt, koennen dann Loesungen angefordert werden. Der Generator stellt dabei ein Objekt dar,

dass bei jedem Zugriff destruktiv veraendert wird. Dieser Ansatz ist am ehesten mit dem objektorientierten von Babylon-Prolog vergleichbar, bei dem Nachrichten zum Anfordern von Loesungen an die Beweiserinstanz geschickt werden koennen.
Mir erscheint es sinnvoll, andere Zugriffsfunktionen zu schreiben, die sich besser in den funktionalen Stil von LISP einfuegen. Zum Beispiel koennen Stroeme von Loesungen als abstrakter Datentyp dargestellt werden. Der Zugriff kann mit zwei Selektionsfunktionen erfolgen, einer um die erste Loesung zurueckzugeben, und einer um den Reststrom zu ermitteln, wobei intern der Generator veranlasst wird, eine weitere Loesung zu produzieren, sowie einem Praedikat, das bestimmt, ob ein Strom leer ist.
Die Verwendung von Flavors ist weniger geeignet, zum einen wuerde ein weiterer Formalismus eingefuehrt, zum anderen die Portabilitaet eingeschraenkt.

## 4.2 Implementation der Interpreter

Die Programmiersprache LISP weist als Implementationssprache fuer PROLOG eine grosse Flexibilitaet auf. Ein Teil der hier verglichenen Systeme dient sogar speziell zur experimentellen Implementation von PROLOG. Daher gibt es in vielen Bereichen Besonderheiten der einzelnen Implementationen, die von den anderen abweichen.
Fuer diesen Vergleich erscheint es zweckmaessig, unter der Vielzahl der speziellen Eigenschaften der Interpreter einige auszuwaehlen, denen in moeglichst vielen Implementationen Aufmerksamkeit geschenkt wurde.
Die Wahl fiel hier auf die Implementation der Kontrolle (Tiefensuche, Backtracking, ...), der Repraesentation von Variablenbindungen und der Darstellung der Datenbasis (-> Indexierung).
Eine ausfuehrliche Diskussion alternativer Implementationstechniken fuer diese Aspekte findet sich in [Kahn & Carlsson 1984].

### 4.4.1 Implementationstechniken fuer die Kontrolle

Die Implementation der Kontrolle des PROLOG-Beweisers ist in der zugrundeliegenden Literatur nur fuer 7 Interpreter genauer beschrieben.

LISPLOG arbeitet in der funktionalen Version [Boley, Kammermeier et al. 1985] mit einer Variante des Konzepts der 'downward success continuations' [Kahn & Carlsson 1984]. Der Beweiser laesst sich in zwei Hauptfunktionen unterteilen:

Eine Funktion, and-process, erhaelt eine Konjunktion von Zielen als Parameter, die bewiesen werden sollen; dazu uebergibt and-prozess das erste Ziel an die Funktion or-process. Die durchsucht die Datenbasis nach einer geeigneten Klausel, deren Rumpf sie mit dem Ziel unifizieren kann, und uebergibt die Praemissen, falls vorhanden, an einen neuen Aufruf von and-process. Die Suche in der Datenbasis wird durch rekursiven Aufruf von or-process durchgefuehrt.

Die restlichen Ziele des and-process werden ebenfalls an or-process uebergeben und von diesem zusammen mit den Praemissen an and-process wieder durchgereicht (=> Downward Success Continuation).

Der korekursive Aufruf der beiden Funktionen verlaeuft weiter, bis alle Praemissen des ersten Ziels bewiesen wurden. An dieser Stelle (=> Downward Success Continuation) wird der Beweisprozess mit den restlichen Zielen des ersten Aufrufs von and-process, die immer mit durchgereicht wurden in der ermittelten Bindungsumgebung fortgefuehrt (=> Downward Success Continuation).

Wenn Backtracking erfolgt, also fuer ein Ziel keine passende Klausel zu finden ist, liefern die Aufrufe von or- und insbesondere and-process als Wert nil zurueck, worauf der darueberliegenden or-process eine weitere passende Klausel fuer sein Ziel zu finden sucht.

Dieser Ansatz hat den Vorteil, dass er weitgehend dem funktionalen Programmierstil von LISP entspricht; da die noch nicht bewiesenen Ziele, die Bindungsumgebung und, im or-process, die verbliebenen Klauseln fuer das Backtracking beim Aufruf als (lokale) Parameter weitergegeben werden, erfolgt die gesamte Verwaltung des Beweisbaums und der Bindungsumgebung (auch im Backtrackfall) ueber den LISP-Stack.
Ein Nachteil ist, dass durch Fortfuehrung des Beweises einer Konjunktion von Zielen durch rekursiven Aufruf am Ende des Beweises des ersten Ziels, der LISP-Stack proportional zur Groesse des gesamten Beweisbaums anwachsen kann [Kahn & Carlsson 1984].
Ein weiterer Nachteil ist, dass Backtracking nur vom Inneren der korekursiven Aufrufkette von and-process und or-process erfolgen kann; wenn die Kontrolle nach gegluecktem Beweis wieder an die rufende Funktion zurueckgegeben wurde, ist die Anforderung weiterer Loesungen nicht mehr moeglich.

HCPRVR arbeitet ebenfalls mit Downward Success Continuations, allerdings erfolgt die Suche in der Datenbasis nicht durch Rekursion, sondern iterativ durch eine Schleife (or-process entfaellt). Die Liste der noch zu beweisenden Ziele und die Bindungsumgebung werden auch nicht als Parameter durchgereicht sondern in einem expliziten globalen Stack verwaltet. Dadurch ist der LISP-Stack, dessen Vergroesserung in vielen LISP-Implementationen problematisch ist, bedeutend weniger belastet. Der Preis ist der Verzicht auf einen klaren funktionalen Programmierstil; Backtracking ist ebenfalls nur vom Inneren der Aufrufkette der prove-Funktion (entspricht and-process) moeglich.

LM-Prolog verwendet in seinem Beweiser 'upward failure continuations' [Kahn & Carlsson 1984]. Zum Beweis einer Konjunktion von Zielen wird zunaechst das erste Ziel bewiesen. Im Gegensatz zur oben erlaeuterten Strategie werden hier jedoch die restlichen Ziele nicht mitgegeben, um am Ende des Beweises in der ermittelten Bindungsumgebung bewiesen zu werden; vielmehr werden ausser der Bindungsumgebung ein oder mehrere Continuations zurueckgeliefert (=> Upward), die aufgerufen werden koennen um weitere Bindungsumgebungen (Loesungen) zu erhalten, wenn der Beweis der restlichen Ziele in der ersten Bindungsumgebung fehlschlaegt (=> Failure). Durch geeignete Primitive zur Verwaltung der Continuations koennen die Resultate der Beweiserfunktionen als Strom von Bindungsumgebungen behandelt werden. So koennen auch nach Verlassen der Beweiserfunktionen beliebig viele Loesungen bearbeitet werden (vgl. 4.1.3).
Allerdings wurden bei der effizienten Implementation der Continuations Closure-aehnliche Konstrukte auf cdr-kodierten

Listen verwendet. Es stellt sich die Frage, wie schwierig eine effiziente Portierung auf andere LISP-Dialekte ist. Obwohl auf imperative Kontrollstrukturen verzichtet wurde, wird bei diesem Ansatz der LISP-Stack nicht mehr proportional zur Groesse des ganzen Beweisbaums belastet; er wird hoechstens so gross, wie die maximale Tiefe des Beweisbaums [Kahn & Carlsson 1984]. Statt dessen kann die Anzahl der Continuations, die mit der Bindungsumgebung zurueckgeliefert werden, die Groesse des Beweisbaums erreichen.

Babylon-Prolog arbeitet im wesentlichen Kern nicht mehr funktional, sondern verwendet Flavors und Iteration. Der Algorithmus baut auf dem Box-Modell von L. Byrd auf [Clocksin & Mellish 1981/84]. Eine Box entspricht hier einer Flavorinstanz. Zwei Methoden sind fuer die Kontrolle des Beweisgangs verantwortlich:
:prove-goal ist fuer die Suche in der Datenbasis zustaendig, verwaltet die Bindungsumgebung und steuert die Kommunikation zwischen einem Ziel und seinen Unterzielen. :prove-subgoals steuert die Verbindung der Box-Ein- und -Ausgaenge innerhalb einer Konjunktion von Zielen.
Die Methode :prove-goal hat einen Parameter, der die Werte try oder retry haben kann, entsprechend den Box-Eingaengen CALL und REDO. Als Wert kann success fuer den Box-Ausgang EXIT oder fail fuer FAIL zurueckgeliefert werden. Der Instanzvariable clauses werden beim ersten Betreten (durch den try-Eingang) die Klauseln zugewiesen, die moeglicherweise fuer das Ziel passend sind. Wenn die erste Klausel, deren Kopf mit dem Ziel unifizierbar ist, eine Regel ist, werden fuer die Praemissen eigene Flavorinstanzen erzeugt und mit :prove-subgoals bewiesen.
Die Bindungsumgebung wird dabei mit Hilfe eines globalen Stacks verwaltet.
Backtracking geschieht auf zwei Art und Weisen:
Wenn der Beweis der Unterziele fehlschlaegt, wird innerhalb der Instanz des Ziels durch Iteration eine andere passende Klausel ermittelt.
Tritt der Fehlschlag dagegen erst nach Verlassen der Instanz in dem nachfolgenden Ziel auf, wird die Instanz durch den retry-Eingang wieder betreten. In diesem Fall wird zuerst versucht, ueber die Unterziele eine andere Loesung zu erhalten. Falls solche nicht vorhanden sind, wird auch hier eine andere Klausel fuer das Ziel ermittelt.

:prove-subgoals steuert das Zusammenspiel zwischen dem success-Ausgang eines Ziels und dem try-Eingang des naechsten einerseits und dem fail-Ausgang eines Ziels und dem retry-Eingang des vorherigen andererseits. Dabei durchlaeuft es die Liste der Unterziele eines Ziels, die einer Instanz-Variable zugewiesen wurde, vorwaerts bzw. rueckwaerts.

Da sowohl die Bindungsumgebung als auch die Ziele in Instanzen global gespeichert werden und die maximale Tiefe einer Aufrufkette der oben beschriebenen Methoden proportional zur

Tiefe des Beweisbaums ist, duerfte die Groesse des LISP-Stacks keine Probleme machen.
Die Portabilitaet ist jedoch aufgrund der Verwendung von Flavors eingeschraenkt. In Anbetracht der Einbettung in das Babylon-System war sie wohl auch nicht das wesentliche Ziel.

Salford LISP/PROLOG wurde in der prozeduralen Programmiersprache FORTRAN geschrieben. Der Beweisvorgangs wird in traditioneller Form [Bruynooghe 1982] mit Hilfe von 'stack frames' zur Repraesentation des Beweisbaums und einem Trail zur Restauration der Bindungsumgebung beim Backtracking kontrolliert.

Fuer LISPLOG.2 wurde aus Effizienzgruenden (mangelnde Zeiteffizienz und Ueberlauf des LISP-Stacks) der Interpreter entrekursiviert. Der iterative Interpreter verwendet mehrere Stacks, einen 'goal stack', fuer Ziele, die noch bewiesen werden muessen, einen 'choice stack', fuer offene Backtracking-Punkte, und einen Trail zur Restauration beim Backtracking.

Nach der vorliegenden Dokumentation scheint LISLOG ebenfalls einen iterativen Ansatz unter Verwendung von Stacks und einem Trail zu verfolgen.

Auch in ConProlog ist die Beweisprozedur in der hier verwendeten Version CPL2 iterativ, unter Verwendung von einem eigenen Stack (durch ein Array implementiert), programmiert worden.
Motivation dafuer war die grosse Belastung des LISP-Stacks in der funktionalen Version CPL1.

LOGLISP arbeitet (breitenorientiert!) iterativ unter Verwendung eines Heaps. Dieser enthaelt als Elemente Konjunktionen von Zielen, die alle zum Beweis der urspruenglichen Ziele fuehren koennen ('implicit constraints'). Am Anfang enthaelt er nur die Konjunktion der vom Benutzer gestellten Ziele.
In jedem Beweisschritt wird nun ein Implicit Constraint ausgewaehlt. Falls es leer ist, also keine Ziele mehr zu beweisen sind, wird die zugehoerige Bindungsumgebung als Loesung in einer globalen Variable gespeichert und der Zyklus neu gestartet. Sonst wird eines der Ziele des Constraints ausgewaehlt, das durch Resolution vereinfacht werden soll. Fuer alle Klauseln der Datenbasis, deren Kopf mit diesem Ziel unifizierbar sind, werden nun neue Constraints gebildet, in denen nun anstelle des ausgewaehlten Ziels die Praemissen der Klausel stehen.
Der Beweiszyklus beginnt jetzt neu und faehrt fort, bis entweder die gewuenschte Zahl von Loesungen gefunden wurde, oder bis im Heap keine Constraints mehr vorhanden sind.
Jedes Implicit Constraint entspricht also den Blaettern eines (noch unvollstaendigen, moeglichen) Beweisbaums. Innere Knoten brauchen nicht dargestellt zu werden, da Backtracking nicht notwendig ist, denn alle Alternativen eines Oder-Knotens sind im Heap durch Implicit Constraints vorhanden sind.
Der Algorithmus bietet viele Moeglichkeiten, die Richtung des Beweisvorgangs zu steuern. Implicit Constraints koennen fuer die

Tabelle 4: Implementation der Kontrolle

| | DSC FCT | DSC ITE | UFC | ITERATIV STCK | ITERATIV HEAP | FLAV |
|---|---|---|---|---|---|---|
| LOGLISP | | | | | x | |
| HCPRVR | | x | | | | |
| LM-Prolog | | | x | | | |
| Babylon-Prolog | | | | | | x |
| Salford LISP/PROLOG | | | | x | | |
| ConProlog | | | | | ? | |
| LISPLOG.1 | x | | | | | |
| LISPLOG.2 | | | | | x | |

DSC FCT = Downward Success Continuations, rein funktional
DSC ITE = Downward Success Continuations, iterative Durchsuchung der Datenbasis, lokale Variable zur Speicherung der restlichen Klauseln (Backtrackpunkt)
UFC = Upward Failure Continuations
ITERATIV STCK = iterativer Interpreter mit globalem Stack zur Speicherung des Beweisbaums (-> Tiefensuche, Backtracking)
ITERATIV HEAP = iterativer Interpreter mit globalem Heap zur Speicherung (der Blaetter) von mehreren Beweisbaeumen (-> Breiten/Bestensuche)
FLAV = Flavorinstanzen und Methoden sowie Iteration

Auswahl mit einem Gewicht versehen werden; auch die Auswahl des zu resolvierenden Ziels in einem Constraint ist prinzipiell

steuerbar. Die Festlegung des sogenannten Deduktionsfensters bestimmt Begrenzungen fuer den Suchvorgang, z.B. die maximale Tiefe des Suchbaums, die maximale Verzweigungsrate und anderes mehr.
Ein 'PROLOG-Modus', der Tiefensuche simuliert, kann ebenfalls gewaehlt werden.

In QLOG wird bei jedem Aufruf einer PROLOG-Prozedur eine LISP-Funktion gleichen Namens aufgerufen, die diese Prozedur repraesentiert. Von dort aus wird die eigentliche Beweisfunktion mit den Klauseln der Prozedur als Parameter aufgerufen.
Zur Implementation von Backtracking wird jedoch ausser dem LISP-Stack noch ein spezieller Kontrollstack verwendet.
Detaillierte Information zur Implementation der Kontrolle stand jedoch weder fuer QLOG noch fuer Horne und LISLOG zur Verfuegung; Symbolics Prolog arbeitet nicht mit einem Interpreter, sondern nur mit einem Compiler, fuer den mir allerdings auch keine Implementationsdetails bekannt sind.

LISPLOG im Kontext:

Die Implementation der Kontrolle mit Hilfe von Downward Success Continuations, wie in LISPLOG.1, ist wegen der hohen Belastung des LISP-Stacks fuer praktikable Systeme leider nicht geeignet. Ein iterativer Interpreter, wie er von LISPLOG.2 verwendet wird, bringt entscheidenden Effizienzgewinn und ermoeglicht darueber hinaus, durch die globale Speicherung der Kontrollinformation, beim Durchgriff von LISP nach PROLOG einen unendlichen Strom von Loesungen zurueckzugeben (vgl. Kapitel 4.1.3c)4. und 6.). Auf der anderen Seite geht dabei doch einiges an Transparenz verloren.

Falls der tiefenorientierte Ansatz von PROLOG beibehalten werden soll, bieten sich als Alternativen ein flavororientierter Ansatz und Upward Failure Continuations an.
Gegen die Verwendung von Flavors spricht vor allem die Einschraenkung der Portabilitaet.
Bei Upward Failure Continuations ist entscheidend, ob fuer die Continuations eine Repraesentation gefunden werden kann, die zum einen portabel ist, zum anderen die Laufzeiteffizienz nicht zu sehr beeintraechtigt.

### 4.2.2 Repraesentation von Terminstanzen und Bindungsumgebungen

Im Verlauf des Beweisprozesses muessen von einem PROLOG-Interpreter Terme behandelt werden, deren Variablen durch die in der Unifikation ermittelte Variablensubstitution ersetzt wurden. Die Methoden zur Repraesentation solcher Terminstanzen lassen sich im allgemeinen in zwei Klassen einteilen:
Die einen beruhen auf 'structure sharing' (SS), das zuerst von Boyer und Moore beschrieben wurde [Boyer & Moore 1972]. Die andere Klasse verwendet einen 'structure copying'- oder 'non structure sharing'-Ansatz (NSS).

Bei Structure Sharing-Ansaetzen wird eine Terminstanz durch ein Paar dargestellt, das aus dem Grundgeruest der Terminstanz (der urspruengliche Term mit den Variablensymbolen, 'sceleton') und einer Bindungsumgebung (entspricht der Variablensubstitution, 'environment') besteht. Allen Terminstanzen eines Terms ist daher das strukturelle Grundgeruest gemeinsam.
Da die Bindungsumgebungen fuer jeden Term spezifisch sind, koennen sie ohne allzu grossen Effizienzverlust einfach als Assoziationslisten repraesentiert werden. In anderen Ansaetzen wird eine Klausel beim Einfuegen in die Datenbasis umgeformt, wobei ihre Variablen durch Indizes ersetzt werden. So koennen fuer die Bindungsumgebungen Vektoren verwendet werden.
Der Zugriff auf die Bindungsumgebungen kann entweder mit Hilfe eines einfachen Zeigers oder ebenfalls durch einen Index auf einen Vektor von Bindungsumgebungen erfolgen.
Schliesslich kann noch ein Basisindex(Zugriff auf die Bindungsumgebung)-Offset(Zugriff auf die einzelne Variable)-Verfahren verwendet werden.

Bei Non Structure Sharing-Ansaetzen wird die Grundstruktur von Terminstanzen gleicher Terme nicht in vollem Umfang geteilt.
Bei einfachen Implementationen soll die Aufteilung der Bindungsumgebung umgangen werden. Da aber waehrend des Beweises Variablen gleichen Namens, z.B. aufgrund mehrfacher Verwendung einer Klausel, durchaus an verschiedene Werte gebunden werden koennen, muessen die Variablen zur Vermeidung des Namenskonflikts umbenannt werden. Dazu wird im allgemeinen die ganze Klausel, deren Kopf unifiziert werden soll, vor der Unifikation kopiert und ihre Variablen durch Umbennenungen ersetzt.
Der Zeit- und vor allem Speicheraufwand ist dabei natuerlich groesser. Verbesserungen lassen sich beispielsweise dadurch erreichen, dass nicht gleich die ganze Klausel umbenannt wird, sondern immer nur die Teile, fuer die es beim augenblicklichen Stand des Beweises notwendig ist. Konstante Terme muessen nie kopiert werden. Weitentwickelte NSS-Ansaetze nennen die Variablen in Termen nicht einfach um, sondern ersetzen sie durch Zeiger auf eine (fuer alle gleichnamigen Variablen einer Klausel gleiche) Speicherzelle, die ihrerseits einen Zeiger auf den Wert der Variablen enthaelt. Dadurch verbessert sich das Zugriffsverhalten auf diese Terme.

Eine ausfuehrliche Eroerterung der Vor- und Nachteile von SS und NSS wuerde hier zu weit fuehren, vor allem, da einige der hier behandelten Implementationen weit von der Effizienz entfernt sind, die mit dem gewaehlten Ansatz erreicht werden koennte. Keiner der Interpreter scheint vor allem zwischen sogenannten lokalen und globalen Variablen zu unterscheiden. Nur die Compiler von LM-Prolog und wahrscheinlich Symbolics Prolog beruecksichtigen die Unterschiede zur Optimierung. Diskussionsmaterial findet sich z.B. in [Mellish 1982], [Bruynooghe 1982] und [Kluzniak & Szpakowicz 1985].

HCPRVR verwendet einen traditionellen SS-Ansatz, bei dem die Bindungsumgebung in Form einer A-Liste gespeichert und ueber einen Zeiger erreicht wird.
LOGLISP verwendet zum Zugriff eine arrayartige Struktur. Die Verwendung von Breitensuche erfordert das haeufige Wechseln der Bindungsumgebung fuer verschiedene Aeste im Suchbaum. Nicht benoetigte Bindungsumgebungen werden durch Listen dargestellt.
Auch LISLOG und LM-Prolog benutzen Structure Sharing, die Bindungsumgebung wird jedoch in Form von kleinen Vektoren dargestellt, die in LM-Prolog als cdr-kodierte Listen implementiert sind. Der Zugriff auf die Bindungsumgebung erfolgt durch Zeiger.
Compiliertes LM-Prolog verwendet dagegen Non Structure Sharing (vgl. Kapitel 4.3 ueber Compilationstechniken).

ConProlog verwendet laut [Knoepfler & Hotop 1985] ebenfalls Structure Sharing. Variablen werden in der Form '(V n)' repraesentiert, wobei n ein Index in ein globales Array von Zeigern auf die im Laufzeit-Stack eingetragenen Unifikatoren ist. Die Umkodierung der Variablen aus der externen Notation in diese Form erfolgt jedoch erst beim Zugriff auf eine Klausel. (Es stellt sich die Frage, ob dabei die Klausel doch kopiert wird.) Diese Umkodierung erfordert einen grossen Zeitaufwand, daher ist fuer CPL3 vorgesehen, den Index auf das globale Array fuer die Bindungsumgebung nicht mehr explizit in die interne Repraesentation einer Variablen zu schreiben, sondern eine Basisindex-Offset-Form zu verwenden (vgl. oben), sodass die Umkodierung schon beim Einfuegen einer Klausel erfolgen kann.

Ein einfacher NSS-Ansatz findet sich in Babylon-Prolog. Hier wird die ganze Klausel vor der Unifikation kopiert, wobei die Variablen umbenannt werden. Die Bindungsumgebung wird mit Hilfe von Arrays implementiert.

In Salford LISP/PROLOG wurde das NSS-Verfahren optimiert. PROLOG-Variablen werden nicht umbenannt, sondern durch Zeiger auf Speicherzellen repraesentiert, die Platz fuer einen Zeiger zum Wert der Variablen enthalten.
Die Argumente eines Ziels werden separat an den zu unifizierenden Klauselkopf uebergeben. So muessen Klauselkoepfe, die im Toplevel nur Variablen und einfache Konstanten haben, nicht kopiert werden.

Aus der zugrundeliegenden Literatur scheint ausserdem hervorzugehen, dass ein Unterziel einer Klausel erst kopiert wird, wenn es mit einem Klauselkopf unifiziert werden soll.

LISPLOG.1 hat den einfachsten und ineffizientesten Ansatz, Terminstanzen zu repraesentieren. Vor der Unifikation wird die gesamte Klausel kopiert, wobei die Variablen umbenannt werden. Die Bindungsumgebung wird durch eine einzige globale A-Liste repraesentiert.
In LISPLOG.2 wurde das Verfahren optimiert. Zum einen wird die Bindungsumgebung jetzt durch ein Array repraesentiert, das fuer jede Klausel einen Zeiger auf eine kleine A-Liste besitzt.
Ausserdem wurde versucht, das Kopieren von Klauseln soweit wie moeglich herauszuzoegern, um unnoetiges Kopieren zu vermeiden. Klauselkoepfe werden nicht mehr vollstaendig kopiert, sondern nur diejenigen ihrer Variablen, an die Terminstanzen des zugehoerigen Ziels gebunden werden sollen, sowie Terme des Kopfes, die an Variablen des zugehoerigen Ziels gebunden werden sollen.
Die Unterziele einer Klausel werden erst kopiert, wenn sie mit einem Klauselkopf unifiziert werden sollen. Dabei werden gebundene Variablen nicht umbenannt, sondern durch ihre ultimate Instanziierung ersetzt (das heisst, dass Variablen, die in dem unmittelbarem Wert dieser Variablen vorkommen, weiter instanziiert werden usw.). Der Aufwand fuer diese Instanziierung ist zwar sehr hoch, aber dafuer gewinnt man bei spaeteren Zugriffen auf die Terminstanzen oder Teilen davon Zeit.
Ein Nachteil dieses Verfahrens im Vergleich zum obengenannten (Salford LISP/PROLOG) ist, dass die Instanziierung von zwei gleichnamigen Variablen zweimal erfolgen muss (oben muss nur in der gemeinsamen Speicherzelle ein Pointer ausgerichtet werden), und dass ungebundene Variable weiterhin nur umbenannt werden. So muessen Terminstanzen, die schon einmal kopiert wurden, in spaeteren Beweisschritten ein weiteres Mal kopiert werden, um solche umbenannten Variablen durch evtl. ermittelte Bindungen zu ersetzen.

Fuer Symbolics Prolog liegen keine genauen Informationen ueber die Repraesentation von Terminstanzen vor; der einzige Hinweis besagt, dass fuer PROLOG-Variablen ein eigener Datentyp geschaffen wurde.
Auch bei QLOG und Horne liegen keine ausreichende Information vor.

Tabelle 5 ist in zwei Teile unterteilt, einen fuer Structure Sharing und eine fuer Non Structure Sharing. In dem jeweiligen Teil ist die Art der Repraesentation der Bindungsumgebung angegeben, sowie bei Non Structure Sharing, ob Optimierungen vorgenommen wurden.

LISPLOG im Kontext:

Tabelle 5: Repraesentation von Terminstanzen

| | STRUCTURE SHARING | | | NON STRUCTURE SHARING | | | |
|---|---|---|---|---|---|---|---|
| | PTR | INDX | ALI VECT | G-ALI | G-ARRAY | PTR | OPT |
| LOGLISP | | x | x | | | | |
| HCPRVR | x | | x | | | | |
| LM-Prolog (interpret.) | x | | x | | | | |
| LM-Prolog (compiliert) | | | | | | x | x |
| Babylon-Prolog | | | | | x | | |
| Salford LISP/PROLOG | | | | | | x | x |
| LISLOG | x | | x | | | | |
| LISPLOG.1 | | | | x | | | |
| LISPLOG.2 | | | | | x | | x |

STRUCTURE SHARING:
PTR = Zugriff auf eine Bindungsumgebung ueber einen Zeiger
INDX = Zugriff auf eine Bindungsumgebung ueber einen Index in einen Vektor von Bindungsumgebungen
ALI = Eine Assoziationsliste fuer jede Klausel
VECT = Ein Vektor fuer jede Klausel
G-ARRAY = Ein globales Array fuer alle Bindungen

NON STRUCTURE SHARING
G-ALI = Eine globale Assoziationsliste fuer alle Bindungen
G-ARRAY = Ein globales Array fuer alle Bindungen
PTR = Fuer jede Variable eine Speicherzelle mit einem Pointer auf die jeweilige Bindung

OPT = Reduktion des Kopieraufwandes bei NSS

Der NSS-Ansatz von LISPLOG.2 zeichnet sich durch Originalitaet aus, der Kopieraufwand wurde hier weitgehend minimiert.
Das Verfahren laesst sich weiter verbessern, indem beim Kopieren der Unterziele Variablen durch den Verweis auf eine Speicherzelle ersetzt werden, die den Wert der Variablen enthaelt (vgl. Salford LISP/PROLOG). Wieviel Zeitverlust das zusaetzliche Verfolgen dieser Zeiger verursacht, muss in der Praxis getestet werden.
In einem compilativen Ansatz kann der Kopieraufwand noch reduziert werden, wenn in den Zielen Terme, die keine Variablen enthalten, gekennzeichnet werden.

Die Alternative fuer ein rein interpretatives System ist Structure Sharing; versuchsweise wurde ein solches Verfahren bereits eingebaut und brachte Effizienzverbesserungen um den Faktor Zwei.

### 4.2.3 Indexierung der Datenbasis

Zum Beweis eines Zieles muss ein PROLOG-Interpreter Klauseln aus der Datenbasis finden, deren Koepfe mit dem Ziel unifizierbar sind. Wenn dafuer die gesamte Datenbasis linear durchsucht werden muesste, waere der Interpreter fuer groessere Anwendungen viel zu langsam.
Es ergibt sich daher die Notwendigkeit, die Anzahl der fuer die Unifikation in Frage kommenden Klauseln durch einfachste Operationen stark einzuschraenken.
Ueblicherweise wird dazu die Datenbasis indexiert, d.h. es wird eine Zugriffsstruktur aufgebaut, mit der ueber einen oder mehrere Schluessel (Indizes) auf eine geeignete Menge von Klauseln zugegriffen werden kann.

Als einfachstes Verfahren bietet sich die sogenannte primaere Indexierung (PI) an, die die Klauseln nach den Praedikatsnamen in Klauselkopf aufteilt. Die so entstandenen Klauselmengen mit gleichem Praedikatsnamen werden auch Prozeduren genannt.
Da alle hier betrachteten LISP/PROLOG-Vereinheitlichungen im Klauselkopf nur atomare Konstanten als Praedikatsnamen erlauben und die Praedikatsnamen von Zielen spaetestens beim Aufruf zu atomaren Konstanten instanziiert sein muessen, ist diese Unterteilung recht einfach zu implementieren. Der Zugriff erfolgt meist ueber einen Eintrag in der LISP-Property-Liste des Praedikatsnamens.
Primaere Indexierung ist fuer diese Vereinheitlichungen Standard.

Fuer grosse Prozeduren erscheint dieses Verfahren allerdings noch ungenuegend. Verschiedene sekundaere Indexierungsverfahren versuchen die Klauselmengen weiter einzuschraenken.
Automatische Verfahren fuer sekundaere Indexierung (SI) legen ein oder mehrere weitere Komponenten der Klauselkoepfe fest, die als Schluessel dienen sollen.
Manuelle Verfahren erlauben es, die Komponenten, die zur Indexierung dienen sollen, fuer jede Prozedur einzeln festzulegen.
Sekundaere Indexierung ist etwas problematischer als die primaere. Zum einen koennen die Komponenten, die als Schluessel dienen sollen, sowohl im Klauselkopf als auch in Zielen, Variablen oder komplexe Terme sein. Dadurch wird die Bestimmung der passenden Klauseln fuer einen sekundaeren Schluessel entweder zeit- oder speicherplatzintensiv (vgl. [Bernardi 1986]). Die Verwendung von mehreren Schluesseln wirft ebenfalls Probleme auf. Je nach Implementation sind z.B. aufwendige Schnittmengenbildungen erforderlich (bei getrennter Abspeicherung der Klauselmengen fuer jeden Schluessel mit anschliessender Schnittmengenbildung, vgl. [Bernardi 1986]) oder es muessen komplexere Zugriffsoperationen durchgefuehrt werden (Konstruktion eines Gesamtschluessels aus den Einzelschluesseln).

LOGLISP verwendet ein automatisches SI-Verfahren. Es wird nur fuer Prozeduren verwendet, deren Klauselkoepfe keine Variablen

enthalten (sog. Datenprozeduren). Hier kann die Unifikation auf die Klauseln eingeschraenkt werden, die alle atomaren Konstanten des jeweiligen Ziels im Klauselkopf enthalten.
Unter der Annahme, dass Prozeduren mit vielen Klauseln hauptsaechlich zur Abspeicherung grosser Faktenmengen verwendet werden, die meist keine Variablen enthalten, erscheint dieses Verfahren sinnvoll.
Die Klauseln werden dafuer in der LISP-Property-Liste von jeder atomaren Konstante abgespeichert, die in ihrem Kopf auftritt.
Das Zugriffsverfahren wird in der hier zugrunde liegenden Literatur leider nicht genauer beschrieben, wahrscheinlich erfolgt eine Schnittmengenbildung (mit linearem Aufwand) der geeigneten Klauselmengen fuer jede atomare Konstante in einem Ziel.

Auch Symbolics Prolog verwendet ein automatisches Indexierungsverfahren, das laut Manual 'highly efficient', also vermutlich sekundaer ist.

Die SI-Verfahren von QLOG und ConProlog werden nicht genau beschrieben, vermutlich sind es automatische Verfahren.

LM-Prolog besitzt ein recht flexibles Verfahren fuer SI. Jeder Prozedur kann eine Liste von Mustern zugeordnet werden, die die Komponenten der Klauselkoepfe kennzeichnen, die als Schluessel dienen koennen. Die Muster sind (moeglicherweise komplexe) S-Ausdruecke, die jeweils genau ein ausgezeichnetes Element enthalten, das die Schluesselkomponente bezeichnet.
Wenn diese Komponente in einem Ziel gebunden ist, kann sie als Schluessel benutzt werden. Fuer ein Ziel wird allerdings maximal ein Schluessel verwendet.
Der Zugriff erfolgt ueber Hashtabellen, fuer die auf der LISP-Maschine bereits Verfahren vordefiniert sind.
Falls fuer eine Prozedur keine Muster angegeben wurden, wird keine SI durchgefuehrt, was besonders bei kleinen Prozeduren sinnvoll ist, bei denen durch die Zugriffsoperation sogar Zeit verloren gehen koennte ('break-even point' bei etwa 25-120 Klauseln).

Auch fuer Horne koennen Komponenten von Klauselkoepfen, die sich als Schluessel eignen, fuer jede Prozedur gekennzeichnet werden.
Anders als in LM-Prolog koennen hier auch mehrere Schluessel fuer ein Ziel gleichzeitig zum Zugriff auf passende Klauseln verwendet werden (so legt es zu mindest [Frisch et al. 1982] nahe). Die Technik, die fuer das sukzessive Hashing verwendet wird, wurde leider nicht genauer beschrieben.

In LISPLOG kann fuer die SI eine Toplevel-Komponente der Klauselkoepfe pro Prozedur als Schluessel festgelegt werden. Der Zugriff erfolgt hier nicht ueber Hashtabellen, sondern mit Hilfe von Binaerbaeumen.
Bei Bedarf kann auch hier auf die sekundaere Indexierung verzichtet werden.

Tabelle 6: Indexierung

| | AUTO | MANU | SCHLUESSEL ONE | MORE | ALTN | OPT |
|---|---|---|---|---|---|---|
| LOGLISP | x | | | x | | |
| Horne | | x | | ? | | ? |
| QLOG | ? | | nicht bekannt | | | |
| LM-Prolog | | x | | | x | x |
| Symbolics Prolog | ? | | nicht bekannt | | | |
| ConProlog | ? | | nicht bekannt | | | |
| LISPLOG | | x | x | | | x |
| Salford LISP/PROLOG (compiliert) | x | | | | x | |

AUTO = automatische sekundaere Indexierung
MANU = Schluesselkomponente zur Indexierung fuer jede Prozedur per Hand waehlbar

SCHLUESSEL
ONE = genau eine festgelegte Komponente wird fuer den Zugriff verwendet
MORE = mehrere Komponenten koennen fuer den Zugriff verwendet werden
ALTN = genau eine Komponente kann aus mehreren Alternativen fuer den Zugriff ausgewaehlt werden
OPT = Das sekundaere Indexierungsverfahren kann abgeschaltet werden

Fuer Babylon-Prolog und HCPRVR werden keine sekundaeren Indexierungsverfahren erwaehnt, Salford LISP/PROLOG verwendet

sekundaere Indexierung nur im Compiler. Durch die dynamische Compilation ist bekannt, an welchen Stellen aufrufende Ziele Konstanten haben. Zusammenhaengende Bloecke von Klauseln, die dort keine Variablen enthalten, werden nach einer (oder mehreren?) dieser Stellen sortiert. Passende Klauseln koennen dann durch binaere Suche gefunden werden. Je kleiner allerdings diese zusammenhaengenden Bloecke sind, desto schlechter schneidet dieses Verfahren gegenueber solchen mit zusammenhaengender Zugriffsstruktur, wie in LISPLOG oder LM-Prolog ab. (vgl. Kapitel 4.3 ueber Compilationstechniken).

LISPLOG im Kontext:

Mit der freien Wahl der Argumentstelle und der Moeglichkeit die Indexierung abzuschalten, bietet LISPLOG ein flexibles Verfahren zur sekundaeren Indexierung.

Muster zur Wahl der Indexierungsstelle, wie in LM-Prolog, bieten dem Benutzer eine weitere Moeglichkeit zur Steuerung der Indexierung. Allerdings vergroessert sich dadurch auch der Aufwand fuer den Benutzer.

Alternative Argumentstellen, die verwendet werden, wenn in einem Ziel an der ersten Indexierungsstelle eine Variable steht, sind eine weitere Verbesserungsmoeglichkeit, die die Wahrscheinlichkeit erhoeht, dass die Indexierung tatsaechlich ausgenutzt wird.

Leider liegt bislang noch keine weitere Information ueber die Verfahren von Horne und LOGLISP vor, die mehr als einen Schluessel zur Indexierung erlauben.
Sobald genauere Angaben hierueber vorliegen, sollte geprueft werden, wie gut die Laufzeit- und Speicherplatzeffizienz der Verfahren ist.

Das Verfahren von Salford LISP/PROLOG ist zwar ausgesprochen originell, es stellt sich aber die Frage, ob der relativ hohe Aufwand zur seiner Implementation in LISPLOG durch den zu erwartenden Effizienzgewinn gerechtfertigt ist.

Gegen die Verwendung von Hashtabellen, wie in LM-Prolog oder Horne spricht die Einschraenkung der Portabilitaet.

## 4.3 Compilationstechniken

Fuer sechs der behandelten LISP/PROLOG-Vereinheitlichungen werden in der zugrundeliegenden Literatur Compiler erwaehnt.

In [Schrag 1984] werden Vorueberlegungen fuer einen Compiler fuer LOGLISP beschrieben; das Papier lag leider nur auszugsweise vor, sodass mir ueber eine tatsaechliche Implementation nichts bekannt ist.

Auch ueber den Compiler von Horne liegt nur wenig Information vor [Frisch et al. 1982]. Horne's Klauseln werden in LISP-Funktionen uebersetzt, wodurch eine tiefere Einbettung der PROLOG-Implementation in LISP erfolgt. Darueberhinaus wurde die Verwaltung der Variablen verbessert, sodass compiliertes Horne um einen Faktor zwei bis vier schneller ist, als interpretiertes Horne.

Fuer Symbolics Prolog existiert nur ein Compiler. Er arbeitet allerdings inkrementell, sodass ein schneller Entwicklungszyklus moeglich ist. Auch zur Laufzeit koennen noch Aenderungen mit assert/retract vorgenommen werden.
Der Benutzer kann auf die Optimierung der Compilation, z.B. durch Angabe von Indexierungsmustern oder 'mode declarations', keinen Einfluss nehmen; nichtsdestotrotz ist Symbolics Prolog ausgesprochen schnell.
Ueber die Implementationstechniken des Compilers liegen keine genaueren Informationen vor.

LM-Prolog besitzt einen sowohl ausgefeilten als auch ausgefallenen Compiler.
Der erzeugte LISP-Code arbeitet wieder mit Upward Failure Continuations. Jede PROLOG-Prozedur wird in eine LISP-Funktion uebersetzt, die in einer eigenen 'stack group', d.h. in einem eigenen LISP-Namensraum arbeitet. Die Funktion ermittelt eine Loesung fuer die zugehoerige Anfrage, kann aber im Backtrackfall als Continuation in ihrer Stack Group wieder aktiviert werden.
Im Gegensatz zum Interpreter arbeitet compiliertes LM-Prolog mit einem optimierten Non Structure Sharing-Verfahren zur Repraesentation von Termen. Dabei konnten spezielle Datenstrukturen von LISP Machine LISP verwendet werden, z.B. Speicherzellen, die nur eine Adresse enthalten, sog. 'locatives', um den Zeiger auf die Wertbindung einer Variablen zu repraesentieren und unsichtbare Zeiger, um von verschiedenen Stellen (mehrmaliges Auftreten einer Variable) auf diese Speicherzellen zuzugreifen. Im erzeugten LISP-Code werden PROLOG-Variablen einfach durch LISP-Variablen ersetzt, die einen unsichtbaren Zeiger auf die Speicherzelle mit der Wertbindung enthalten. So wird im allg. nur der LISP-Stack der jeweiligen Stack Group durch die Variablenbindungen belastet.
Der Compiler fuehrt eine Reihe von Optimierungen durch, z.B. um die unnoetige Schaffung von Stack Groups zu vermeiden. Die Unifikation wurde, je nach Version, microcodiert oder unter

Beruecksichtigung der Klauselkoepfe partiell evaluiert.
Der Benutzer kann die Compilation durch eine Reihe von Optionen beeinflussen, z.B. kann er angeben, ob Prozeduren statisch oder dynamisch sind, ob sie deterministisch sind oder Backtracking moeglich ist, er kann Muster fuer moegliche Ziele a la 'mode declarations' angeben und bestimmen, dass Prozeduren, vergleichbar mit LISP-Macros, offen compiliert werden.

In Salford LISP/PROLOG wird dynamisch compiliert. Wenn im Verlauf eines Beweises eine PROLOG-Prozedur aufgerufen wird, bestimmt Salford LISP/PROLOG zuerst den Typ der Toplevel-Argumente (Zahl, Atom, Struktur, leere Liste oder Variable) und compiliert sie dann, speziell fuer diesen Typ von Argumenten optimiert. Weitere Beweise von Zielen dieses Typs erfolgen dann bedeutend schneller, Ziele anderen Typs loesen eine weitere Uebersetzung aus. Dieses Verfahren eignet sich natuerlich nur fuer Prozeduren, die nur auf wenige verschiedene Weisen aufgerufen werden.
Aufgrund der Kenntnis des Typs der Argumente werden eine Reihe von Optimierungen vorgenommen. Zum Beispiel wird die Anzahl der mit dem Typ kompatiblen Klauseln eingeschraenkt und die zur sekundaeren Indexierung verwendete(n) Argumentstelle(n) kann(koennen) so gewaehlt werden, dass in den Zielen hier keine Variable steht (vgl. Kapitel 4.2.3 ueber Indexierungsverfahren).
Ausserdem werden Variablen, die direkt mit Eingabeargumenten assoziiert werden koennen, bei wiederholtem Auftreten schon waehrend der Compilation durch entsprechende Referenzen ersetzt.

LISPLOG besitzt fuer die Version .1 einen kleinen Compiler fuer Prozeduren [Herr 1985]. Er zieht seinen Hauptvorteil aus der partiellen Evaluation des Unifikationsalgorithmus mit den Klauselkoepfen.
In [Herr 1986] wird eine Verbesserung dieses Compilers beschrieben, bei der eine iteratives Laufzeitsystem erstellt wurde und die Repraesentation der Variablen durch Structure Sharing mit einem Basisindex-Offset-Verfahren erfolgt.
Fuer LISPLOG.2 wurde der Compiler nicht uebernommen, da der zusaetzliche Speicherbedarf durch die partielle Evaluation in keinem Verhaeltnis zur erzielten Geschwindigkeitssteigerung stand.

## LISPLOG im Kontext:

In [Herr 1986] werden viele der Schwierigkeiten beschrieben, die LISP als Implementations- und Zielsprache eines Compilers fuer PROLOG, insbesondere fuer die traditionellen Optimierungen aufweist.

Vor allem LM-Prolog gibt einige Anregungen, wie dennoch eine Effizienzsteigerung durch Compilation erreicht werden kann.

Ob allerdings der ganze Ansatz, mit Upward Failure Continuations, portabel gemacht werden kann, sei dahingestellt. Er haette den Vorteil einer ueberwiegend funktionalen und transparenten Implementation.

In LISPLOG wurde zur Verbesserung der Transparenz von PROLOG-Programmen bewusst auf den allgemeinen Cut verzichtet und nur der initiale Cut vor der ersten Praemisse erlaubt. Dadurch sind aber auch die Effizienzsteigerungen, die der allgemeine Cut ermoeglicht, nicht mehr zugaenglich. Eine Alternative, besonders bei einem compilativen Ansatz, stellt die Deklaration deterministischer Prozeduren dar. Sie ist bedeutend verstaendlicher, wenn auch nicht so maechtig wie der Cut.

Mit einem Compiler kann auch der Kopieraufwand fuer ein NSS-Verfahren verringert werden.

Ein Nachteil der partiellen Evaluation der Unifikation mit Hilfe von Macros ([Herr 1985], [Herr 1986]) ist die Trennung vom eigentlichen Unifikationsalgorithmus; Aenderungen fallen dadurch nicht mehr so leicht.. Ein automatischer partieller Evaluator fuer LISP-Programme [Kahn 1983] wuerde das Problem loesen.

Offene Compilation bietet in verschiedenen Abstufungen weitere Effizienzsteigerungen:
Mit relativ wenig Aufwand kann ein Verfahren entwickelt werden, das fuer ein Ziel zur Definitionszeit (mit weiterer Wartung beim Einfuegen neuer Klauseln) durch Unifikation die Menge der Klauseln einschraenkt, mit denen es zur Laufzeit resolviert werden kann. Dies ist eine moegliche Ergaenzung/Ersatz zur Klauselindexierung.
In einem weiteren Schritt kann das Ergebnis der Unifikation teilweise gespeichert werden; sie kann dann zur Laufzeit fortgesetzt werden. Dieses vorausberechnete Ergebnis der Unifikation benoetigt, fuer ein Ziel und eine Klausel betrachtet, wahrscheinlich weniger Speicherplatz als die partielle Evaluation, wie sie in [Kahn & Carlsson 1983], [Herr 1985] und [Herr 1986] beschrieben wird.
Fuer die Teile der Klauselkoepfe bzw. Ziele, die erst zur Laufzeit behandelt werden, kann dann noch eine partielle Evaluation der Unifikation erfolgen.
In LM-Prolog wird darueber hinaus noch vermieden, dass fuer den Beweis des Ziels eine neue Stack Group aufgerufen wird. Statt dessen wird in der alten 'inline code' erzeugt.
In jedem Fall sollte die offene Compilation nur als Option zur Verfuegung stehen, da die dabei generierten Datenstrukturen einen ziemlich grossen Speicherplatz benoetigen. Am wenigsten Probleme bereitet hier der erste Fall.

Wie schon im Kapitel 4.2.3 ueber Indexierung bemerkt wurde, stellt sich bei einer dynamischen Compilation wie in Salford LISP/PROLOG fuer LISPLOG die Frage, ob der Entwicklungsaufwand durch den zu erwartenden Effizienzgewinn gerechtfertigt ist.

## 5. LISPLOG im Kontext

LISPLOG gehoert, wie z.B. auch LOGLISP und HCPRVR bzw. Horne, zu der Klasse von LISP/PROLOG-Vereinheitlichungen, die sich durch Portabilitaet, Flexibilitaet und Transparenz auszeichnen.
Fuer eine ueberwiegend experimentelle Sprache, die in Richtung eines adaequaten Werkzeugs zum Entwurf von Expertensystemen entwickelt werden und fuer moegliche Anwender aus dem SFB 314 auf verschiedenen Maschinen offen sein soll, sind dies wichtige Eigenschaften, die weiter gepflegt werden sollten.

Dadurch sind der Effizienz natuerliche Grenzen gesetzt. Innerhalb dieser Grenzen ist die Implementation mit iterativem Interpreter, verfeinertem Verfahren zur Repraesentation von Terminstanzen und Indexierung der Datenbasis relativ schnell.

Einige Verbesserungsmoeglichkeiten gibt es noch bei der Repraesentation der Terminstanzen. Der gegenwaertige Versuch, ein Structure Sharing-Verfahren zu verwenden, ist eine Moeglichkeit. Aber auch das Non Structure Sharing-Verfahren kann noch verbessert werden, insbesondere im Zusammenhang mit einem compilativen Ansatz.
Die Indexierung der Datenbasis kann, um ein Beispiel zu nennen, noch um Alternativen fuer die Wahl der Indexierungskoordinate erweitert werden.
Die Transparenz koennte durch Verwendung von Upward Failure Continuations bei der Kontrolle wieder gesteigert werden, jedoch ist noch nicht klar, ob ein solches Verfahren ohne allzu grosse Effizienzverluste implementiert werden kann.

Fuer einen Compiler fuer LISPLOG (falls ein solcher doch noch implementiert werden sollte) sollte offene Compilation zur Effizienzsteigerung in Betracht gezogen werden.

Nun zu der Integration der Programmierstile. LISPLOG bietet alle grundlegenden Moeglichkeiten, von einem Teil der Sprache auf den anderen durchzugreifen.
Die Verwendung des PROLOG-Beweisers als Generator von Loesungen fuer LISP, wie sie von M. Dahmen entwickelt wurde [Dahmen 1986b], ist ein weiterer Verbesserungsschritt; die Form der Schnittstelle kann jedoch noch funktionaler gestaltet werden.
Beim Durchgriff von PROLOG auf LISP koennen spezielle evaluierbare Terme in Zielen eingefuehrt werden. Es sollte dabei jedoch vorsichtig vorgegangen werden, um die Semantik von LISPLOG-Programmen nicht zu stark zu komplizieren. SASL ([Noekel 1985], [Rehbold 1985] und [Noekel & Rehbold 1986]), eine 'lazy' funktionale Sprache, die inzwischen als Partner des PROLOG-Teils von LISPLOG vorgesehen ist [Mayer & Richter 1987], wuerde sich hier sehr gut einpassen; die Evaluation braucht hier, von der Unifikation gesteuert, erst zum letztmoeglichen Zeitpunkt zu erfolgen.
Eine weitere Moeglichkeit die Schnittstelle zu erweitern ist die Interpretation von Listen, die als Resultat der Evaluation von

LISP-Ausdruecken zurueckgegeben wurden, als Menge alternativer Loesungen, die beim Backtracking nacheinander verwendet werden koennten. Auch hier waere SASL mit 'lazy evaluation' der geeignetere Partner.
Es stellt sich aber die Frage, ob dies die richtige Verwendung des eigentlich deterministischen Partners LISP (bzw. SASL) ist.

Mit SASL als Partner waere auch die Entscheidung zu ueberdenken, ausschliesslich Cambridge-Polnische Praefix-Notation zu verwenden und nicht zwischen Listen und Termen zu unterscheiden.

Eine Besonderheit von LISPLOG ist die Beschraenkung auf den initialen Cut. Die Transparenz von LISPLOG-Programmen wurde dadurch bedeutend unterstuetzt. Es beeintraechtigt aber auch die Effizienz und extra-logische Ausdrucksstaerke von LISPLOG.
Als Ausgleich bietet sich die Verwendung klarer Kontrollstrukturen, wie LM-Prolog's IF, CASES bzw. PROVE-ONCE, oder die Typisierung von Termen an, wie sie in Horne beispielsweise mit Hilfe von Klauseln oder in LISLOG mit Hilfe von LISP-Praedikaten erfolgt .
Schliesslich koennten auch Prozeduren als deterministisch deklariert werden.

Anerkennung: Mein besonderer Dank gilt Ansgar Bernardi, der vor allem fuer das Kapitel ueber Indexierung ein anregender Diskussionspartner war, sowie Michael Dahmen, fuer Anregungen und Hinweise zur Implementation der Kontrolle und der Repraesentation von Terminstanzen.
Weiterer Dank gilt Knut Hinkelmann, Juergen Herr und Robert Rehbold, und, nicht zuletzt, meinem Betreuer Harold Boley.

## Literatur

[Allen et al. 1983] J. F. Allen, M. Giuliano, A. M. Frisch: The HORNE Reasoning System. University of Rochester, Computer Science Department, Rochester, NY 14627, TR 126, Dec. 1983

[Bailey 1985] D. Bailey: The University of Salford Lisp/Prolog System. Software - Practice and Experience, 15(6), Juni 1985, pp. 595-609

[Bernardi 1986] A. Bernardi: Ein Indexierungskonzept fuer LISPLOG-Datenbasen. Universitaet Kaiserslautern, FB Informantik, SEKI Working Paper SWP 86-10, Dezember 1986

[Bernardi et al. 1986] A. Bernardi, M. Dahmen, M. Meyer: LISPLOG Benutzerhandbuch. Universitaet Kaiserslautern, FB Informantik, SEKI Working Paper SWP 86-11, Dezember 1986

[Boley 1986a] H. Boley: RELFUN: A Relational/Functional Integration with valued Clauses. Universitaet Kaiserslautern, FB Informatik, SEKI Report SR-86-04, Mai 1986

[Boley 1986b] H. Boley: A Bird's-Eye View Of LISPLOG: The LISP/PROLOG Integration with Initial-cut Tools. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-08, Dezember 1986

[Boley & Kammermeier et al. 1985] H. Boley, F. Kammermeier u. die LISPLOG-Gruppe: LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung. Universitaet Kaiserslautern, FB Informatik, MEMO SEKI-85-03, August 1985. Short version in: B. Nebel (Ed.): Papiere zum Workshop Logisches Programmieren und Lisp. TU Berlin, FB Informatik, KIT-REPORT 31, Dec. 1985, pp. 36-53

[Boley & Meyer 1986] H. Boley, M. Meyer: Implementation einer LISP/PROLOG-Vereinheitlichung und ihrer Interaktionsumgebung. In: F. Simon (Ed.): Implementierung von funktionalen und logischen Programmiersprachen, Bericht Nr. 8603, Institut fuer Informatik und Praktische Mathematik, Christian-Albrechts-Universitaet Kiel, Mai 1986.

[Boyer & Moore 1972] R. S. Boyer, J. S. Moore: The Sharing of Structure in Theorem Proving Programs. Machine Intelligence 7, Edinburgh 1972, pp. 101-116

[Bourgault et al. 1985] S. Bourgault, M. Dincbas, J. P. Le Pape: THE LISLOG SYSTEM. Centre National d'Etudes des Telecommunications, Note technique NT/LAA/SLC/186, Mai 1985

[Bruynooghe 1982] M. Bruynooghe: The Memory Management of PROLOG Implementations. In: K. Clark, S. A. Taernlund (Eds.): Logic programming. Academic Press, London, 1982, pp. 83-98

[Campbell & Hardy 1984] J. A. Campbell, S. Hardy: Should PROLOG be List or Record oriented. In: J. A. Campbell (Ed.): Implementations of PROLOG. Ellis Horwood Ltd., 1984

[Carlsson 1981] M. Carlsson: (Re)implementing PROLOG in LISP or YAQ - Yet Another QLOG. UPMAIL Technical Report 5, Uppsala University, Oktober 1981

[Carlsson 1985] M. Carlsson: A Microcoded Unifier for Lisp Machine Prolog. 1985 Symposium on Logic Programming, Juli 1984, Boston, Massachusetts, IEEE Computer Society Press, pp. 162-171

[Chester 1979] D. L. Chester: Using HCPRVR. Internal Report, Department of Computer Science, University of Texas at Austin, August 1979

[Chester 1980] D. Chester: HCPRVR: An Interpreter for Logic Programs. 1st NCAI-80, Stanford University, August 1980

[Clocksin & Mellish 1981/84] W. Clocksin & C. Mellish: Programming in PROLOG. Springer Verlag, Berlin Heidelberg New York, 1981. Second Edition 1984

[Cohen 1986] S. Cohen: The APPLOG Language. In: D. DeGroot & G. Lindstrom [Eds.]: Logic Programming - Functions, Relations, and Equations. Prentice-Hall, Englewood Cliffs, NJ, 1986

[Combuechen 1985] M. Combuechen: Symbolics to announce PROLOG Support for the 3600 Family of LISP Machines. SYMBO 10/84 E, European Symbolics Users Newsletter, 1(4), Januar 1985

[Dahmen 1985] M. Dahmen: Ein Translator von CPROLOG nach LISPLOG. In: [Dahmen, Herr, Hinkelmann, Morgenstern 1985]

[Dahmen 1986a] M. Dahmen: Iterativer LISPLOG Interpreter Implementierung, Dokumentation und Evaluation. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-03, Juni 1986

[Dahmen 1986b] M. Dahmen: LAZY-LISPLOG. Unveroeffentlichtes Manuskript, Universitaet Kaiserslautern, FB Informatik, 1986

[Dahmen, Herr, Hinkelmann, Morgenstern 1985] M. Dahmen, J. Herr, K. Hinkelmann, H. Morgenstern: LISPLOG: Beitraege zur LISP/PROLOG-Vereinheitlichung. Universitaet Kaiserslautern, FB Informatik, MEMO SEKI-85-10, November 1985

[Fogelholm 1984] R. Fogelholm: Exeter Prolog - some Thoughts on Prolog Design by a LISP User. In: J. A. Campbell (Ed.): Implementations of PROLOG. Ellis Horwood Ltd., 1984

[Frisch et al. 1983] A. M. Frisch, J. F. Allen, M. Giuliano: An Overview of the HORNE Logic Programming System. SIGART

Newsletter, No. 84, April 1983, pp. 27-19

[Greussay 1983] P. Greussay: LOVLISP: Une extension de VLISP vers PROLOG. Documentation en ligne. Universite Paris-8 & L.I.T.P., August 1983. Also in: M. Dincbas (Ed.): Programmation en Logique. Actes du Seminaire 1983, Perros-Guirec, Maerz 1983

[Gross 1985] E. Gross: BABYLON-Prolog. In: B. Nebel (Ed.): Papiere zum Workshop Logisches Programmieren und Lisp. TU Berlin, FB Informatik, KIT-REPORT 31, Dec. 1985, pp. 30-35

[Herr 1985] J. Herr: Breitensuche und Klauselcompilation fuer LISPLOG. In: [Dahmen, Herr, Hinkelmann, Morgenstern 1985]

[Herr 1986] J. Herr: Ansatz fuer einen LISPLOG Compiler mit LISP als Zielsprache. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-06, Dezember 1986

[Hinkelmann 1986] K. Hinkelmann: Uebersetzung von LISPLOG-Programmen nach CPROLOG. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-05, September 1986

[Hinkelmann & Morgenstern 1985] K. Hinkelmann, H. Morgenstern: Ein Verfahren zur Transformation von LISP-Funktionen in PROLOG-Relationen. In: [Dahmen, Herr, Hinkelmann, Morgenstern 1985]

[Kahn 1982] K. M. Kahn: Unique Features of Lisp Machine Prolog. In: Procedures of the Workshop on PROLOG Programming Environments, Linkoeping, March 1982

[Kahn 1983] K. M. Kahn: Unique Features of Lisp Machine Prolog. UPMAIL Technical Report No. 15, Uppsala University, 14. Februar 1983

[Kahn 1983/84] K. M. Kahn: Pure PROLOG in Pure LISP. Logic Programming Newsletter 5, Winter 83/84, pp.3-4

[Kahn 1986] K. M. Kahn: UNIFORM - A Language based upon Unification which unifies (much of) LISP, PROLOG and ACT 1. (Revised version) In: Doug DeGroot & Gary Lindstrom (Eds.): Logic Programming - Functions, Relations and Equations. Prentice-Hall, Englewood Cliffs, NJ, 1986, pp. 411-438

[Kahn & Carlsson 1983] K. M. Kahn, M. Carlsson: LM-Prolog User Manual. UPMAIL, Department of Computing Science, Uppsala University, October 1983

[Kahn & Carlsson 1984] K. M. Kahn, M. Carlsson: How to implement Prolog on a LISP Machine. In: J. A. Campbell (Ed.): Implementations of PROLOG. Ellis Horwood Ltd., 1984

[Kammermeier 1985] F. Kammermeier: Dokumentation der Prolog-Implementation in Franz-Lisp. Universitaet Kaiserslautern, FB

Informatik, April 1985

[Kluzniak & Szpakowicz 1985] F.Kluzniak, S. Szpakowicz, J. S. Bien (Contr.): Prolog for Programmers. Academic Press, London, 1985

[Knoepfler & Hotop 1985] S. Knoepfler, S. Hotop: ConProlog - Eine Prolog-Implementation in Interlisp-D. In: B. Nebel (Ed.): Papiere zum Workshop Logisches Programmieren und Lisp. TU Berlin, FB Informatik, KIT-REPORT 31, Dec. 1985, pp. 54-59

[Komorowski 1982] H. Komorowski: QLOG - The Programming Environment for PROLOG in LISP. In: K. Clark, S. A. Taernlund (Eds.): Logic programming. Academic Press, London, 1982, pp. 315-322

[Lessel 1986] M. Lessel: micro-UNIXPERT Ein wissensbasiertes System zur Behandlung von Problemen bei UNIX-Druckauftraegen. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-04, Mai 1986

[Mayer & Richter 1987] O. Mayer, M. M. Richter: Funktional/logische Expertensystemtools. Allgemeine Angaben zum Teilprojekt FLEX, SFB 314, Universitaet Kaiserslautern, FB Informatik, Januar 1987

[Mellish 1982] C. S. Mellish: An Alternative to Structure Sharing in the Implementation of a PROLOG Interpreter. In: K. Clark, S. A. Taernlund (Eds.): Logic programming. Academic Press, London, 1982, pp. 99-106

[Meyer 1987] M. Meyer: Entwurf und Implementierung einer Interaktionsumgebung fuer LISPLOG. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-87-02, Januar 1987

[Nilsson 1984] M. Nilsson: The World's Shortest Prolog Interpreter? In: J. A. Campbell (Ed.): Implementations of PROLOG. Ellis Horwood Ltd., 1984

[Noekel 1985] K. Noekel: SASL: Implementierung eines Reduktionsalgorithmus und Steuerung und Organisation eines Beweissystems fuer eine Logik. Diplomarbeit, RWTH Aachen, Dezember 1985

[Noekel & Rehbold 1986] K. Noekel, R. Rehbold: SASL: Implementierung einer rein funktionalen Sprache mit Lazy Evaluation. Universitaet Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-07, November 1986

[Okuno et al. 1984] H. G. Okuno, I. Takeuchi, N. Osato, Y. Hibino, K. Watanabe: TAO: A Fast Interpreter-Centered System on Lisp Machine ELIS. Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, August 1984,

pp. 140-149

[Read & Dyer 1985] W. Read, M. G. Dyer: TLOG - Yet another Logic System in Lisp? Tech. Rep. UCLA-AI-85-1, 1985

[Rehbold 1985] R. Rehbold: SASL: Implementierung eines Abstraktionsalgorithmus und Beweisalgorithmen fuer eine Logik. Diplomarbeit, RWTH Aachen, Dezember 1985

[Robinson & Sibert 1981] J. Robinson, E. Sibert: The LOGLISP User's Manual. School of Computer and Information Science, Syracuse University, December 1981

[Robinson & Sibert 1982a] J. Robinson, E. Sibert: LOGLISP: Motivation, Design and Implementation. In: K. Clark, S. A. Taernlund (Eds.): Logic Programming. Academic Press, London, 1982, pp. 299-313

[Robinson & Sibert 1982b] J. Robinson, E. Sibert: LOGLISP: an alternative to PROLOG. Machine Intelligence 10, Chichester, 1982, pp. 399-419

[SALFORD 1984] The University of Salford LISP/PROLOG Reference Manual. University of Salford, Second Edition, March 1984

[Sansonnet 1986] J. P. Sansonnet: The Machine for Artificial Intelligence Applications: MAIA. Proc. 7th ECAI-86, Volume I, Brighton, Juli 1986

[Sato & Sakurai 1983] M. Sato, T. Sakurai: Qute: A Prolog/Lisp Type Language for Logic Programming. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, pp. 507-513

[Sato & Sakurai 1984] M. Sato, T. Sakurai: QUTE: A Functional Language based on Unification. Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT 1984, pp. 157-165

[Schrag 1984] Robert C. Schrag: Compilation and Environment Optimisations for LOGLISP. Rome Air Developement Center, In-House Report RADC-TM-84-14, July 1984

[SYMBOLICS 1985] User's Guide to Symbolics Prolog. Symbolics Inc., Cambridge, Massachusetts, June 1985

[Takeuchi et al. 1983] I. Takeuchi, H. Okuno, N. Ohsato: TAO - A harmonic mean of Lisp, Prolog and Smalltalk. SIGPLAN Notices, V18 #7, Juli 1983, pp. 65-74

[Wallace 1983] R. S. Wallace: An Easy Implementation of PiL. SIGART Newsletter, No. 85, July 1983, pp. 29-32

## Anhang A: Wissenswertes ueber LISPLOG

Post-Anschrift:

LISPLOG-Projekt, SFB 314, Bau 14
Fachbereich Informatik
Universitaet Kaiserslautern
Postfach 3049

D-6750 Kaiserslautern
W.-Germany

E-MAIL-Adresse:

uucp: unido!uklirb!lisplog

- oder -

lisplog@uklirb.UUCP

Telefon:

LISPLOG-Buero : (0631) 205-2805
Sekretariat (vormittags) : -2802
(nachmittags): -2612

Derzeitige Zusammensetzung des Projekts

Leitung
Harold Boley

Effizienz
Iterativer Interpreter
Michael Dahmen
Klausel-Indexierung
Ansgar Bernardi

Konzepte
Module
Michael Dahmen
Literaturvergleich
Franz Kammermeier

Interaktion
Break/Trace/Cut-Anzeiger
Manfred Meyer

Anwendung
micro-UNIXPERT
Michael Lessel

Translator
CPROLOG
Knut Hinkelmann

## Versand von Software

Das LISPLOG-System ist als LISP-Quellprogramm fuer UNIX-Rechner mit FRANZ LISP oder COMMON LISP erhaeltlich.

Der Versand erfolgt ueber E-MAIL im UUCP-Netz oder auf Band im tar- oder Standard-ANSI-Format mit 800 oder 1600 bpi (bitte das Format angeben!!!) und ist zur Zeit gebuehrenfrei.

Die Dokumentation wird per Post zugesandt. Erhaeltlich ist momentan [Bernardi 1986], [Bernardi et al. 1987], [Boley 1986b], [Boley & Kammermeier et al. 1985], [Dahmen, Herr, Hinkelmann, Morgenstern 1985], [Dahmen 1986a], [Herr 1986], [Hinkelmann 1986], [Lessel 1986] und [Meyer 1987].

## Copyright Vermerk