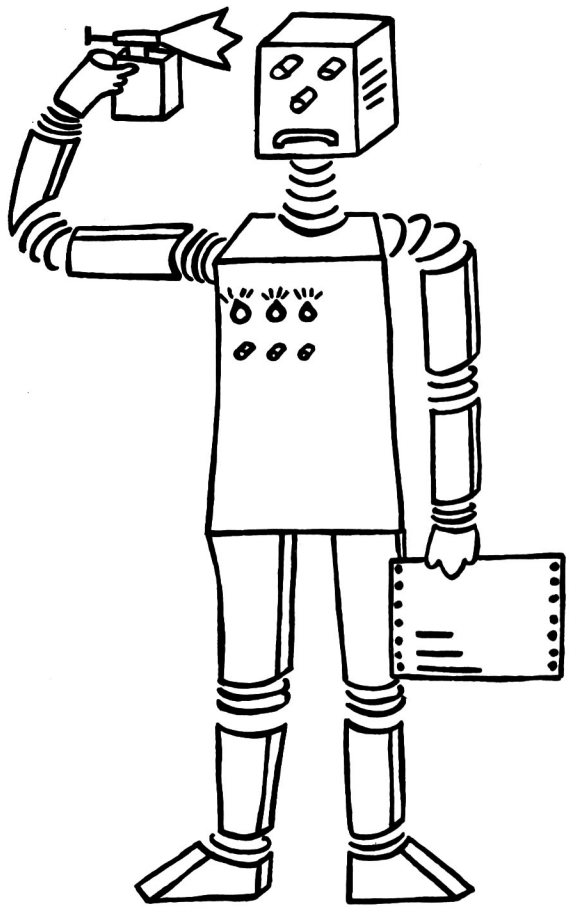
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI-Working Paper



# SASLOG: Eine funktional-logische Sprachintegration mit Lazy Evaluation und semantischer Unifikation

Knut Hinkelmann

SEKI Working Paper SWP-88-06

# SASLOG:

## Eine funktional-logische Sprachintegration

## mit Lazy Evaluation und semantischer Unifikation

Diplomarbeit

von

Knut Hinkelmann

Fachbereich Informatik
Universität Kaiserslautern

## Abstract

SASLOG is a combined functional/logic programming language which contains SASL, a fully-lazy, higher-order functional language and the logic language Prolog. The integration is symmetric allowing functional terms to appear in the logic part and Prolog goals in the functional part.

Exploiting the natural correspondence between backtracking and lazy streams yields an elegant solution to the problem of transferring alternative variable bindings to the calling functional part of the program.

The evaluation of functional expressions in the logic part is driven by the extended unification algorithm which takes into account the semantics of function symbols.

The rewriting approach to function evaluation is replaced by combinator graph reduction, thereby regaining computational efficiency and the structure sharing properties. The integration fits well to combinator graph reduction. So the instantiation of logic variables supports structure sharing. On the other hand we provide a solution to the reduction of functional expressions containing logic variables in different binding environments.

# Gliederung

# 1 Einleitung

Es gibt viele Programmiersprachen und Programmierstile, die je nach Art der Anwendung für die Modellierung mehr oder weniger angemessen sind. Die ältesten und am weitesten verbreiteten sind die imperativen Sprachen. Sie orientieren sich am von Neumann- Rechnermodell und beruhen auf der Änderung von Variablenwerten. Imperative Sprachen sind sequentiell. Die parallele Ausführung verschiedener Operationen scheitert am sogenannten von Neumann-Flaschenhals ([Backus 1978]), d.h. dem sich in der Zuweisungsoperation widerspiegelnden wortweisen Informationstransport zwischen Rechen- und Speichereinheit, der wegen verbreiteter Datenabhängigkeiten nur für spezielle Operationen parallelisiert werden kann. Die Semantik dieser konventionellen Programmiersprachen wird beschrieben durch Zustandsübergänge. Wegen der uneingeschränkten Änderungsmöglichkeiten von Variablenwerten (Seiteneffekte) ist sie sehr komplex. Von Neumann-Programme sind operational, d.h. man muß sie geistig ausführen, um sie zu verstehen.

Deklarative Programmiersprachen dagegen beschreiben gewünschte Sachverhalte, ohne explizite Berechnungsvorschriften anzugegeben. Sie zeichnen sich gegenüber imperativen Programmiersprachen dadurch aus, daß sie anhand eines mathematischen Modells unabhängig von einer konkreten Maschine entwickelt wurden. Sie erhalten dadurch Vorteile wie z.B. größere Ausdruckskraft, Erleichterung paralleler Ausführungen und die Möglichkeit algebraischer Umformungen, was den Beweis der Korrektheit von Programmen unterstützt. Die beiden Hauptrichtungen deklarativer Programmiersprachen sind

- funktionale Sprachen, die auf dem λ-Kalkül und den Rekursionsgleichungen (Ersetzen von Gleichem durch Gleiches) beruhen, und

- logische (relationale) Sprachen, die auf der prozeduralen Interpretation der Prädikatenlogik erster Stufe aufbauen.

Um die Vorteile der logischen wie auch der funktionalen Programmierung nutzen zu können - indem man sich lokal entscheiden kann, welchen Teil eines Problems man in welchem Programmierstil repräsentiert - gibt es viele Versuche, beide Techniken in einem System anzubieten. Die im Rahmen dieser Arbeit vorgestellte Sprache SASLOG erlaubt sowohl funktionale und relationale Programmierung als auch eine Kombination beider Programmierstile. Die Verbindung beider Stile beruht dabei auf der Verzahnung der bei modernen funktionalen Sprachen häufig zu findenden Lazy Evaluation-Berechnungsstrategie und der Suche mit Backtracking. Die verzahnte Berechnung wird koordiniert durch den semantischen Unifikationsalgorithmus in der logischen und die Mengenabstraktionsausdrücke in der funktionalen Komponente.

Ein allgemeiner Vergleich funktionaler und logischer Programmierung, bei dem nicht auf Eigenschaften spezieller Sprachen eingegangen wird, findet sich im zweiten Kapitel. In Kapitel 3 werden verschiedene Integrationsansätze vorgestellt. Sie reichen von der losen Kopplung bestehender Sprachen bis zur Entwicklung einer Programmiersprache, die sowohl die funktionale als auch die relationale Programmierung in einem modelltheoretischen Rahmen unterstützt. Die Sprache SASLOG wird in die Reihe der unterschiedlichen Vereinigungsmöglichkeiten eingeordnet. Eine Beschreibung von SASLOG findet sich in Kapitel 5. Daran anschließend wird die Semantik der Integrationskonzepte erläutert. Die restlichen Kapitel gehen näher auf die konkrete Realisierung von SASLOG ein.

Einige der in dieser Arbeit behandelten Aspekte von SASLOG sind in kürzerer Fassung auch bei [Hinkelmann, Nökel, Rehbold 1988] beschrieben.

# 2 Vergleich funktionaler und logischer Programmierung

Relationale und funktionale Programmierung haben gemeinsame Vorzüge, die von ihrem deklarativen Charakter herrühren. Andererseits unterscheiden sie sich aber in der Art der Problembeschreibung und in der Programmausführung. [Hölldobler 1984] stellt die Grundideen der funktionalen und logischen Programmierung vor. Die folgenden Abschnitte sollen die wichtigsten Unterschiede beider Stile verdeutlichen. Die Beispiele für logische Programme in diesem Kapitel werden in PROLOG-Syntax notiert, wobei die Bekanntheit von PROLOG vorausgesetzt wird (für eine kurze Einführung siehe Abschnitt 4.1). Für die Darstellung funktionaler Ausdrücke wird eine Syntax verwendet, die derjenigen von PROLOG ähnlich ist.

## Berechnung

Ein logisches Programm besteht aus einer Menge von Hornklauseln, wie das folgende zum Zusammensetzen von Listen:

```
append([ ],L,L).
append([A|L1], L2, [A|L3])  :-  append(L1,L2,L3).
```

Die Ausführung eines Programms wird angestoßen durch die Eingabe eines Ziel-Ausdrucks mit Variablen und/oder Werten. Das System berechnet meist durch ***Resolution*** ([Robinson 1965]) die Variablenwerte, die den Ausdruck wahr machen.

Ein funktionales Programm besteht aus einer Menge von Funktionsgleichungen:

```
apnd([ ],L)          =  L
apnd([A|L1], L2)     =  [A | apnd(L1,L2)]
```

Der Wert eines Ausdrucks ist bedingt durch die Werte seiner Unterausdrücke. Die Berechnung besteht in der ***Reduktion*** eines Ausdrucks zu seiner Normalform, indem die Gleichungen als gerichtete Ersetzungsregeln angewandt werden.

## Gerichtetheit

Bei der Reduktion von Ausdrücken in funktionaler Programmierung erlaubt das Pattern Matching des Ausdrucks mit dem Aufrufmuster der Funktion nur einen Informationsfluß von dem Ausdruck zur Funktionsdefinition (***directionality***). Bei der Resolution, dem Beweisverfahren der logischen Programmierung, wird das Ziel

mit der Konklusion einer Klausel unifiziert, was einen Informationsfluß in beiden Richtungen erlaubt (***undirectionality***, vgl. [Reddy 1986]). Für eine logische Variable ist also nicht festgelegt, ob sie als Eingabe- oder Ausgabevariable anzuwenden ist. Diese Ungerichtetheit erlaubt die Definition eleganter und mächtiger Programme, wie sie in funktionalen Sprachen nicht möglich sind. Relationale Programmiersprachen sind in diesem Sinn ausdrucksstärker.

Beispiel: Die oben definierte Relation append kann z.B. zum Zusammensetzen von Listen,

```
?- append([1,2],[3,4],L). ,
```

zum Aufspalten,

```
?- append(L1,L2,[1,2,3,4]). ,
```

zum Test, ob zwei Listen zusammengesetzt eine dritte Liste ergeben,

```
?- append([1],[2,3,4],[1,2,3,4]). ,
```

oder in einer anderen Kombination freier und belegter Parameterstellen verwendet werden. In einem funktionalen Programm sind für jeden dieser Fälle eigene Funktionsdefinitionen notwendig.

## Determinismus - Nichtdeterminismus

Funktionale Sprachen sind ***deterministisch***. Für jedes Tupel von Eingabewerten gibt es genau einen Resultatswert, so daß bei der Berechnung eines funktionalen Ausdrucks keine Suche notwendig ist. Logische Programme dagegen sind ***nichtdeterministisch***, eine Anfrage kann zu mehreren Lösungen führen. Die Ausführung eines logischen Programms erfordert einen Suchprozeß. Zwar existiert mit der Breitensuche eine vollständige Suchstrategie, jedoch wird in PROLOG-Systemen meist die Tiefensuche implementiert, so daß der Programmierer das Ausführungsverhalten beachten muß. Das widerspricht aber dem Prinzip der deklarativen Programmierung.

## Teilinstantiierte Ausdrücke

In der relationalen Programmierung können Resultate, im Gegensatz zur funktionalen Programmierung, nur teilweise festgelegt sein, indem sie logische Variablen enthalten, d.h. sie sind ***non-ground***. Solche Datenstrukturen können als Werte übergeben werden.

## Komposition

Während funktionale Sprachen die Möglichkeit der ***Komposition*** von Funktionsausdrücken bieten, müssen in logischen Sprachen zur Übergabe von Werten an andere Relationen neue Variablen eingeführt werden.

Beispiel: funktional: `member(X,append(L1,L2))`
relational: `append(L1,L2,L), member(X,L)`

## Operationen höherer Ordnung

Viele funktionale Programmiersprachen erlauben ***Funktionen höherer Ordnung***, d.h. Funktionen können als Parameter übergeben und als Werte zurückgeliefert werden. Im Gegensatz dazu sind logische Programmiersprachen meist Sprachen erster Ordnung.

## Spracherweiterungen

Funktionale Sprachen können mit mächtigen Konzepten wie ***Lazy Evaluation***, Bearbeitung ***unendlicher Datenstrukturen*** oder ***polymorphen Typen*** ausgerüstet werden, die auf die logische Programmierung nicht so einfach zu übertragen sind.

Ein schönes Beispiel für die elegante Programmierung, die auf der Verwendung von freien Variablen und der Ausnutzung der Ungerichtetheit logischer Programmierung beruht, findet man bei [Reddy 1986]. Dargestellt ist ein Übersetzungsproblem, das in Compilern und Bindern auftritt. Gegeben sind zwei Arten von Objekten: `def(a)` und `use(a)` für ein Atom `a`. Gesucht ist eine Übersetzung von `def(a)` nach `asgn(a,n)`, wobei `n` eine ganzzahlige Adresse ist, und `use(a)` wird zu `use(n)`, wobei `n` die gleiche Zahl ist wie in `asgn(a,n)`. Die Atome werden bei 1 beginnend in der Reihenfolge ihrer Definition numeriert. Die Liste `[def(a), use(a), use(b), def(c), def(b)]` wird übersetzt zu `[asgn(a,1), use(1), use(3), asgn(c,2), asgn(b,3)]`. Folgendes PROLOG-Programm realisiert diese Übersetzung:

```
translate(Inlist,Outlist) :- map(Inlist,Outlist,Table,1).

map([ ],[ ], _, _).
map([def(A)|Inlist], [asgn(A,N)|Outlist], Table, N) :-
                    member(asgn(A,N),Table),
                    map(Inlist,Outlist,Table,N+1).
```

```
map([use(A)|Inlist], [use(Addr)|Outlist], Table, N) :-
                    member(asgn(A,Addr),Table),
                    map(Inlist,Outlist,Table,N).

member(A,[A|X])  :- !.
member(A,[B|X])  :- member(A,X).
```

In einer funktionalen Sprache sind die Vorwärtsreferenzen - hier use(a) für Atome, die noch nicht definiert sind - nur sehr schwierig zu handhaben. Die in solchen Fällen gewöhnlich benutzte Technik (Backpatching) ist ohne Seiteneffekte nur sehr aufwendig und unübersichtlich zu programmieren. In einem logischen Programm werden Vorwärtsreferenzen als partielle Datenstrukturen mit ungebundenen Adreßvariablen aufgebaut, die gebunden werden, wenn die Definition auftritt. Das Backpatching wird automatisch durch die Resolution durchgeführt.

Die Ungerichtetheit purer logischer Programme erlaubt keine Kontrollinformation. Dies kann zu ineffizienten und sogar zu nicht-terminierenden Programmen führen. [Reddy 1986] bringt dazu ein einfaches Beispiel:

Die Relation reverse zur Umkehrung von Listen ist durch folgende Klauseln gegeben:

```
reverse([ ],[ ]).
reverse([A|X], Y) :-  reverse(X, Z),
                      append(Z, [A], Y).
```

Die Relation append sei wie oben definiert. Die Anfrage ?- reverse([1,2],L) führt nach einem Schritt zu folgender Liste von Goals:

```
reverse([2], Z), append(Z, [1], L).
```

Wählt man bei einer Liste von Goals immer zuerst das linkeste aus, hier reverse([2],Z), kommt man zur Lösung L = [2,1] und das Programm terminiert. Wählt man aber immer zuerst das rechte Goal, so läuft das Programm nach dem Finden der richtigen Substitution bei einer weiteren Anfrage in eine Schleife. Bei einer Anfrage ?-reverse(L,[2,1]) ist es genau umgekehrt.

Funktionale Programme haben durch ihre Gerichtetheit die Kontrollinformation, um solche Terminierungsprobleme zu vermeiden, benötigen aber für jede der beiden Anfragen eine eigene Funktion. Die in einem funktionalen Programm enthaltene Kontrollinformation kann durch effiziente Implementierungen genutzt werden. Eine funktionale Sprache ist daher für die Definition von Algorithmen vorzuziehen.

Ein Beispiel für die Anwendung von Funktionen höherer Ordnung liefert die Filterfunktion, die für eine Liste L die Liste der Elemente liefert, für die die Funktion p den Wert TRUE hat:

```
filter(p,[a|x])   = IF p(a) THEN [a | filter(p,x)]
                             ELSE  filter(p,x)
filter(p,[ ])     = [ ]
```

Funktionen höherer Ordnung erlauben extrem komprimierte Programmierung. Ein kleines Beispiel aus [Turner 1981] soll dies verdeutlichen:

Die Funktion sum soll alle Elemente einer Liste summieren. Diese Art der Bearbeitung von Listenelementen mit einem binären Operator kann man durch eine Funktion fold zusammenfassen. Die Funktion hat drei Argumente: Einen binären Operator, das neutrale Element dieser Operation und eine Liste (die Syntax entspricht der Syntax von SASL, siehe Abschnitt 4.2):

```
fold op s [ ]    = s
fold op s (a:x)  = op a (fold op s x)
```

Die Funktion sum kann in einer Zeile definiert werden:

```
sum = fold plus 0
```

Analoge Funktionen können nun ebenfalls ohne Aufwand definiert werden, wie die Funktion product zur Multiplikation aller Listenelemente:

```
product = fold times 1
```

# 3 Funktional-logische Programmiersprachen

## 3.1 Verbindung funktionaler und logischer Programmierung

Viele Programmsysteme können auf natürliche Weise in zwei Teile gegliedert werden: Einen Teil, der Informationen durch Inferenz über einer Wissensbasis berechnet, und eine Menge von Algorithmen, die die so gewonnenen Informationen verarbeiten. Es gibt nun viele Versuche, die Vorteile der funktionalen und der logischen Programmierung in einem System zur Verfügung zu stellen. Mehrere Autoren haben die verschiedenen Ansätze unter unterschiedlichen Aspekten verglichen (u.a. [Bellia, Levi 1986], [Reddy 1986], [Kammermeier 1986]).

### Verbindung durch Schnittstellendefinition

Viele Systeme, die die logische und funktionale Programmierung in einer Umgebung anbieten, basieren auf einer losen Kopplung zweier bestehender Sprachen. Dabei wird die logische Komponente in einer funktionalen Sprache (meist LISP) implementiert und eine Schnittstelle zur Implementationssprache definiert. Zu dieser Gruppe gehören z.B. LISPLOG ([Boley 1986]) und LOGLISP ([Robinson, Sibert 1982a], [Robinson, Sibert 1982b]). Die einzelnen Systeme unterscheiden sich durch die logische Komponente und die Defintion der Schnittstelle. Während die meisten als logischen Teil PROLOG verwenden, erlaubt LISPLOG in der logischen Komponente nur den initialen Cut-Operator (der Cut als erste Prämisse); LOGLISP verwendet eine möglichst pure logische Sprache mit Breitensuche. Eine ausführliche Diskussion dieser LISP/PROLOG-Vereinheitlichungen findet man bei [Kammermeier 1986].

### Funktionale Sprachen mit Narrowing

[Reddy 1986] beschreibt drei Sprachklassen, die die stärkere Ausdrucksfähigkeit der logischen Sprachen in funktionale Sprachen einbinden. Die erste dieser Sprachklassen, die Klasse der N-Sprachen, erweitert die operationale Semantik der funktionalen Sprachen durch das Narrowing-Verfahren. Dabei werden Ausdrücke, die noch Variablen enthalten, gegen linke Seiten von Funktionsgleichungen unifiziert (im Gegensatz zum Matching bei der Reduktion) und die für die Eingabevariablen erzeugten Bindungen werden als Narrowing-Substitution gesammelt.

Zu dieser Klasse gehört die Sprache FRESH ([Smolka 1985]). FRESH wurde schrittweise aus FM, einer funktionalen Sprache höherer Ordnung,entwickelt. Das Matching des zu reduzierenden Ausdrucks gegen das Funktionspattern ist bei FM so gegenüber anderen funktionalen Sprachen erweitert, daß das Pattern Variablen

mehrfach enthalten kann. Da das Matching scheitern kann, wurde das "failure" als Konzept in die Sprache aufgenommen, so daß eine Reduktion entweder mit einem Resultat erfolgreich ist oder scheitert. Durch Verallgemeinerung von Matching zu Unifikation erhält man die Sprache FU. Nun können auch die zu reduzierenden Terme Variablen enthalten, so daß Terme durch spätere Unifikation spezialisiert werden können (Narrowing). Durch Einführung vielfacher Resultate geht FRESH aus FU hervor. Vielfache Resultate sind Disjunktionen von Ergebnissen, wobei FRESH - wie PROLOG - eine Tiefensuch-Strategie mit Backtracking realisiert.

Eine weitere Sprache mit Narrowing ist Eqlog ([Goguen, Meseguer 1984/86]), die aber eine allgemeinere Gleichungssprache statt einer funktionalen Sprache zu Grunde legt (s.u.).

## Funktionale Sprachen und Mengenabstraktion

N-Sprachen erlauben nicht die Erzeugung neuer Variablen als Ausgabe. Dies erreicht man durch die zusätzliche Einführung von Mengenabstraktionen mit freien Variablen, die Reddy ([Reddy 1986]) als freie Mengen bezeichnet. Eine funktionale Programmiersprache mit freien Mengen gehört zur Klasse F.

Ein Beispiel für die Erweiterung einer funktionalen Programmiersprache durch Mengenabstraktion und Unifikation zur Darstellung von Relationen ist SUPERLOGLISP ([Robinson 1983]). Dessen funktionale Berechnung beruht aber nicht auf Narrowing. Definitionen von Relationen sind Vereinigungen mengenwertiger Ausdrücke. Betrachte als Beispiel die bekannte Relation append:

```
append(a,b,c) = {a,b,c | (a = [ ]  and  b = x  and  c = x)
                         or
                         (a = [x | y]   and
                          b = z         and
                          c = [x | w]   and
                          append(y,z,w))}
```

Der Nichtdeterminismus wird erreicht durch die Vereinigung von Mengen (hier dargestellt durch das or), Unifikation wird explizit dargestellt durch Gleichungen zwischen Variablen und Termen. Ein Goal ist hier ebenfalls ein Mengenausdruck, der zu seiner Normalform reduziert wird:

```
{ [x,y] | append(x,y,[1,2,3])}
```

ist die Menge aller Paare x und y, die zusammengesetzt die Liste [1,2,3] ergeben.

[Darlington, Field, Pull 1986] stellen ebenfalls eine um Mengenabstraktion zur Relationendarstellung erweiterte funktionale Sprache vor. Im Unterschied zu SUPERLOGLISP, wo die Qualifier nur Funktionsaufrufe und Gleichungen zwischen Variablen und Termen sind, können die Gleichungen dieser Sprache weitaus komplexer sein. Während aber für jede mögliche Kombination von belegten und freien Parameterstellen beim Aufruf einer Relation eine eigene Funktion existieren muß, wird dies in SUPERLOGLISP einfach durch Auswertung von Mengenausdrücken erreicht ([Bellia, Levi 1986]).

## Übersetzung funktionaler Programme in logische Programmme

Die beiden Sprachklassen N und F erreichen die Ungerichtetheit, ohne die Vorteile der Gerichtetheit zu verlieren. Dagegen eliminieren die Sprachen der Klasse R die Gerichtetheit (und verlieren dadurch Kontrollinformation), indem sie funktionale Programme in logische Programme übersetzen mit Resolution als operationaler Semantik.

[Reddy 1986] selbst ordnet die Sprache LEAF ([Barbuti, Bellia, Levi 1986]) in diese Klasse ein. Trifft dies für die Abarbeitung mit Resolution noch zu, so gilt es allerdings nicht in Bezug auf die Gerichtetheit. Die Funktionen in LEAF sind definiert als Hornklauseln mit mode-Deklarationen, d.h. die Terme sind so als Eingabe- oder Ausgabeterme deklariert, daß Funktionen deterministisch sind (allgemeine logische Variablen sind also nicht erlaubt). Jede Klausel eines LEAF-Programms wird in eine kanonische Form übersetzt, indem in Terme eingebettete Funktionsaufrufe durch neue Variablen ersetzt und diese mit dem Funktionsaufruf identifiziert werden ([Bellia, Levi 1986]). Für die funktionalen Werte gibt es eine producer-consumer-Interpretation. Die Auswertung durch die prozedurale Komponente (LCA, [Bellia u.a. 1982]) wird anforderungsgetrieben (demand driven) gesteuert. Dadurch erreicht man Lazy Evaluation und durch die gerichteten Informationskanäle bleiben die Vorteile der Gerichtetheit erhalten.

## Logik mit Gleichheit

[Kornfeld 1983] zeigt anhand von Beispielen, wie durch die Erweiterung von PROLOG um Gleichheit die Ausdrucksfähigkeit erhöht und funktionale Darstellung ermöglicht wird. Allerdings ist seine Implementierung noch sehr eingeschränkt, z.B. wird die Reflexivität der Gleichheit nicht beachtet.

Eqlog ([Goguen, Meseguer 1984/86]) integriert reine logische Programmierung und funktionale Programmierung erster Ordnung zu einer mehrsortigen Hornklausel-Logik erster Ordnung mit Gleichheit. Eine operationale Semantik ist gegeben durch Narrowing. Eine vollständige, modelltheoretische Semantik wird ebenfalls angegeben. Die durch die Gleichungen definierten Rewrite-Regeln müssen kanonisch sein, was keine unendlichen Datenstrukturen erlaubt. Neben der Beantwortung von logischen Anfragen (durch Resolution und Backtracking) und der Berechnung von Funktionsausdrücken über Grundtermen (durch Rewriting) ist es auch möglich, Gleichungen mit logischen Variablen zu lösen (durch Narrowing). Um den Gegensatz zwischen Allgemeinheit (mit Vollständigkeit) und Effizienz zu lösen, versucht Eqlog, ineffiziente Berechnungen zu vermeiden, indem möglichst viel funktional berechnet und Relationen hauptsächlich für die Kontrolle verwendet werden. Negation, Operationen höherer Ordnung und außerlogische Merkmale zur Steuerung der Kontrolle (wie der Cut in PROLOG) sind nicht enthalten.

Bei [Fribourg 1984] ist die Gleichheit das einzige Prädikat. Ein Programm ist eine Folge von Hornklauseln. Die Konklusion wird als gerichtete Ersetzungsregel interpretiert, die Prämissen sind die Bedingungen. Die Berechnung bezeichnet Fribourg als "clausal superposition".

## Erweiterte Unifikation

Für die Eingliederung einer funktionalen Sprache in eine bestehende logische Sprache unterscheidet man zwei Möglichkeiten:

- Die Integration auf Prädikatenebene wird ermöglicht durch Einführung eines Prädikats eq(X,Y), das den Term Y berechnet und den Wert mit X unifiziert (vgl. is in Edinburgh PROLOG). [van Emden, Yukawa 1986] bezeichnen dies als schwache Verschmelzung ("weak amalgamation").

- Bei der starken Verschmelzung ("strong amalgamation") der Integration auf Termebene sind funktionale Ausdrücke als Terme in Prädikaten erlaubt.

Integration auf Termebene impliziert die Integration auf Prädikatenebene, denn das Prädikat eq entspricht dann gerade der Unifikation. Termebenenintegration erfordert eine Modifikation des Unifikationsalgorithmus, der die Semantik von Funktionssymbolen beachtet. [Dincbas, van Hentenryck 1987] stellen verschiedene erweiterte Unifikationsalgorithmen vor, die sich in der Art der Berechnung funktionaler Ausdrücke unterscheiden (Evaluation, Delayed Evaluation, Derivation, Lazy Derivation, Surderivation, Lazy Surderivation). Die Reduktion funktionaler Ausdrücke wird in den Unifikationsalgorithmus integriert.

FUNLOG ([Subrahmanyam, You 1984]) kombiniert funktionale und logische Programmierung durch das Konzept der semantischen Unifikation mit Lazy Derivation. Die Reduktion ist mustergesteuert (pattern driven) und wird nur dann ausgeführt, wenn die syntaktische Unifikation zweier Terme nicht möglich ist. Dabei werden, gesteuert durch den Unifikationsalgorithmus, jeweils nur die Reduktionsschritte ausgeführt, die notwendig sind, um die beiden Terme semantisch zu unifizieren (reduction-by-need). Dies erlaubt die Berechnung mit konzeptuell unendlichen Datenstrukturen. Terme mit freien Variablen sind nicht reduzierbar.

[Hölldobler, Fuhrbach, Laußermair 1985] vereinigen funktionale und logische Programmierung ebenfalls über einen erweiterten Unifikationsalgorithmus, der auf der Reduktion funktionaler Ausdrücke, Anwendung inverser Funktionen und der Erzeugung äquivalenter logischer Programme für Funktionsdefinitionen beruht. Sie definieren die Semantik ihrer Sprache FHCL vollständig im logischen Rahmen. Es bleiben allerdings noch einige offene Fragen: Wann soll ein Ausdruck reduziert werden? Wie erhält man zu einer Funktion ein äquivalentes logisches Programm? ([Hölldobler, Fuhrbach, Laußermair 1985]).

## Symmetrische Integration zu einer Termlogik

RF-Maple ([Voda, Yu 1984]) ist eine Kombination der getrennt entwickelten Sprachen R-Maple und F-Maple. R-Maple ist eine relationale Sprache, die parallele und sequentielle Ausführung erlaubt, explizite Quantoren und Negation enthält. Die Berechnung in der funktionalen Sprache F-Maple beruht auf Lazy Evaluation. RF-Maple hat beide Teilsprachen als echte Untermenge. Die Vereinigung hat als Zielvorstellung eine Termlogik, bei der Relationen einfach Funktionen mit booleschen Werten sind. Die Ausdruckskraft der logischen Programmierung wird erreicht durch die sogenannten Generatoren, das sind boolesche Funktionen mit Ausgabeparametern.

## Logische Sprachen und Lazy Evaluation

[Hansson, Haridi, Tärnlund 1982] stellen eine auf natürlicher Deduktion beruhende Sprache vor. Sie unterstützt funktionale Notation und hat eine auf Lazy Evaluation beruhende Auswertungsregel.

FPL ([Bellia, Degano, Levi 1982]) ist eine um bedingte Gleichungen und mehrwertige Funktionen (beschrieben durch relationale Syntax) erweiterte funktionale Sprache mit einer call-by-name-Semantik, die die Definition nicht-strikter Funktionen und Prozeduren mit unendlichen Datenstrukturen (Streams) als Ein- bzw. Ausgabe gestattet.

[Narain 1986] präsentiert eine Technik zur Definition von Reduktionsregeln in PROLOG, die den Wert eines Ausdrucks mit Lazy Evaluation Strategie berechnen. Dies erlaubt die Verwendung unendlicher Listen, ohne in den PROLOG-Interpreter einzugreifen. Wenn der Beweis mit einem vorher berechneten Wert gescheitert ist, kann das Backtracking des PROLOG-Interpreters die Reduktion eines weiteren Listenelementes anstoßen.

Die Sprache HASL ([Abramson 1986]) ist ein in PROLOG implementierter Nachfolger der funktionalen Programmiersprache SASL (was auch durch die Namensgebung angedeutet werden soll). Im Zusammenhang mit der logischen Programmierung wird sie durch eine Erweiterung interessant, die Abramson ([Abramson 1986]) als "unification based conditional binding" bezeichnet. Der Ausdruck

```
A {- B => C;D
```

bedeutet: Unifiziere B mit A; falls die Unifikation mit der Substitution $\sigma$ erfolgreich ist, hat der Ausdruck den Wert $C\sigma$, ansonsten ist der Wert D (vgl. dazu die "conditional let-clause" von FRESH, [Smolka 1985]).

## 3.2 SASLOG im Vergleich

Die in dieser Arbeit vorgestellte Sprache SASLOG basiert nicht auf der Erweiterung einer logischen Sprache um funktionale Aspekte (z.B. durch Gleichheit) oder umgekehrt. Sie ist eine Vereinigung zweier bestehender Sprachen:

- PROLOG ([Colmerauer u.a. 1972], [Clocksin, Mellish 1981/84]) ist die wohl bekannteste logische Programmiersprache. Sie beruht auf der prozeduralen Interpretation der Hornklausellogik erster Stufe.

- SASL ([Turner 1983]) zeichnet sich aus durch Lazy Evaluation und Funktionen höherer Ordnung.

SASLOG ist eine symmetrische Sprachintegration. Die funktionale und die logische Komponente sind gleichwertig. Im funktionalen Teil ist die Verwendung relationaler Ausdrücke ebenso möglich, wie die Reduktion funktionaler Ausdrücke im logischen Teil.

Ziel der Entwicklung von SASLOG war es, eine möglichst pure, aber auch praktisch verwendbare Sprache zu entwerfen, in der sowohl wissensbasierte Systeme als auch algorithmische Verfahren adäquat repräsentiert werden können. So wurde ein Effizienzgewinn dadurch erzielt, daß man die Reduktion nur auf Grundtermen ausführt, statt wie bei Narrowing Variablen zu erlauben. Dadurch ist das Beweisverfahren aber nicht vollständig. SASLOG gehört also nach der Klassifisierung von [Reddy 1986] nicht zu den Sprachen der Klasse N, die Narrowing als operationale Semantik funktionaler Ausdrücke haben, und somit auch nicht zur Klasse F, deren Sprachen ja eine Erweiterung der N-Sprachen um freie Mengen sind. Zwar hat SASLOG ein Konzept der Mengenabstraktion, jedoch handelt es sich hierbei nicht um freie Mengen, da als Werte keine freien Variablen erlaubt sind. Natürlich kann SASLOG auch nicht in die Klasse R eingeteilt werden. Während R-Sprachen nämlich funktionale Programme in logische Programme mit Resolution als operationaler Semantik übersetzen, beruht die funktionale Berechnung von SASLOG auf der Reduktion.

Die Mengenabstraktion, die in das schon bestehende ZF-Konstrukt von SASL integriert ist, dient der Berechnung von PROLOG-Anfragen in der SASL-Komponente. Im Unterschied zu SUPERLOGLISP werden die Relationsdefinitionen aber nicht ebenfalls durch Mengenabstraktion sondern wie in PROLOG durch Hornklauseln beschrieben.

Die Einbindung der funktionalen Komponente in PROLOG ist eine Integration auf der Termebene (strong amalgamation). Der erweiterte (semantische) Unifikationsalgorithmus beruht nach der Klassifizierung von [Dincbas, van Hentenryck 1987] nicht wie derjenige von FUNLOG auf Lazy Derivation, sondern auf Delayed Evaluation. Der Hauptunterschied liegt darin, daß Terme mit freien Variablen hier nicht irreduzibel sind. Stattdessen werden freie Variablen mit dem Wert $\perp$ ("bottom", "undefiniert") belegt, wenn die Reduktion eines Terms für die Unifikation notwendig wird. So ist auch der Wert des ganzen Terms $\perp$, wenn die zu reduzierende Funktion in diesem Argument strikt ist.

Die bisher genannten Sprachen mit Lazy Evaluation ([Bellia, Levi 1986], [Hansson, Haridi, Tärnlund 1982], [Bellia, Degano, Levi 1982], [Voda,Yu 1984]) unterstützen verzögerte Berechnung nur im funktionalen Teil. Lazy Evaluation ist jedoch gut mit der Backtracking-Strategie von PROLOG vereinbar, was auch [Narain 1986] für die Definition von Reduktionsregeln in PROLOG ausnutzt. [Wadler 1985] zeigt, wie man Backtracking durch Lazy Evaluation ersetzt. Die erweiterten ZF-Ausdrücke und die semantische Unifikation in SASLOG erlauben dagegen eine Verzahnung von Lazy Evaluation und Backtracking: Der Reduktionsalgorithmus von SASL verzögert eine Berechnung, bis der Wert eines Ausdrucks tatsächlich benötigt wird. Ein erweiterter ZF-Ausdruck ist eine Liste mit (möglicherweise unendlich vielen) PROLOG-Lösungen, wobei jede Lösung durch Backtracking erst berechnet wird, wenn der Wert für die weitere Berechnung notwendig ist. Umgekehrt werden durch Delayed Evaluation in der Unifikation nur solche SASL-Ausdrücke berechnet, die mit anderen Termen unifiziert werden sollen. Dabei kann durch Backtracking jeweils die Berechnung eines weiteren Elementes einer (möglicherweise unendlichen) Datenstruktur angestoßen werden. Dafür ist kein eigenes Konstrukt nötig, die Reduktion wird durch den Unifikationsalgorithmus für den Benutzer unsichtbar gesteuert.

Auch LISPLOG hat die Möglichkeit, Streams von PROLOG-Lösungen zu bearbeiten ([Dahmen 1987]). Jedoch liegt hier die Verwaltung in der Verantwortung des Programmierers.

# 4 Die Ausgangssprachen

SASLOG ist eine Kombination der funktionalen Programmiersprache SASL und der logischen Sprache PROLOG, die im folgenden kurz vorgestellt werden.

## 4.1 PROLOG

PROLOG ([Colmerauer u.a. 1972]) ist die wohl bekannteste logische Programmiersprache. Aus diesem Grund wird an dieser Stelle nur eine kurze Einführung gegeben, eine detaillierte Beschreibung findet man in [Clocksin, Mellish 1981/84].

Ein PROLOG-Programm ist eine partiell geordnete Folge von Hornklauseln. Eine Hornklausel besteht aus der Konklusion und keiner, einer oder mehreren Prämissen, die von der Konklusion durch ":-" getrennt sind, mit der Bedeutung, daß die Konjunktion der Prämissen die Konklusion impliziert. Eine Klausel ohne Prämisse heißt ein Fakt. Konklusion und Prämissen nennt man Literale, sie sind Relationen mit einem n-stelligen Prädikat. Die Argumente der Literale, die Terme, können Konstanten (Atome), Zahlen, Variablen oder Listen von Termen sein. Variablen sind Bezeichner, die mit einem Großbuchstaben beginnen.

Beispiel:
```
sterblich(X) :- mensch (X).
sterblich(X) :- tier(X).
mensch(sokrates).
grieche(sokrates).
```

Eine Berechnung wird gestartet durch eine Anfrage, das ist eine Hornklausel ohne Konklusion, die durch "?-" eingeleitet wird.

Beispiel:
```
?- sterblich(Y), grieche(Y).
```

Die Anfrage stellt eine Konjunktion von Zielen dar, die alle mit konsistenter Variablenbindung bewiesen werden müssen (UND-Nichtdeterminismus). Die Beweisprozedur von PROLOG ist die ***Resolution*** mit ***Tiefensuche*** und ***Backtracking***. Der UND-Nichtdeterminismus wird dadurch aufgelöst, daß das nächste zu beweisende Ziel nach einer Art last-in-first-out-Methode ausgewählt wird. Dabei wird das jeweils erste der noch zu beweisenden Ziele - es wird im folgenden als das aktuelle Goal bezeichnet - mit der Konklusion einer Klausel unifiziert. Bei erfolgreicher Unifikation wird das aktuelle Goal durch die Prämissen der Klausel ersetzt und die bei der Unifikation erzeugten Variablenbindungen auf alle noch zu

beweisenden Ziele angewandt. Ist das aktuelle Goal mit mehreren Klauselköpfen unifizierbar (ODER-Nichtdeterminismus), entscheidet die Ordnung der Klauseln über die Anwendungsreihenfolge. Dabei wird eine spätere Klausel nur angewandt, wenn alle früheren nicht zu einer Lösung geführt haben (Backtracking).

## 4.2 SASL

### 4.2.1 Konzepte

SASL ([Turner 1983]), wurde 1976 von David Turner an der Universität St. Andrews entwickelt. Die folgende Einführung geht kurz auf wichtige Merkmale von SASL ein, für eine ausführliche Darstellung siehe [Turner 1983] oder [Richards 1984].

SASL ist eine applikative Sprache, d.h. eine Berechnung beruht ausschließlich auf der Anwendung von Funktionen auf Argumente. Ein applikatives Programm besteht aus Ausdrücken, die Werte repräsentieren. Eine entscheidende Eigenschaft der Ausdrücke in SASL ist die ***Referential Transparency***. Sie besagt, daß das einzig relevante an einem Ausdruck sein Wert ist und daß jeder Unterausdruck durch einen anderen gleichwertigen Ausdruck ersetzt werden kann. Außerdem ist der Wert eines Ausdrucks immer der gleiche, wo er auch vorkommt, vorausgesetzt, der Geltungsbereich der Definitionen für die in dem Ausdruck vorkommenden Namen wird nicht verlassen ([Stoy 1977]). Voraussetzung für Referential Transparency ist die Seiteneffektfreiheit. So sind z.B. keine Änderungen globaler Variablen durch Zuweisungen erlaubt.

Eine wichtige Rolle spielt die Reihenfolge der Auswertung von Ausdrücken. Im Gegensatz zur Strategie der ***Applicative-order Evaluation*** (auch ***call-by-value*** genannt), bei der die Auswertung auf der tiefsten Schachtelungsebene des jeweils am weitesten links stehenden Ausdrucks beginnt (innermost-left), wird bei der ***Normal-order Evaluation*** (***call-by-name***) zuerst der äußerste linke Funktionsaufruf ausgewertet (outermost-left). Man kann es auch so ausdrücken, daß bei der call-by-value Strategie zuerst alle Argumente eines Aufrufs ausgewertet werden, während mit call-by-name die Argumente unausgewertet in den Funktionsrumpf übergeben und erst bei Bedarf berechnet werden.

In Erweiterung der Normal-order Evaluation wird bei der für SASL implementierten ***call-by-need***-Strategie ein Ausdruck erst ausgewertet, wenn sein Wert tatsächlich benötigt wird, und überall durch seinen Wert ersetzt. So wird eine mehrfache Auswertung des gleichen Ausdrucks vermieden.

Die Verzögerung der Berechnung von Ausdrücken, die man auch als ***Lazy***

***Evaluation*** oder ***Delayed Evaluation*** bezeichnet, hat Auswirkung auf die Terminierung der Berechnung. Normal-order Evaluation garantiert, daß der Endzustand einer Berechnung stets erreicht wird, sofern es eine terminierende Folge von Reduktionsschritten gibt ([Wadsworth 1971]). Sie erlaubt die Behandlung unendlicher Datenstrukturen, die eine endliche Darstellung haben.

Eine Funktion, deren Resultat nur definiert ist, wenn ihr Argument definiert ist, heißt strikt. Lazy Evaluation erlaubt ***nicht-strikte*** Funktionen, also Funktionen, deren Wert auch für nichtdefinierte Argumente definiert ist. Die Funktion

```
IF a THEN b ELSE c
```

ist nicht-strikt in Bezug auf b und c. So hat z.B. der Ausdruck

```
IF TRUE THEN b ELSE c
```

den Wert b, auch wenn c nicht definiert ist, da wegen Lazy Evaluation die Berechnung von c nicht angestoßen wird.

Alle Werte von SASL werden einheitlich behandelt. Das erlaubt insbesondere, daß Funktionen selbst wieder funktionale Argumente und Werte haben können. Eine solche Funktion, deren Definitions- und/oder Wertebereich Funktionen enthält, ist eine ***Funktion höherer Ordnung***.

### 4.2.2 Ausdrücke

In SASL gibt es 6 verschieden Typen von Objekten: Zahlen, boolesche Werte (TRUE und FALSE), Zeichen (erkennbar an vorgestelltem %, z.B. %a), Funktionen, Listen (Sequenzen von Objekten) und den Wert "undefiniert". Objekte aller Typen können in einheitlicher Weise verwendet werden, sie können

- mit Namen versehen werden,
- Werte und Argumente von Funktionsanwendungen sein,
- Komponenten von Listen sein,
- Werte von Ausdrücken sein.

Weil der Wert "undefiniert" in Fehlersituationen auftritt, z.B. als Wert von Ausdrücken, die bestimmte Typrestriktionen nicht beachten (z.B. 4 * TRUE) oder bei nicht-terminierenden Berechnungen, existiert keine Repräsentation für die Eingabe. Oft wird als Abkürzung $\perp$ verwendet, was aber kein Bestandteil von SASL ist. Listen können aus Objekten beliebigen Typs gebildet werden. Für den Aufbau von Listen

steht der rechtsassoziative Infix-Operator ":" und für den Zugriff auf das erste Element und den Rest einer Liste die Funktionen hd und tl zur Verfügung. Die leere Liste wird als [ ] dargestellt. Endliche Listen kann man statt durch

```
a1 : a2 : ... : an : [ ]
```

in der Form

```
[a1,a2,...,an]
```

schreiben. Ist das zweite Argument von ":" keine Liste, so nennt man das Konstrukt eine schwach-terminierende Liste. Strings sind Listen von Zeichen.

Konstanten, Listen und Namen sind einfache Ausdrücke. Jeder andere Ausdruck wird ebenfalls zu einem einfachen Ausdruck, indem man ihn in Klammern einschließt.

Die wichtigste syntaktische Form ist die Funktionsanwendung. Sie besteht aus zwei aufeinanderfolgenden Ausdrücken, einer Funktion und ihrem Argument. f x bezeichnet die Anwendung von f auf x. Zweideutigkeiten werden beseitigt, indem implizit Linksklammerung angenommen wird. Soll von dieser Regel abgewichen werden, muß explizit geklammert werden. So bedeutet

```
f x y
```

die Anwendung von f auf x, deren funktionaler Wert dann auf y angewandt wird:

```
((f x) y).
```

In dem Ausdruck f (x + 1) sind die Klammern notwendig, weil f x + 1 äquivalent wäre zu (f x) + 1.

Ist die Funktion f eine Liste, so ist f n das n-te Element der Liste f, also [2,4,6] 2 ist gleich 4.

In SASL gibt es vordefinierte Infixoperatoren (z.B. arithmetische Operatoren oder den Listenkonstruktor ":"), einstellige Präfixoperatoren, einen einstelligen Postfixoperator (x... bezeichnet die Liste der ganzen Zahlen beginnend bei x) und einen dreistelligen Operator bestehend aus zwei Symbolen ("->" und ";") für bedingte Ausdrücke:

```
x < 0 -> -x ; x
```
berechnet den Absolutbetrag des numerischen Wertes x.

Neben der Beschreibung von Listen durch die primitiven Listenoperationen existieren

in SASL die sogenannten ZF-Ausdrücke. Ihre Darstellung orientiert sich an der Zermelo-Fränkel-Mengenabstraktion $\{E \mid G_1,...,G_n\}$, wobei ein ZF-Ausdruck allerdings eine Liste darstellt, bei der insbesondere Elemente mehrfach auftreten können und die Reihenfolge von Bedeutung ist. Die allgemeine Form eines ZF-Ausdrucks ist:

$$[E;Q_1;...;Q_n]$$

E ist der Resultatsterm, jeder der sogenannten Qualifier $Q_i$ hat eine der folgenden Formen:

| | |
|---|---|
| $V_i$ <- $E_i$ | (Generator) |
| $E_i$ | (Filter), $E_i$ boolescher Ausdruck |

Beispiel: [[a,b]; a <- l1; b <- l2; a < b] beschreibt das kartesische Produkt der Listen l1 und l2, wobei das erste Element kleiner ist als das zweite. Man kann den Audruck lesen als: "Die Liste aller Paare [a,b], für die gilt, daß a ein Element von l1, b ein Element von l2 und a kleiner als b ist."

Weitere zulässige Ausdrücke sind die where-Ausdrücke zur lokalen Definition von Namen, die im folgenden Abschnitt erläutert werden.

### 4.2.3 Funktionsdefinitionen

In SASL gibt es zwei Arten, Werten einen Namen zuzuordnen, entweder durch formale Parameter von nicht-nullstelligen Funktionen oder durch globale oder lokale Funktionsdefinitionen.

Eine globale Funktionsdefinition wird durch das Schlüsselwort def eingeleitet. Eine Funktionsdefinition hat die Form

<Name> <Parameterliste> = <Ausdruck>

Ist die Parameterliste leer, so handelt es sich um eine Konstantendefinition. <Ausdruck> heißt der definierende Ausdruck oder Funktionsrumpf.

Beispiele:
```
def suc n = n + 1          definiert die Nachfolgerfunktion.
def fak n = n = 0 -> 1; n * fak (n - 1)     ist die Fakultätsfunktion.
```

Fallunterscheidungen kann man auch dadurch erzielen, daß man formale Parameter durch Konstanten ersetzt:

```
def  fak 0 = 1
     fak n = n * fak (n - 1)
```

Sind die Fälle nicht disjunkt, wird der erste passende Fall angewandt, bei dem die formalen Parameter mit den aktuellen Parametern unifiziert werden können.

Außer Konstanten können auch strukturierte Parameter verwendet werden. So läßt sich die Funktion hd, die als Wert das erste Element einer Liste liefert, sehr kurz definieren durch

```
def hd (a : x) = a
```

Eine Fortführung dieses Konzepts stellen Namelist-Definitionen dar, die mehrere in einer Listenstruktur enthaltene Parameter simultan definieren:

```
def (a : x) = [1,2,3,4]
```

ist gleichbedeutend mit

```
def a = 1
def x = [2,3,4]
```

Where-Ausdrücke dienen zur lokalen Definition von Namen. Sie bestehen aus einem Ausdruck, in dem die Definitionen gelten sollen, gefolgt von dem Schlüsselwort where und beliebig vielen durch Semikolon getrennten Definitionen. Der Definitionsbereich erstreckt sich über den Ausdruck vor dem where und alle Definitionen; die Reihenfolge spielt keine Rolle. Eingebettete where-Ausdrücke können außerhalb gültige Definitionen überschreiben:

```
a + (a + b  where  a = 3 * b;  b = 12) where a = 2
```

hat den Wert 50.

In SASL sind alle Funktionen einstellig, auch wenn die Form der Definitionen eine andere Sicht nahelegt. Die zweistellige Funktion plus

```
def plus n m = n + m
```

ist z.B. ein Funktional, das seinem Argument n die Funktion zuordnet, die zu ihrem Argument n addiert, was in der folgenden gleichwertigen Definition deutlich wird:

```
def  plus n  =  f  where  f m = n + m
```

Dies beruht auf einem Ergebnis der Kombinatorlogik zur variablenfreien Darstellung von Ausdrücken ([Schönfinkel 1924]). Man bezeichnet den Prozeß, mehrstellige Funktionen auf einstellige zurückzuführen, als ***currying*** und die Funktionen als ***curried functions***. Diese Begriffe sind vom Namen des Mathematikers H.B. Curry abgeleitet, der die Idee von Schönfinkel weiterentwickelte ([Curry, Feys 1958]).

So ist es auch möglich, eine n-stellig definierte Funktion f auf $m < n$ Argumente anzuwenden. Der Wert ist ein Funktional, das auf n-m Argumente angewandt genau die Funktion f realisiert.

Beispiel: `def suc = plus 1`
definiert die Nachfolgerfunktion suc mit obiger Definition für plus.

# 5 Die SASLOG-Syntax

## 5.1 Objekte in SASLOG

Bevor die Verbindung der funktionalen und der logischen Berechnung in SASLOG beschrieben wird, werden kurz die speziellen Implementierungen von PROLOG und SASL erläutert, auf denen die Integration aufbaut.

PROLOG hat viele außerlogische Bestandteile, von denen die Operationen assert und retract zum dynamischen Ändern der Datenbasis und der Cut-Operator, der Backtracking verhindert, zu den problematischsten gehören. Sie erschweren die Beschreibung der Sprachsemantik und widersprechen dem deklarativen Charakter der Sprache.

Die PROLOG-Komponente von LISPLOG ([Boley 1986]) - einer LISP/PROLOG-Vereinheitlichung, die die Grundlage für die logische Seite von SASLOG bildet - hat deshalb die Verwendung des Cut eingeschränkt. Es ist nur der initiale Cut erlaubt, d.h. der Cut darf nur als erste Prämisse einer Klausel stehen. An dieser Stelle bewirkt er, daß eine einmal als anwendbar erkannte Klausel - deren Konklusion mit dem Goal unifizierbar ist - auch zum Beweis herangezogen wird. Dadurch wird die deklarative Lesbarkeit eines PROLOG-Programms erhöht.

Eine für SASLOG neu eingeführte Einschränkung ist die fehlende Unterstützung von dynamischen Änderungen der PROLOG-Datenbasis durch Wegfall der Kommandos assert und retract. Diese sind nur als Kommandos für die Interaktion auf dem SASLOG-Toplevel vorgesehen.

Die funktionale Komponente von SASLOG bildet der in [Nökel, Rehbold 1986] beschriebene SASL-Interpreter. Das System erlaubt keine globalen Namelist-Definitionen, unterstützt aber deren Verwendung in where-Ausdrücken.

Die logische und die funktionale Komponente müssen auf den gleichen Objekten operieren. Das sind

- Konstanten,
- Zahlen,
- boolesche Werte (TRUE und FALSE),
- Listen,
- Funktionen,
- logische Variablen,
- der Wert "undefiniert" (im Text dargestellt durch $\perp$).

Konstanten wurden als ein neuer Datentyp für SASL eingeführt, sie entsprechen den Atomen in PROLOG. Konstanten sind Bezeichner, die durch ein Apostroph gekennzeichnet sind (vgl. "quote" in LISP).

Beispiele: 'john, 'a

Die Syntax für Listen wurde von Edinburgh PROLOG übernommen. Sie werden begrenzt durch eckige Klammern, die einzelnen Elemente werden durch Kommata getrennt.

Beispiel: [1,2,'a,[3,4]]

Der senkrechte Strich dient zum Trennen von Anfang und Rest einer Liste. Statt a : l wie in SASL schreibt man jetzt also [a | l].

Die logischen Variablen erkennt man am vorgestellten Unterstrich.

Beispiele: _x, _Ort, _z1

Die anonyme Variable wird dargestellt durch _?. Jede anonyme Variable steht für eine Variable, die nur einmal vorkommt.

Die Reduktion arbeitet auf Grundtermen. Freie logische Variablen werden zu ⊥ reduziert.

Man beachte, daß Bezeichner für Funktionen und Relationen zwar im Prinzip frei wählbar sind, zur Vermeidung von Konflikten dürfen jedoch nicht gleichzeitig eine Funktion und eine Relation mit übereinstimmenden Namen gültig sein.

Die Gesamtheit der vordefinierten Funktionen von SASL wird als Prelude bezeichnet. Das SASLOG-Prelude geht aus dem SASL-Prelude durch Hinzufügen einiger vordefinierter Relationen hervor. So ist z.B. die Relation == zur Unifikation zweier Terme im Prelude enthalten. Die Bezeichnung mit doppeltem Gleichheitszeichen wurde gewählt, um die Unifikation von der Gleichheitsfunktion in SASL zu unterscheiden.

Die hier beschriebene Originalsyntax wurde im Rahmen dieser Arbeit noch nicht realisiert, sie wird aber wegen der besseren Lesbarkeit im Text verwendet. Eine vollständige Beschreibung der Originalsyntax und der implementierten, an LISP angepaßten L-Notation findet sich im Anhang.

## 5.2 Relationen in SASL

Die Verbindung von SASL nach PROLOG wird durch zwei Konstrukte ermöglicht:

- prove-Ausdrücke
- ZF-Ausdrücke

Überall dort, wo ein boolescher Wert erwartet wird, kann ein Ausdruck der Form

```
prove (prolog-goal)
```

stehen. `prolog-goal` ist eine durch Kommata getrennte Folge von PROLOG-Literalen, die konjunktiv verknüpft sind. Falls der Wert des Ausdrucks benötigt wird, wird der PROLOG-Interpreter mit `prolog-goal` als Anfrage aufgerufen. Ist der Beweis erfolgreich, liefert der Ausdruck `TRUE`, scheitert der Beweis, ist der Wert `FALSE`.

Betrachte als Beispiel die folgende Funktion `gm`, die aus einer gegebenen Liste von Personen die Liste derjenigen Personen berechnet, die Großmutter sind:

```
def gm [ ]    = [ ]
    gm [p|l]  = prove (mother(p,_Z),parent(_Z,_?)) -> [p | (gm l)] ;
                                                       gm l
```

Man beachte, daß die PROLOG-Anfrage sowohl eine lokale PROLOG-Variable (`_Z`) als auch eine SASL-Variable (`p`) enthält.

Die zweite interessantere Verbindung von SASL nach PROLOG, bei der auch durch PROLOG erzeugte Variablenbelegungen nach SASL übergeben werden können, wird durch eine Erweiterung des schon existierenden ZF-Konstrukts erreicht (Abschnitt 4.2.2).

Die SASLOG-ZF-Ausdrücke haben folgende Syntax:

$[E;Q_1;...;Q_n]$

wobei E ein beliebiger Ausdruck ist (Resultatsterm) und jedes $Q_i$ eine der folgenden Formen hat:

| | |
|---|---|
| $V_i$ <- $E_i$ | (Generator) |
| $[V_{i1},...,V_{im}]$ <- prolog-goal$_i$ | (PROLOG-Generator) |
| $E_i$ | (Filter) |

PROLOG-Generatoren, die die Erweiterung gegenüber SASL-ZF-Ausdrücken bilden, unterscheiden sich von "normalen" Generatoren dadurch, daß links von "<-" mehrere Variablen stehen können und diese zu einer Liste zusammengefaßt sind.

Die Variablen $V_i$ bzw. $V_{ij}$ dürfen alle im Resultatsterm E vorkommen, jedoch nur in den Ausdrücken $E_k$ und PROLOG-Anfragen prolog-goal$_k$ mit $k > i$.

Ein Generator $V_i$ <- $E_i$ (lies "<-" als "$\epsilon$") bindet die Variable $V_i$ sukzessive an die Elemente der durch den Ausdruck $E_i$ beschriebenen Liste.

Ein PROLOG-Generator $[V_{i1},...V_{im}]$ <- prolog-goal$_i$ dagegen bindet die $V_{ij}$ jeweils simultan an die Werte, die für die in prolog-goal$_i$ vorkommenden logischen Variablen gleichen Namens während des Beweises erzeugt wurden (lies: "$V_{i1},...,V_{im}$ erfüllen prolog-goal$_i$"). prolog-goal$_i$ ist wie bei prove-Ausdrücken eine durch Kommata getrennte Konjunktion von Literalen.

Die Filter sind boolesche Ausdrücke, die Tests über Variablenwerte durchführen. Sie können also insbesondere selbst PROLOG-Anfragen in Form von prove-Ausdrücken sein.

Beispiel: Für ein Praktikum haben sich mehr Studenten angemeldet als es Plätze gibt. Als Kriterium für die Zulassung wird gefordert, daß alle Teilnehmer eine der Vorlesungen "Künstliche Intelligenz" oder "Expertensysteme" gehört haben. Die folgende Funktion teilnehmer trifft diese Auswahl:

```
def teilnehmer p  =  [[name,sem];
                      name <- anmeldung p;
                      [v,sem] <- vorlesung(name,_v),semester(name,_sem);
                      member ['KI,'XPS] v]
```

Die Funktion anmeldung hat als Wert die Liste der Studenten, die sich für das Praktikum p angemeldet haben. v und sem sind innerhalb der PROLOG-Anfrage logische Variablen und werden außerhalb der Anfrage zu SASL Parametern, was syntaktisch durch Weglassen des Unterstrichs deutlich wird.

Die verzahnte Berechnung mit Lazy Evaluation und Backtracking wird nun beispielhaft an folgender Datenbasis nachvollzogen:

```
meldeliste('KI-Praktikum,['Michael,'Peter,'Werner,'Paul]).

semester('Michael,7).
semester('Peter,9).
semester('Paul,7).
semester('Werner,7).

vorlesung('Michael,'KI).
vorlesung('Michael,'DB).
vorlesung('Werner,'Logik).
vorlesung('Peter,'KI).
vorlesung('Peter,'XPS).
vorlesung('Paul,'XPS).
```

Die folgende Definition realisiert die Funktion anmeldung:

```
def anmeldung p  =  head [x; [x] <- meldeliste(p,_x)]
                        where head [ ]     = [ ]
                              head [a | l] = a
```

Die Berechnung des Ausdrucks

```
E = teilnehmer 'KI-Praktikum
```

zeigt die Korrespondenz von Lazy Evaluation und Backtracking:

1. Der Wert des ersten Generatorausdrucks anmeldung 'KI-Praktikum ist ['Michael,'Peter,'Werner,'Paul].
2. Die Variable name erhält als Wert das erste Element dieser Liste, also 'Michael.
3. Der PROLOG-Interpreter wird gestartet, um die Anfrage ?-vorlesung('Michael,_v), semester('Michael,_sem). zu beweisen.
4. Die Variable v wird an 'KI und die Variable sem an 7 gebunden.
5. Der Filter member ['KI,'XPS] 'KI hat den Wert TRUE.
6. Damit sind alle Qualifier abgearbeitet und die Liste ['Michael,7] ist das erste Element des Ausdrucks E.
7. Wird das nächste Element des Ausdrucks benötigt, muß der letzte erfolgreiche Generator ein neues Element berechnen. In diesem Fall ist es der PROLOG-Generator und Backtracking wird angestoßen.
8. PROLOG findet die nächste Lösung mit 'DB für v und wiederum 7 für sem.

9. member ['KI,'XPS] 'DB = FALSE, so daß Backtracking erneut angestoßen wird.
10. Es gibt keine weitere Lösung für die PROLOG-Anfrage, also generiert der erste Qualifier einen neuen Wert für name, in diesem Fall 'Peter.
11. Der PROLOG-Interpreter wird neu gestartet mit der Anfrage ?-vorlesung('Peter,_v), semester('Peter,_sem). und ist erfolgreich mit den Bindungen 'KI und 9 für v bzw. sem.
12. 'KI ist in der Liste ['KI,'XPS] enthalten, so daß ['Peter,9] das zweite Element des ZF-Ausdrucks ist.
13. Wird ein weiteres Element gebraucht, berechnet die PROLOG-Anfrage die Werte 'XPS und 9. Diese werden ebenfalls akzeptiert, so daß man als drittes Element des Ausdrucks wiederum ['Peter,9] bekommt.
14. Bei der Berechnung einer weiteren Lösung scheitert der PROLOG-Interpreter und der erste Generator berechnet eine neue Bindung für name, nämlich 'Werner.
15. Die vom PROLOG-Interpreter für die Anfrage ?-vorlesung('Werner,_v), semester('Werner,_sem). berechnete Lösung 'Logik für v und 7 für sem wird durch den Filter abgelehnt, so daß der erste Generator einen neuen Wert ('Paul) für name erzeugt.
16. Die PROLOG-Anfrage ?-vorlesung('Paul,_v),semester('Paul,_sem). liefert die Werte 'XPS für v und 7 für sem.
17. member ['KI,'XPS] 'XPS ist TRUE und das vierte Element ist ['Paul,7].
18. Eine weitere Lösung kann nicht berechnet werden. Backtracking des PROLOG-Interpreters findet mit 'Paul keine weitere Lösung und auch für name kann kein weiterer Wert gefunden werden, so daß die Berechnung des Ausdrucks E terminiert.

Man hat also

teilnehmer 'KI-Praktikum = [['Michael,7],['Peter,9],['Peter,9],['Paul,7]].

Man sieht insbesondere, daß Studenten, die sowohl die Vorlesung "Künstliche Intelligenz" als auch "Expertensysteme" gehört haben, in der Liste mehrfach vorkommen.

Die obige Funktion teilnehmer kann auch geschrieben werden in der Form

```
def teilnehmer p  =  [[name,sem];
                      [name,v,sem] <-  meldeliste(p,_x),
                                       member-r(_name,_x),
                                       vorlesung(_name,_v),
                                       semester(_name,_sem);
                      member ['KI,'XPS] v]
```

Die Definition von member-r ist im SASLOG-Prelude enthalten. Das Suffix "-r" deutet an, daß es sich um eine Relation und nicht um die SASL-Funktion member handelt:

```
member-r(_e,[_e|_?]).
member-r(_e,[_?|_l]) :- member-r(_e,_l).
```

In dieser zweiten Version wird die Berechnung der Vorlesungen, die die auf der Meldeliste eingetragenen Studenten besucht haben, rein im logischen Teil durchgeführt. Dazu ist die Relation member-r notwendig, die das Durchsuchen der Meldeliste explizit durchführt, was durch die Verknüpfung von Reduktion und Backtracking in der ersten Version implizit geschah.

## 5.3 Funktionale Ausdrücke in PROLOG

Jeder Term einer Prämisse einer Hornklausel bzw. einer Anfrage kann ein beliebiger SASL-Ausdruck sein. Reduzierbare Ausdrücke in der Konklusion sind nicht erlaubt. Enthält ein funktionaler Term logische Variablen, müssen diese bei der Reduktion gebunden sein. Der Zeitpunkt der Reduktion wird durch den Unifikationsalgorithmus bestimmt (siehe Abschitt 6.2).

Natürlich können die Terme auch wieder PROLOG-Goals enthalten (durch prove- und ZF-Ausdrücke) genauso wie die Terme in PROLOG-Anfragen von prove- und ZF-Ausdrücken wieder SASL-Terme sein können. So ist eine beliebig tiefe Schachtelung der Übergänge möglich.

Wegen der Integration auf Termebene muß es keinen Operator geben, der die Auswertung von Ausdrücken anstößt, wie das "is" in PROLOG. Da jeder Term ein funktionaler Ausdruck sein kann, reicht das normale "unify" (dargestellt durch ==) völlig aus.

Viele aus PROLOG bekannte vordefinierte Prädikate, z.B. zum Vergleich arithmetischer Werte, sind in SASLOG nicht mehr notwendig. Auch auf den Aufruf boolescher Funktionen als Prämissen, wie er z.B. in LISPLOG für LISP-Funktionen möglich war, kann verzichtet werden. Den gleichen Effekt erreicht man durch die Unifikation von Aufrufen der entsprechenden SASL-Funktionen mit booleschen Werten.

Beispiel: Eine quadratische Gleichung $ax^2 + bx + c = 0$ sei gegeben durch die Liste der Koeffizienten [a,b,c]. Die Relation quadrat berechnet die Lösungen dieser Gleichung:

```
quadrat([_a,_b,_c],_roots)   :-
                quadrat1(_a,_b,(dis _a _b _c),_roots).

quadrat1(_a,_b,_d,[ ])   :-   (_d < 0) == TRUE.
quadrat1(_a,_b,0,[_r])   :-   _r == (0 - _b) / (2 * _a).
quadrat1(_a,_b,_d,[_r1,_r2])   :-
                (_d > 0) == TRUE,
                _r1 == ((0 - _b) + (sqrt _d)) / (2 * _a),
                _r2 == ((0 - _b) - (sqrt _d)) / (2 * _a).

def dis a b c = (b * b) - (4 * a * c)
```

Während im PROLOG-Teil in den Klauseln für quadrat1 getestet wird, um welchen Fall es sich dabei handelt (ob die Diskriminante kleiner, gleich oder größer als 0 ist, also ob es keine, eine oder zwei Lösungen für x gibt), wird die eigentliche Berechnung der Lösungen in SASL ausgeführt.

Die Werte funktionaler Terme können auch unendliche Datenstrukturen sein. Die Lazy Evaluation-Strategie von SASL und der Unifikationsalgorithmus sorgen dafür, daß ein Ausdruck nur soweit berechnet wird, wie es notwendig ist. So kann man z.B. einen Term nacheinander - gesteuert durch Backtracking - mit den Elementen einer (möglicherweise unendlichen) Liste unifizieren. Dazu wird das oben definierte Prädikat member-r aufgerufen, das im SASLOG-Prelude enthalten ist.

```
?-member-r(_x, (from _Z where from n = [n| (from (n+1))])), p(_x).
```

Der Ausdruck (from _Z where from n = [n| (from (n+1)) stellt die Liste der natürlichen Zahlen beginnend mit dem Wert von _Z dar. _x wird nacheinander mit diesen Zahlen unifiziert. Die Liste wird so weit berechnet, bis ein Element mit _x unifizierbar ist. Scheitert das nachfolgende Goal p(_x) mit diesem Wert, stößt Backtracking die weitere Berechnung der Liste an.

Es besteht natürlich auch mit Lazy Evaluation die Gefahr einer nicht-terminierenden Berechnung. Dies kann der Fall sein, wenn bei einem Aufruf von member-r(_x,_l), das Argument _x mit keinem Element einer unendlichen Liste _l unifizierbar ist oder wenn z.B. zwei Terme verglichen werden, die als Werte die gleiche unendliche Datenstruktur haben.

Die Argumente der metalogischen Prädikate not (negation as failure) und call sind PROLOG-Anfragen und werden nicht als Terme reduziert. Ist das Argument von call nicht in Form einer Struktur sondern als Liste gegeben, wird es vor dem Beweis in eine Struktur umgewandelt. Auf diese Weise wird das bekannte PROLOG-Prädikat "=.." zur Umwandlung von Listen in Strukturen überflüssig.

Beispiel: Sowohl ?-call(p([1,2]))
als auch ?-call([p,[1,2]])
rufen den Interpreter mit p([1,2]) auf.

# 6 Operationale Semantik

SASLOG ist eine Integration zweier existierender Sprachen, die die beiden Teilsprachen als Untermengen hat. Weil die in den Ausgangssprachen SASL und PROLOG geschriebenen Programme auch in der neuen Sprache noch lauffähig sein sollen, basiert die operationale Semantik von SASLOG auf den operationalen Semantiken für SASL und PROLOG. In den folgenden beiden Abschnitten wird die Semantik der Sprachkonstrukte, die für die Verbindung zwischen beiden Sprachen wichtig sind, näher beschrieben.

## 6.1 Erweiterte Reduktionssemantik für SASL

Die operationale Semantik für SASL wird bestimmt durch die Definition der Reduktionsregeln zusammen mit der Reihenfolge ihrer Anwendung (Normal-order Reduction). Wie in Abschnitt 5.2 beschrieben, ergaben sich Änderungen der SASL-Komponente durch die Einführung von Konstanten und prove-Ausdrücken sowie die Erweiterung der ZF-Ausdrücke um PROLOG-Generatoren.

Die Erweiterung von SASL um Konstanten ist unproblematisch. Konstanten werden wie die Zahlen und die booleschen Werte TRUE und FALSE behandelt. Ihre Einführung erfordert eine kleine Erweiterung der Reduktionsregel für die Gleichheit, um zwei Konstanten vergleichen zu können. Die vordefinierte Funktion isconstant testet für ein Argument x, ob es vom Typ ***constant*** ist (vgl. isboolean, isnumber):

| | | | |
|---|---|---|---|
| isconstant x | = TRUE, | falls x eine Konstante ist |
| isconstant x | = FALSE, | falls x keine Konstante ist |
| isconstant x | = ⊥, | falls x undefiniert ist |

Logische Variablen werden zu ⊥ reduziert:

_x = ⊥, für alle logischen Variablen _x.

Es bleibt, die Übersetzung der prove- und der erweiterten ZF-Ausdrücke in Kombinatorgraphen sowie die Reduktionsregeln der neu eingeführten Kombinatoren zu erläutern. Alle anderen Kombinatoren und ihre Reduktionsregeln bleiben unverändert (siehe [Nökel, Rehbold 1986]).

### 6.1.1 prove-Ausdrücke

Das Argument eines prove-Ausdrucks ist eine logische Anfrage. Die Reduktionsregel für prove lautet wie folgt:

| | | |
|---|---|---|
| prove prolog-goal | = TRUE, | falls der Resolutionsbeweiser mit prolog-goal als Anfrage eine Lösung berechnet |
| prove prolog-goal | = FALSE, | falls der Beweis von prolog-goal in endlicher Zeit scheitert |
| prove prolog-goal | = ⊥, | sonst |

Das Ziel prolog-goal ist eine Folge von Prädikatsaufrufen, den Literalen. Prädikate können als boolesche Funktionen und prolog-goal somit als Konjunktion boolescher Werte interpretiert werden. Der Ausdruck prove prolog-goal hat den Wert TRUE, falls prolog-goal bewiesen werden kann, also in dem System eine wahre Aussage ist. Der Kombinator prove zeigt dem Reduktionsalgorithmus an, daß der Wahrheitswert von prolog-goal nicht durch Reduktion, sondern durch die Resolution in der logischen Komponente berechnet werden muß.

### 6.1.2 ZF-Ausdrücke

Die ZF-Ausdrücke von SASL werden vor der Abstraktion in gleichwertige ZF-freie Ausdrücke umgewandelt ([Nökel, Rehbold 1986]). Dieser Umwandlungsalgorithmus wurde dahingehend geändert, daß auch die PROLOG-Generatoren korrekt erfaßt werden.

Ein ZF-Ausdruck Z hat folgende Form:

$$Z = [E;Q_1;...;Q_n] ,$$

wobei der Resultatsterm E ein beliebiger Ausdruck ist und jeder der sogenannten Qualifier $Q_i$ eine der folgenden Formen hat:

| | |
|---|---|
| $V_i$ <- $E_i$ | (Generator) |
| $[V_{i1},...,V_{im}]$ <- prolog-goal$_i$ | (PROLOG-Generator) |
| $E_i$ | (Filter) |

Die Qualifier haben intuitiv folgende Bedeutung: Der Generator $V_i$ <- $E_i$ bindet die Variable $V_i$ sukzessive an die Elemente der Liste $E_i$. Der PROLOG-Generator projeziert nacheinander alle beim Beweis von prolog-goal$_i$ berechneten Variablenbindungen auf die Liste der Variablen $[V_{i1},...,V_{im}]$. Die Filter legen

Bedingungen fest, die die Variablenwerte erfüllen müssen.

Der Umwandlungsprozeß läuft wie folgt:

Def.: Sei $[V_{k1},...,V_{kr}]$ die Liste aller Generatorvariablen von Z in der Reihenfolge ihres Auftretens, wobei die Variablen von PROLOG-Generatoren in die Liste eingebettet werden. Die ***Tupelfunktion*** T(Z) ist dann definiert durch

$$T(Z) := [V_{kr},...,V_{k1}].$$

Def.: Ein ZF-Ausdruck $Z = [E;Q_1;...;Q_n]$ ist in ***Normalform***, wenn gilt $E = T(Z)$.

$$Z = [E;Q_1;...;Q_n]$$

ist äquivalent zu

$$Z' = \text{map } f\ [T(Z);Q_1;...;Q_n] \text{ where } f\ T(Z) = E$$

Mit map wird eine Funktion f auf jedes Element T(Z) der Liste angewandt, so daß es wieder die Form E hat. In einem ersten Schritt wird also ein beliebiger ZF-Ausdruck in Normalform umgewandelt, indem die durch E definierte Funktion aus Z herausgezogen wird.

Danach wird der in Normalform gegebene ZF-Ausdruck in einen gleichwertigen ZF-freien Ausdruck transformiert. Das Verfahren arbeitet rekursiv bis alle Qualifier $Q_i$ umgewandelt sind. Sei also $NZ_n$ ein ZF-Ausdruck in Normalform mit n Qualifiern. Ein äquivalenter ZF-freier Ausdruck norm($NZ_n$) ist wie folgt definiert:

1. Ist n = 0, d.h. der ZF-Ausdruck enthält keine Qualifier, dann gilt:
   $\text{norm}(NZ_0) := [[\ ]]$.
2. Wenn n > 0, dann gibt es drei Möglichkeiten:
   2.1. $Q_n = E_n$ (Filter):
   $\text{norm}(NZ_n) := \text{filter f norm}(NZ_{n-1}) \text{ where f } T(NZ_{n-1}) = E_n$
   2.2. $Q_n = V_n <- E_n$ (Generator):
   $\text{norm}(NZ_n) := \text{cp f norm}(NZ_{n-1}) \text{ where f } T(NZ_{n-1}) = E_n$
   2.3. $Q_n = [V_{n1},...,V_{nm}] <- \text{prolog-goal}_n$ (PROLOG-Generator):
   $\text{norm}(NZ_n) := \text{cpp f norm}(NZ_{n-1})$
   $\text{where f } T(NZ_{n-1}) = \text{goal prolog-goal}_n\ [\_V_{n1},...,\_V_{nm}]$

Die Funktion filter ist eine vordefinierte Listenfilterfunktion mit zwei Argumenten, einer booleschen Funktion p und einer Liste L. Der Wert von filter p L ist eine Liste mit den Elementen x aus L, für die p x = TRUE gilt.

Der Ausdruck $E_n$ eines Generators entspricht einer Funktion f, die in Abhängigkeit von $T(NZ_{n-1})$ eine Liste von Werten für $V_n$ berechnet. Die mit dem kartesischen Produkt verwandte Funktion cp fügt diese Werte in die durch die Qualifier $Q_1,...,Q_{n-1}$ definierte Liste $norm(NZ_{n-1})$ ein. Die Funktionen cp und join wurden unverändert von SASL übernommen. Sie sind Kombinatoren mit folgenden Reduktionsregeln:

```
cp f [ ]      =  [ ]
cp f [a|x]    =  append (join (f a) a) (cp f x)
join [ ] e    =  [ ]
join [a|x] e  =  [[a|e] | (join x e)]
```

Im Laufe der Transformation von PROLOG-Generatoren wird die Liste der Generatorvariablen in eine Liste mit gleichnamigen logischen Variablen umgewandelt. Auf das $prolog\text{-}goal_n$ und diese Liste wird die Funktion goal angewandt. Der PROLOG-Generator

$$[V_{n1},...,V_{nm}] \text{ <- } prolog\text{-}goal_n$$

steht also für die Form

$$[V_{n1},...,V_{nm}] \text{ <- goal } prolog\text{-}goal_n \ [\_V_{n1},...,\_V_{nm}].$$

Weil PROLOG-Generatoren an der Listenform der Generatorvariablen aber eindeutig erkannt werden, kann die Syntax vereinfacht und die Transformation automatisch vorgenommen werden.

Die eigentliche Verbindung zwischen SASL und PROLOG ergibt sich aus den Reduktionsregeln der Kombinatoren goal und next:

```
goal p-g r-l  =  [r-l-inst| (next p-c r-l)],   falls der Resolutionsbeweiser für
                                               p-g eine Lösung berechnet
goal p-g r-l  =  [ ],                          falls der Beweis von p-g in
                                               endlicher Zeit scheitert
goal p-g r-l  =  ⊥,                            sonst

next p-c r-l  =  [r-l-inst| (next p-c' r-l)],  falls der Resolutionsbeweiser
                                               beginnend im Zustand p-c eine
                                               weitere Lösung berechnet
next p-c r-l  =  [ ],                          falls der Beweis in endlicher Zeit
                                               scheitert
next p-c r-l  =  ⊥,                            sonst
```

Dabei bedeuten:

p-g: Das prolog-goal$_n$.

r-l: Die Resultatsliste, also die Liste der zu instantiierenden PROLOG-Variablen [_V$_{n1}$,...,_V$_{nm}$].

r-l-inst: Die instantiierte Resultatsliste, wobei die Ergebnissubstitution rekursiv auch auf die Werte der Variablen angewendet wird. Freie Variable werden auf $\perp$ ("bottom") abgebildet, den SASL-Wert für "undefiniert".

p-c: Die PROLOG-Continuation; sie enthält alle Daten, die den Zustand des PROLOG-Interpreters zum Zeitpunkt der Rückkehr nach SASL definieren.

p-c': Die neue PROLOG-Continuation.

Der Ausdruck goal p-g r-l entspricht der Mengendarstellung von PROLOG-Anfragen, wie sie in Kapitel 3 als Erweiterung funktionaler Programmierung um logische Aspekte erläutert wurde.

Beispiel: goal append(_X,_Y,[1,2,3]) [_X,_Y]

entpricht der Mengendarstellung

{[_X,_Y] | append(_X,_Y,[1,2,3])}.

Der Wert von goal p-g r-l ist die Liste aller möglichen Lösungen für die PROLOG-Anfrage p-g. Jede Lösung ist repräsentiert durch die entsprechende instantiierte Form der Resultatsliste r-l. Hier zeigt sich die Verzahnung von Lazy Evaluation und Backtracking, denn die Liste der Lösungen wird nicht auf einmal berechnet. Die Reduktion von goal p-g r-l liefert eine Liste mit der ersten Lösung. Die Berechnung der weiteren Lösungen wird verzögert. Sie sind bestimmt durch den Ausdruck next p-c r-l. Wird die nächste Lösung benötigt, also next p-c r-l reduziert, kann durch p-c der Zustand des PROLOG-Interpreters wiederhergestellt und zur Berechnung einer weiteren Lösung Backtracking angestoßen werden.

Die Variablen des PROLOG-Generators sind zu einer Liste zusammengefaßt, so daß dieser nicht eine Liste von Werten (wie der "normale" Generator), sondern eine Liste von Wertlisten berechnet. Diese dürfen nicht einfach vorn in die Listen der vorher berechneten Variablenwerte eingefügt werden (wie in join). Stattdessen müssen die Wertlisten und die Liste der durch die vorherigen Generatoren berechneten Werte zusammengesetzt werden (mit append). Darum wurden die Funktionen cpp und joinp, die den Funktionen cp und join sehr ähnlich sind, eingeführt (der Suffix "p" deutet ihre Verwendung in PROLOG-Generatoren an):

```
cpp f [ ]      =  [ ]
cpp f [a|x]    =  append (joinp (f a) a) (cpp f x)
joinp [ ] e    =  [ ]
joinp [a|x] e  =  [(append a e) | (joinp x e)]
```

## 6.2 Geänderte Beweisprozedur von PROLOG

Nach [van Emden, Kowalski 1976] ist die operationale Semantik einer Programmiersprache bestimmt durch die Definition eines implementationsunabhängigen Interpreters. Für die Prädikatenlogik verhält sich die Beweisprozedur wie ein solcher Interpreter. PROLOG liegt eine Teilmenge der Prädikatenlogik, die Hornklauselogik, zugrunde. [Bowen, Kowalski 1982] definieren den Toplevel eines Interpreters für Hornklauselprogramme einer Sprache L in einer Metasprache M, die ebenfalls durch Hornklauseln gegeben ist. Die folgenden Klauseln definieren in Anlehnung an diese Definition den Toplevel für die logische Komponente von SASLOG:

```
D1) demo(_Prog,[]).
D2) demo(_Prog,[_Goal|_Rest]) :- member(_Clause,_Prog),
                                 rename(_Clause,_Goals,_Variantclause),
                                 parts(_Variantclause,_Concl,_Premises),
                                 unify(_Concl,_Goal,_Substitution),
                                 append(_Premises,_Rest,_Goals),
                                 apply(_Goals,_Substitution,_Newgoals),
                                 demo(_Prog,_Newgoals).
```

Demo ist ein zweistelliges Prädikatssymbol aus M. Demo repräsentiert $\vdash_L$ genau dann, wenn für jede endliche Menge von Klauseln A und jede kopflose Klausel B (die Goals) über L gilt

$$A \vdash_L B \quad \text{genau dann, wenn} \quad Pr \vdash_M demo(A',B'),$$

wobei Pr eine Menge von Klauseln über M ist. A' und B' stellen die Repräsentationen von A bzw. B in M dar. L ist hier die logische Komponente von SASLOG.

D1 besagt, daß die leere Menge von Goals aus jedem Programm beweisbar ist. D2 gibt die Beweisprozedur für eine nichtleere Menge von Zielen an, die durch folgende Schritte bestimmt ist:

1. Wähle eine Klausel _Clause aus der Klauselmenge _Prog aus.
2. Benenne die Variablen der Klausel um, so daß sie von denen des Goals verschieden sind; man erhält die neue Klausel _Variantclause.
3. Spalte die Klausel auf in die Konklusion _Concl und die Prämissen _Premises.
4. Unifiziere die Konklusion mit dem ersten Goal. Wenn die Unifikation nicht erfolgreich ist, wähle eine neue Klausel aus (durch Backtracking; rename und parts sind deterministisch).
5. Hänge bei erfolgreicher Unifikation die Prämissen der Klausel vorne in die Liste der restlichen Goals ein.
6. Wende die bei der Unifikation erhaltene Substitution auf die neue Goalliste an.
7. Beweise die neue Menge von Zielen.

Alle Schritte außer der Unifikation (4), also insbesondere die Strategie zur Auswahl der Ziele (last-in-first-out) und zur Auswahl der Klauseln (bestimmt durch die Reihenfolge in der Datenbasis), stimmen mit PROLOG überein. Im Gegensatz zu PROLOG beruht ein Beweis in SASLOG nicht auf der syntaktischen Unifikation (Unifikation bzgl. der leeren Theorie, [Robinson 1965]), sondern auf einer semantischen Unifikation.

UNIFY($t_1$,$t_2$,$\sigma$) hat für zwei SASL-Termen $t_1$, $t_2$ und eine Substitution $\sigma$, die die Bindungsumgebung zum Zeitpunkt des Funktionsaufrufs darstellt, als Wert

- eine Substitution $\lambda$, wenn $t_1\lambda = t_2\lambda$
- "FAIL", falls $t_1\sigma$ und $t_2\sigma$ nicht unifizierbar sind
- $\perp$, falls die SASL-Funktion "=" für $t_1\sigma$ und $t_2\sigma$ nicht definiert ist, was zu einem Abbruch des Beweises führt.

UNIFY($t_1$,$t_2$,$\sigma$):

1. IF $\sigma$ = "FAIL" THEN RETURN "FAIL";
2. FOR i=1,2 DO $t_i$ := ULTIMATE-ASSOC($t_i$,$\sigma$);
   2.1 IF t1 Variable THEN RETURN $\sigma \cup \{(t_1/t_2)\}$;
   2.2 IF t2 Variable THEN RETURN $\sigma \cup \{(t_2/t_1)\}$;
   2.3 IF $t_1$ und $t_2$ in Normalform
       THEN IF $t_i = g(s_{i1},...,s_{ik})$, g Konstruktor, i=1,2
            THEN RETURN UNIFY($s_{1k}$,$s_{2k}$,UNIFY(...UNIFY($s_{11}$,$s_{21}$,$\sigma$)))
            ELSE RETURN "FAIL";
   2.4 IF $t_1$ in Normalform und $t_2$ nicht
       THEN RETURN UNIFY($t_1$,S-REDUCE(ULTIMATE-INST($t_2$,$\sigma'$)),$\sigma$);
   2.5 IF $t_2$ in Normalform und $t_1$ nicht
       THEN RETURN UNIFY(S-REDUCE(ULTIMATE-INST($t_1$,$\sigma'$)),$t_2$,$\sigma$);
   2.6 FOR i=1,2 DO $t_i$ := ULTIMATE-INST($t_i$,$\sigma'$);
       2.6.1 IF S-REDUCE($t_1 = t_2$) = TRUE THEN RETURN $\sigma$;
       2.6.2 IF S-REDUCE($t_1 = t_2$) = FALSE THEN RETURN "FAIL";
       2.6.3 ABORT WITH $\perp$

Dabei gilt

$\sigma'(x) = \sigma(x)$, falls x in Dom($\sigma$)
$\sigma'(x) = \perp$, sonst

für eine Variable x, d.h. freie Variablen bzgl. $\sigma$ werden zu $\perp$.

Die Funktion ULTIMATE-ASSOC instantiiert eine Variable bis sie auf einen Wert trifft, der keine Variable ist bzw. bis keine Substitution mehr möglich ist. Die Funktion ULTIMATE-INST dagegen instantiiert auch alle in einem Term vorkommenden Variablen. Die Funktion S-REDUCE aktiviert den Reduktionsalgorithmus. Die Fälle 1 bis 2.3 entsprechen dem normalen Robinson-Algorithmus. Nur wenn zwei Terme syntaktisch nicht unifizierbar sind, wird die SASL-Reduktion aufgerufen.

Zwei SASL-Terme sind genau dann unifizierbar, wenn sie zu gleichen Grundtermen reduzierbar sind. Vor der Reduktion werden alle in den SASL-Termen enthaltenen logischen Variablen durch ihre zu diesem Zeitpunkt gültige Bindung ersetzt. Freie Variablen bekommen den Wert $\perp$, so daß für nicht-strikte Funktionen die Reduktion trotzdem einen definierten Wert liefern kann. Zwei Grundterme sind genau dann gleich, wenn die Gleichheitsfunktion in SASL mit ihnen als Argumenten den Wert TRUE hat. Die Unifikation scheitert in genau den Fällen, in denen der Wert von "=" FALSE ist und bricht ab mit einem Fehler, wenn der Wert $\perp$ ist. Letzteres ist zum

Beispiel beim Vergleich zweier funktionaler Werte der Fall (siehe dazu auch [Nökel, Rehbold 1986]). Ein funktionaler Wert kann allerdings während der Unifikation an eine Variable gebunden und später auf andere Werte angewendet werden.

Beispiel: Das Prädikat all(f,l) ist erfüllt, wenn die Funktion f angewandt auf jedes Element der Liste l den Wert TRUE hat:

```
all(_?,[ ]).
all(_f,[_a|_l]) :- (_f _a) == TRUE, all(_f,_l).
```

Die Unifikation von Standard-PROLOG ist ein Spezialfall dieser semantischen Unifikation von SASLOG. Da Terme in reinen PROLOG-Programmen keine SASL-Ausdrücke sind, sind sie bzgl. der Funktionsdefinitionen irreduzibel, so daß sie nur auf syntaktische Unifizierbarkeit untersucht werden.

Es soll nun gezeigt werden, daß der durch diesen Algorithmus berechnete allgemeinste Unifikator eindeutig ist: Die semantische Unifikation ist ein Spezialfall der Unifikation über einer Gleichungstheorie (E-Unifikation), dargestellt als $=_E$. Eine Substitution $\sigma$ ist ein E-Unifikator für zwei Terme A und B, wenn $A\sigma =_E B\sigma$. Der allgemeinste Unifikator ist nicht notwendigerweise eindeutig. Nach [Subrahmanyam, You 1986] ist der allgemeinste Unifikator eindeutig (bis auf Äquivalenz bzgl. $=_E$), wenn folgende Bedingungen erfüllt sind:

1. Die funktionalen Terme in den zu unifizierenden Termen sind Grundterme.
2. Die Gleichungen, die die Reduktionsregeln definieren, sind konfluent.

zu 1: Die zu reduzierenden Terme in SASLOG sind Grundterme. Terme mit Variablen werden instantiiert, wobei freie Variablen zu $\perp$ werden.

zu 2: Es ist zu zeigen: Hat ein Term x zwei verschiedene Nachfolger $x_1$ und $x_2$, dann gibt es einen Term y, der Nachfolger sowohl von $x_1$ als auch von $x_2$ ist.

(i) Sind von den definierenden Gleichungen für eine Funktion f mehrere Regeln anwendbar, so ist die tatsächlich anzuwendende Regel durch die Reihenfolge der Definitionen eindeutig bestimmt. Bei der Anwendung einer Funktion in SASL kann es also keine zwei verschiedenen Nachfolger geben.

(ii) Durch die Normal-order-Auswertungsstrategie ist die Reihenfolge der Reduktion eindeutig bestimmt (der Reduktionsalgorithmus ist deterministisch), also kann es zu keinem Term - durch Reduktion verschiedener Unterterme - zwei verschiedene Nachfolger geben.

Aus (i) und (ii) folgt, daß die durch die Funktionsgleichungen definierten und den Reduktionsalgorithmus bestimmten Ableitungen konfluent sind.

Der allgemeinste Unifikator ist also eindeutig bis auf Äquivalenz bzgl. $=_E$.

Der semantische Unifikationsalgorithmus ist aber nicht vollständig. Die Anfrage

```
?- append [1 | _x] [3,4]  ==  append [1,2,3] [4]
```

ist mit der Funktionsdefinition

```
def  append [ ] l          =  l
     append [a | l1] l2    =  [a | (append l1 l2)]
```

nicht beweisbar, denn die freie Variable _x wird mit $\perp$ instantiiert. Da append im ersten Argument strikt ist, liefert die Reduktion von append [1 | _x] [3,4] den Wert $\perp$, und die Unifikation bricht mit einem Fehler ab.

Hier zeigt sich ein Unterschied zu dem Unifikationsalgorithmus von FUNLOG ([Subrahmanyam, You 1986]). Während SASLOG die Funktionsausdrücke bis zu ihrer Normalform reduziert, werden sie in FUNLOG unter der Kontrolle des Unifikationsalgorithmus Schritt für Schritt abgearbeitet. Obige Unifikation ist auch in FUNLOG nicht erfolgreich, aber der Unifikationsalgorithmus bricht nicht ab, sondern liefert "FAIL", weil Terme mit freien Variablen als irreduzibel interpretiert werden.

Das Ergebnis "FAIL" ist in obigem Beispiel sicherlich falsch, denn es bedeutet, daß zwei Terme nicht unifizierbar sind. Ein Abbruch der Unifikation heißt dagegen, daß der Unifikator in diesem System nicht berechnet werden kann (unabhängig davon, ob er tatsächlich existieren würde).

# 7 Einzelheiten der Implementierung

Die Ausgangsbasis für die Impementierung von SASLOG bilden der in [Nökel, Rehbold 1986] beschriebene SASL-Interpreter und das LISPLOG-System ([Boley 1986]), die in den Abschnitten 7.1.1 bzw. 7.2.1 vorgestellt werden.

Grob gesagt vollzieht sich die Berechnung in SASLOG in zwei Phasen. Zuerst werden funktionale Ausdrücke, also auch die Terme der PROLOG-Klauseln, durch Abstraktion in Kombinatorausdrücke übersetzt. Anschließend wird der Ausdruck ausgewertet. Handelt es sich um einen Funktionsausdruck, wird der Kombinatorausdruck zu seiner Normalform reduziert. Relationsausdrücke werden durch Resolution bewiesen.

Dieses Kapitel behandelt die Aspekte des Interpreters, die die Verbindung funktionaler und relationaler Programmierung betreffen. Abschnitt 7.1 behandelt die Abstraktion und Reduktion funktionaler Ausdrücke während Abschnitt 7.2 die Berechnung in der logischen Komponente erläutert.

## 7.1 Abstraktion und Reduktion

### 7.1.1 Kurzbeschreibung des SASL-Interpreters

Die Realisierung des SASL-Interpreters folgt einem Vorschlag von Turner ([Turner 1979]) für die Implementierung seiteneffektfreier funktionaler Programmiersprachen, die das Problem der Festlegung von Geltungsbereichen gebundener Variablen umgeht.

Gebundene Variablen treten in SASL auf als formale Parameter in Funktionsdefinitionen, in lokalen Definitionen von where-Ausdrücken und in Generatoren von ZF-Ausdrücken. Durch die Einführung einer endlichen Zahl neuer Konstanten (den Kombinatoren) kann man jeden SASL-Ausdruck in eine variablenfreie Notation übersetzen. Diesen Kompilationsprozeß bezeichnet man als Variablenabstraktion. Der entstandene Code ist eine binäre Struktur, die an das Laufzeitsystem übergeben wird. Das Laufzeitsystem transformiert diese Struktur durch Anwendung von Reduktionsregeln zu einer Normalform. Die Reduktionsregeln sind als Graphtransformationsregeln implementiert, d.h. bei einer Regelanwendung wird der reduzierte Ausdruck durch seinen Wert überschrieben. Durch Einführung von Superkombinatoren für häufig vorkommende Funktionen und Telescoping kann die Effizienz der Reduktionsmaschine verbessert werden.

In der LISP-Implementierung von SASL werden die Kombinatorgraphen als LISP-Strukturen repräsentiert. Um einen schnellen Zugriff auf die jeweils zu reduzierenden Elemente der Struktur zu haben, wird der vom Abstraktionsalgorithmus erzeugte Ausdruck auf allen Schachtelungsebenen umgedreht. Die anzuwendende Reduktionsregel hängt von dem Kombinator ab, der am exponierten Knoten des Kombinatorgraphen hängt. Der exponierte Knoten ist der letzte Knoten, den man findet, wenn man den Kombinatorgraphen in *CDR*-Richtung durchläuft. Bei diesem Durchlaufen wird das Anfangsstück in einem Stack aufbewahrt, der als Schnittstelle der Reduktionsfunktionen zum Kombinatorgraphen dient. Alle Reduktionsfunktionen orientieren sich an einem gemeinsamen Grundmuster: Zuerst werden so viele Elemente, wie der zu reduzierende Kombinator Argumente hat, vom Stack entfernt und ggf. reduziert. Wenn die Argumente den an sie gestellten Bedingungen entsprechen (z.B. müssen die Argumente arithmetischer Kombinatoren numerisch sein), wird die eigentliche Graphtransformation durchgeführt, indem die Wurzelzelle mit neuem Inhalt gefüllt wird, der dem Resultat der Regelanwendung entspricht. Die LISP-Zellen werden durch die LISP-Funktionen *RPLACA* und *RPLACD* manipuliert, die bestehende LISP-Strukturen verändern, statt neue anzulegen. Zu Einzelheiten der Implementierung siehe [Nökel, Rehbold 1986].

Die folgenden Abschitte behandeln die für die Verbindung zu PROLOG wichtigen Konzepte.

### 7.1.2 Abstraktion von Hornklauseln

Wie in Abschnitt 5.3 beschrieben, können Terme in der logischen Komponente reduzierbare Funktionsausdrücke sein. Unter reduzierbaren Ausdrücken sind solche Ausdrücke zu verstehen, die keine logischen Variablen, Konstanten oder Listen sind. Listen werden rekursiv nach reduzierbaren Ausdrücken durchsucht. Sobald eine Klausel in die Datenbasis eingefügt wird, werden die Funktionsausdrücke in den Prämissen in Kombinatorausdrücke übersetzt.

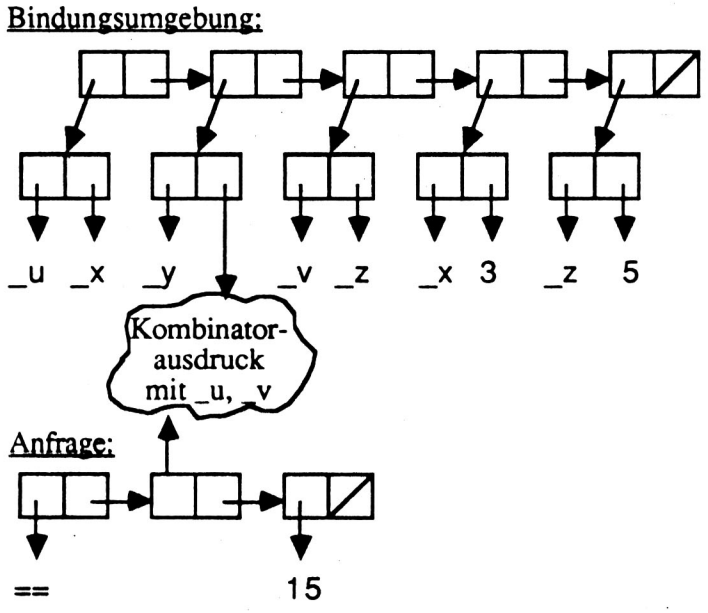
Die zu übersetzenden Ausdrücke können natürlich logische Variablen enthalten, da nur auf diese Weise Werte zwischen Literalen übergeben werden. Die destruktive Reduktionstechnik von SASL kann nicht direkt auf solchen Ausdrücken arbeiten, da sich die Bindungen logischer Variablen durch Backtracking ändern können. Eine physikalische Ersetzung dieser Variablen durch ihren Wert würde aber die Reduktion des Ausdrucks mit einer neuen Bindung verhindern.

Beispiel:

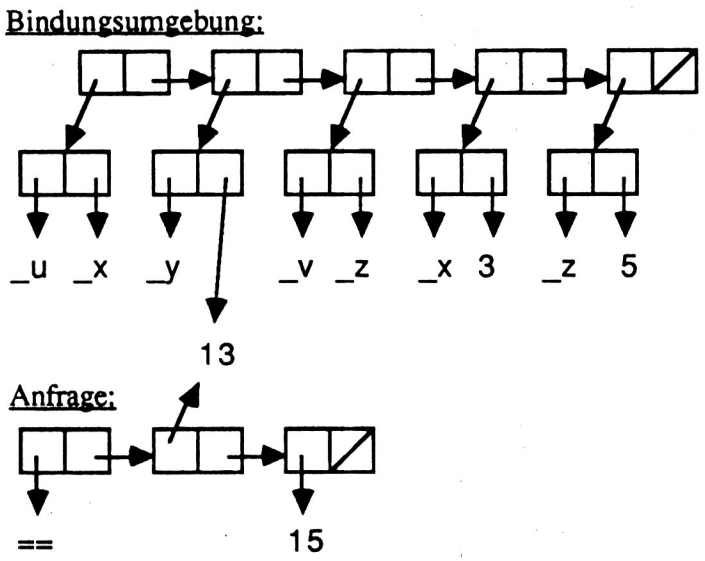
Datenbasis: p(_x,_y,_z) :- r(_x,_z), _y == 15.
r(3,5).
r(4,6).

Anfrage: ?- p(_u,_u+(_v+5),_v).

Die Unifikation der Anfrage mit der Konklusion ergibt die Substitution $\sigma$ = {(_u/_x), (_y/_u+(_v+5)), (_v/_z)}. Man beachte, daß _u+(_v+5) als Kombinatorausdruck gegeben ist. Der Beweis von r(_x,_z) ergibt die neue Substitution $\sigma' = \sigma \cup$ {(_x/3), (_z/5)}. Zum Beweis von _y == 15 muß _u+(_v+5), die Bindung von _y, reduziert werden. Vor diesem Reduktionsschritt ist die interne Situation, vereinfacht dargestellt, wie folgt:




Die Reduktion würde den Kombinatorgraph verändern. _u würde physikalisch durch seinen Wert 3, _v durch 5 ersetzt und der Ausdruck zu 13 reduziert werden und man erhielte:




Da die Unifikation scheitert, ist Backtracking nötig. Zwar werden neue Bindungen für _x und _z und somit auch für _u und _v gefunden, der Wert von _y ist aber nach der Reduktion konstant, was offensichtlich falsch ist.

Eine naive Lösung für dieses Problem wäre es, vor der Reduktion den Ausdruck zu kopieren. Dies könnte sogar mit der Instantiierung verknüpft werden. Aber Kombinatorausdrücke können aufgrund der Reduktion des Y-Kombinators zyklisch sein (zur Reduktion von Y siehe [Nökel, Rehbold 1986]). Das Kopieren und Instantiieren eines zyklischen Graphen ist aber eine sehr aufwendige Operation.

Um einen korrekten Beweis zu gewährleisten ohne den Kopieraufwand zur Laufzeit in Kauf zu nehmen, werden Ausdrücke mit logischen Variablen in eine spezielle Form übersetzt: Für jede Klausel werden die globalen logischen Variablen gesammelt, das sind diejenigen logischen Variablen, die außerhalb eingebetteter prove- oder ZF-Ausdrücke vorkommen. Bindungen von Variablen, die nur innerhalb von prove- und ZF-Ausdrücken vorkommen, sind außerhalb nicht sichtbar und deshalb unproblematisch. Bei der Übersetzung eines Funktionsausdrucks E werden die darin vorkommenden globalen Variablen in einer Liste $[\_V_1,\ldots,\_V_n]$ zusammengefaßt und E wie gewohnt in einen Kombinatorausdruck COMB übersetzt. Ist $n = 0$, d.h. E enthält keine globale Variable, ist die Übersetzung abgeschlossen. Ansonsten werden die logischen Variablen $\_V_1,\ldots,\_V_n$ aus COMB abstrahiert und man erhält den von globalen Variablen freien Kombinatorausdruck COMB'. Der Ausdruck E in der Klausel wird schließlich ersetzt durch die Form

$$(\text{graph COMB}' [\_V_1,\_V_2,\ldots,\_V_n]).$$

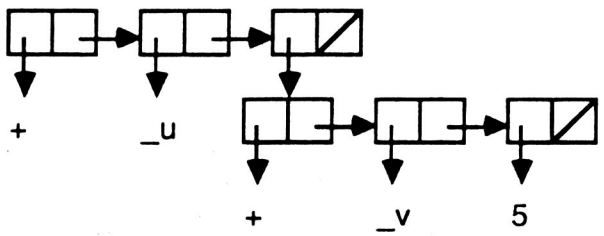
Bei der Übergabe des Ausdrucks an den Reduktionsalgorithmus wird im Verlauf der Instantiierung der Ausdruck

$$(\text{COMB}' V_1' \ldots V_n')$$

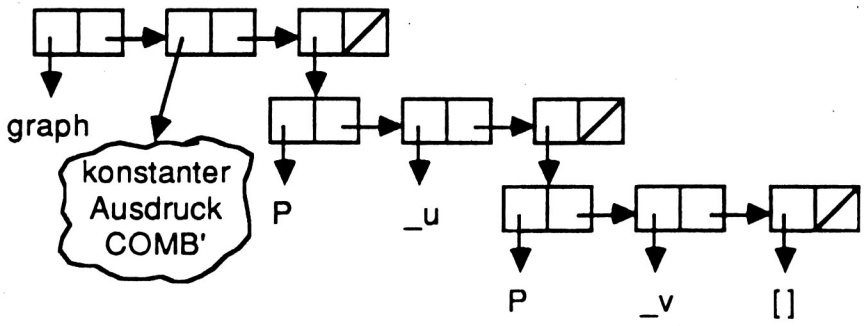
aufgebaut (dies wird im folgenden als Auflösung des graph-Ausdrucks bezeichnet). Die $V_i'$ sind die vollständig instantiierten Werte der Variablen $\_V_i$. Freie Variablen werden an $\perp$ gebunden. COMB' wird auf die $V_i'$ angewandt, die Abstraktion wird also rückgängig gemacht. Das Gesetz der Abstraktion besagt nämlich, daß die Abstraktion die Umkehrung der Applikation ist ([Turner 1979]).

Durch dieses Verfahren wird das Kopieren und Instantiieren beliebig komplexer Kombinatorausdrücke vermieden, weil alle Kombinatorausdrücke, die nicht in das graph-Konstrukt eingebunden sind, als konstant vorausgesetzt werden können. Gleichzeitig wird aber auch die Zerstörung der Bindungsumgebung verhindert, da die reduzierbaren Ausdrücke frei von Variablen sind und die Reduktion nicht direkt auf den Einträgen der Bindungsumgebung arbeitet.

Die Anfrage ?-p(_u,_u+(_v+5),_v) des obigen Beispiels ist ein Literal mit einem einzigen SASL-Ausdruck _u+(_v+5). Die interne Repräsentation ist:



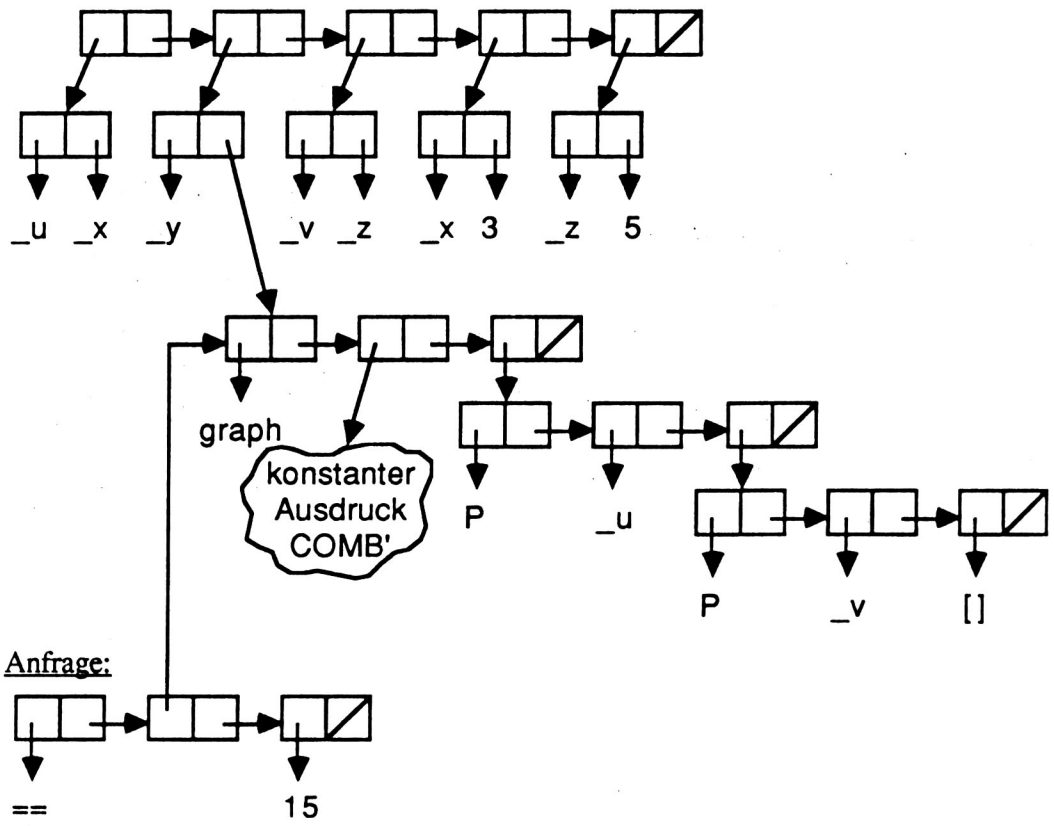

Durch die Abstraktion der logischen Variablen erhälten man den Kombinatorausdruck COMB' = [_u][_v](_u+(_v+5)) = (C (B' +) (C + 5)) und das Ergebnis der Übersetzung ist:




(der Kombinator P ist der interne Listenkonstruktor).

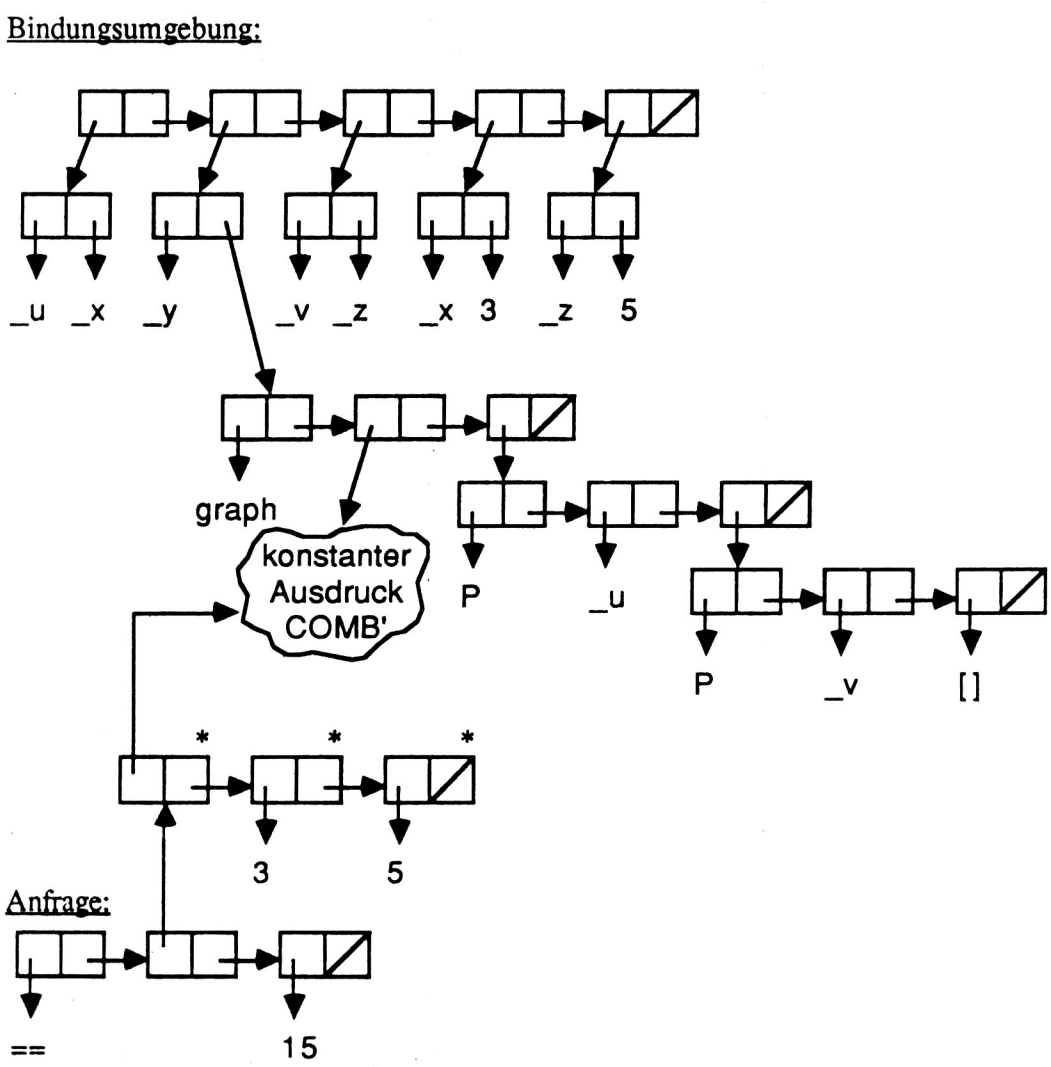
Wenn jetzt die Stelle erreicht wird, an der _y == 15 zum ersten Mal bewiesen werden muß, haben Bindungsumgebung und Goal folgende Form:

Bindungsumgebung:




Anfrage:

Die Instantiierung resultiert in folgender Situation:




Die Wolke, die den konstanten Teil des Ausdrucks _u+(_v+5) repräsentiert, braucht also nicht kopiert zu werden. Während der Reduktion können nur konstante Unterterme von COMB' geändert werden, die Funktionalität des Kombinatorausdrucks bleibt erhalten. Man sieht sofort, daß das Instantiierungsmuster für _y korrekt bleibt, wenn sich die Bindungen von _x und _z durch Backtracking ändern. Das Goal ist natürlich eine durch die notwendige Instantiierung erzeugte Kopie. Die mit * markierten Zellen werden beim nächsten Beweisversuch mit anderen Werten für _u und _v neu angelegt, so daß der Beweis erfolgreich abgeschlossen werden kann.

### 7.1.3 Abstraktion von prove- und ZF-Ausdrücken

Die Einführung der Konstanten als neuen Typ erforderte einige kleine Änderungen im Abstraktionsalgorithmus. Wenn eine Variable aus einem Ausdruck abstrahiert wird, der auch Konstanten des gleichen Namens enthält, dürfen diese natürlich nicht wegabstrahiert werden. Ebensowenig dürfen logische Variable gleichen Namens abstrahiert werden. Dies wird durch zusätzliche Tests im Abstraktionsprozeß sichergestellt.

Die Funktion *ELIM-ZF*, die einen Ausdruck mit ZF-Ausdrücken in einen gleichwertigen ZF-freien Ausdruck umwandelt, wurde so erweitert, daß sie PROLOG-Generatoren und auch prove-Ausdrücke behandeln kann.

Bei Ausdrücken, die nicht explizit geklammert sind, wird vom Abstraktionsalgorithmus implizit Linksklammerung angenommen. Da der Interpreter auf LISP-Strukturen arbeitet, hat ein prove-Ausdruck die Form (prove $g_1$ $g_2$... $g_n$), was als (..((prove $g_1$) $g_2$)..$g_n$ ) angesehen wird, also als Anwendung der Funktion (prove $g_1$) auf $g_2$, deren Wert auf $g_3$ angewandt wird usw.. Die Argumente von prove sind aber als Konjunktion von Literalen zu interpretieren. Deshalb werden die Literale nach der Abstraktion darin enthaltener SASL-Terme zu einer Liste zusammengefaßt, die dann das einzige Argument von prove wird. Ein Ausdruck

prove ($g_1$,...,$g_n$)

wird also umgewandelt in den Ausdruck

prove [$g_1$',$g_2$',...,$g_n$'].

Der Kombinator goal, der für PROLOG-Generatoren im ZF-Ausdruck erzeugt wird, bekommt neben dem prolog-goal (das wie bei prove zu einer Liste transformiert wurde) als zweites Argument die Liste der Resultatsvariablen. Diese Liste erhält man aus der Liste der Generatorvariablen, die man zu logischen Variablen mit gleichen Namen umwandelt ( x => _x). Man erhält also einen Kombinatorausdruck

goal [$g_1$',$g_2$',...,$g_n$'] [_$V_{i1}$,...,_$V_{im}$],

der (wie in Abschnitt 6.1.2 beschrieben) in den ZF-freien Ausdruck integriert wird.

## 7.1.4 Die neuen Reduktionsregeln

Die Reduktionsfunktionen für die in Kapitel 6.1 erläuterten neuen Kombinatoren isconstant, cpp, joinp, prove, goal und next orientieren sich an dem oben beschriebenen Grundmuster. Dieser Abschnitt geht deshalb nur auf die Besonderheiten bei der Verbindung zu PROLOG näher ein.

### Die Reduktion von prove

Die Reduktionsregel für prove lautet:

| | | |
|---|---|---|
| prove $[g_1,g_2,...,g_n]$ | = TRUE, | falls der PROLOG-Interpreter mit der Anfrage ?-$g_1,g_2,...,g_n$ eine Lösung berechnet |
| prove $[g_1,g_2,...,g_n]$ | = FALSE, | falls der Beweis von ?-$g_1,g_2,...,g_n$ in endlicher Zeit scheitert |
| prove $[g_1,g_2,...,g_n]$ | = $\perp$, | sonst |

Die PROLOG-Anfrage ist zum Zeitpunkt der Reduktion von prove ein Kombinatorausdruck, der die Liste von Literalen darstellt. Die Literale selbst haben durch die Abstraktion evtl. darin enthaltener Variablen ebenfalls nicht mehr ihre ursprüngliche Form. Aus dem Kombinatorausdruck wird wieder eine Folge von Literalen erzeugt, mit denen der PROLOG-Interpreter aufgerufen wird (s.u.). Bei erfolgreichem Beweis liefert die Beweisfunktion als Wert die Continuation (ungleich *NIL*), die den Zustand zum Zeitpunkt des Verlassens des PROLOG-Systems darstellt (zum Aufbau der Continuation siehe Abschnitt 7.2.2). In diesem Fall ist der Wert von prove $[g_1,g_2,...,g_n]$ gleich TRUE. Scheitert der Beweis, ist die Continuation *NIL* und das Ergebnis der Reduktion von prove ist FALSE.

### Die Reduktion von goal

Die Reduktionsregel für goal lautet:

goal $[g_1,g_2,...,g_n]$ $[\_V_1,...,\_V_m]$
= $[[V_1',...,V_m'] \mid$ (next continuation $[\_V_1,...,\_V_m])]$,
falls der PROLOG-Interpreter für ?- $g_1,...,g_n$ eine Lösung berechnet

goal $[g_1,g_2,...,g_n]$ $[\_V_1,...,\_V_m]$ = [],
falls der Beweis von ?- $g_1,...,g_n$ in endlicher Zeit scheitert

goal $[g_1,g_2,...,g_n]$ $[\_V_1,...,\_V_m]$ = $\perp$, sonst

Dabei sind die $V_i'$ die berechneten Bindungen der Variablen $\_V_i$.

Das erste Argument von goal ist ein Kombinatorausdruck, der die Liste der Literale darstellt. Wie bei der Reduktion von prove wird daraus eine Folge von Literalen erzeugt, mit denen nun der PROLOG-Interpreter gestartet wird (s.u.). Die Funktion goal hat als zweites Argument die Liste der Generatorvariablen. Diese werden in der durch den Beweis ermittelten Bindungsumgebung (das dritte Element der Continuation, siehe Abschnitt 7.2.2) instantiiert. Die Liste der instantiierten Variablen ist das erste Element der durch den Kombinator goal berechneten Liste von Lösungen. Die Liste der restlichen Lösungen wird bei Bedarf durch die Reduktion des Kombinators next bestimmt. Bei erfolglosem Beweis ist die Continuation *NIL*, d.h. die Reduktion liefert die leere Liste.

**Die Reduktion von next**

Die Reduktionsregel für next lautet:

next continuation $[\_V_1,...,\_V_m]$
= $[\,[V_1',...,V_m']$ | (next continuation' $[\_V_1,...,\_V_m])\,]$,
falls der PROLOG-Beweiser beginnend im Zustand continuation eine Lösung berechnet
next continuation $[\_V_1,...,\_V_m] = [\,]$,
falls der Beweis in endlicher Zeit abbricht
next continuation $[\_V_1,...,\_V_m] = \perp$, sonst

Dabei sind die $V_i'$ die berechneten Bindungen der Variablen $\_V_i$ und continuation' ist die neue Continuation.

Der Kombinator next hat zwei Argumente: Die Continuation, die den Zustand des PROLOG-Interpreters nach der Berechnung der letzten Lösung bestimmt, und den Resultatsterm. Bei der Reduktion von next wird der PROLOG-Interpreter mit der Continuation als Argument aufgerufen. Ist der Wert des Aufrufs *NIL*, so gibt es keine weiteren Lösungen mehr und das Reduktionsergebnis ist [ ]. Bei erfolgreichem Beweis ist wie bei der Reduktion von goal die Liste der instantiierten Variablen das erste Element der Liste der noch möglichen Lösungen. Die verbleibenden Lösungen werden bei Bedarf durch den Kombinator next bestimmt, der als Argument die neue Continuation und wieder die Liste der Variablen bekommt.

## Übergabe der Anfrage an den Resolutionsbeweiser

Die PROLOG-Anfrage, die Argument von prove oder goal ist, wurde bei der Abstraktion zu einer Liste von Literalen zusammengefaßt. Aus dieser Liste muß vor dem Beweis wieder die vom PROLOG-Interpreter erwartete Folge von Goals erzeugt werden. Da aus der Anfrage Variablen abstrahiert worden sein können, wird zur Erzeugung der Folge von Literalen der Reduktionsalgorithmus auf jedes Element der die Anfrage darstellenden Liste angewandt.

An dieser Stelle ist von Bedeutung, daß es für kein Literal $q(x_1,...,x_n)$ eine Funktion des gleichen Namens q gibt, da sonst $(q\ x_1\ ...\ x_n)$ als Kombination reduziert würde. Wenn man diese Einschränkung fallen lassen möchte, könnte man eine Reduktion von q auch verhindern, indem man als zusätzliches Argument für prove und goal die Liste der Relationsnamen des Goals angibt. Die Funktion *REDEXPO* bricht dann deren Reduktion ab. Dieses zusätzliche Argument bedeutet aber neben einem Mehraufwand an Speicherplatz auch einen Effizienzverlust, weil bei jedem Aufruf von *REDEXPO* ein solcher Test notwendig wäre. Außerdem ist ein Programm verständlicher, wenn die Mengen der Bezeichner für Relationen und Funktionen disjunkt sind.

Nun ist noch ein Problem zu beachten: Aus den Argumenten $x_i$ der Literale können SASL-Variablen abstrahiert worden sein. Man muß nun verhindern, daß beim Aufruf des Beweisers evtl. vorhandene graph-Ausdrücke in Kombinatorausdrücke eingebettet sind. Hierdurch wäre eine korrekte Variableninstantiierung bei der Rückkehr nach SASL nicht mehr gewährleistet.

Man kann dies an folgendem, einfachen Beispiel sehen:

```
[x; y <- [1 2 3]; [x] <- q(_z), _x == y + _z]
```

Das Goal _x == y + _z enthält einen reduzierbaren Ausdruck mit einer logischen Variablen, der bei der Abstraktion in den Ausdruck (graph (y +) (P _z [ ])) umgewandelt wird. Aus diesem Ausdruck wird die Variable y abstrahiert. Der graph-Ausdruck steckt nun innerhalb des Kombinatorausdrucks (C (B graph +) (P _z [ ])). Beim Beweis der Anfrage wird _x an den nicht-reduzierten Ausdruck gebunden, der aber beim Verlassen des PROLOG-Systems nicht instantiiert werden kann, weil es sich nicht um einen graph-Ausdruck handelt.

Das Problem läßt sich dadurch lösen, daß man die Abstraktion der Variablen aus graph-Ausdrücken vor dem Beweis der Anfrage durch Reduktion wieder rückgängig macht. Wie geht man nun dabei vor ohne unnötig viele Reduktionsschritte durchzuführen? Die Variablenabstraktion erfolgt in den angegebenen Fällen

ausschließlich durch Verwendung der Grundkombinatoren S, K, I und den Vereinfachungen mit Hilfe der Kombinatoren B, B', C, C', S und S'. Will man die abstrahierten Variablenwerte also an ihre ursprüngliche Stelle bringen und somit verhindern, daß graph-Ausdrücke in Kombinatoraudrücken versteckt sind, braucht man nur den Reduktionsalgorithmus so lange anzuwenden, wie der linke Sohn des exponierten Knotens einer der oben angegebenen Kombinatoren ist (Gesetz der Abstraktion, siehe [Turner 1979]). Auf diese Weise werden die graph-Ausdrücke direkt an PROLOG übergeben und können bei der Rückgabe auch instantiiert werden, da sie nicht von anderen Ausdrücken umschlossen sind.

## Variableninstantiierung

Wie oben erwähnt, werden die logischen Variablen des Resultatsterms bei der Reduktion von goal und next direkt nachdem die Bindungsumgebung bekannt ist instantiiert. Besser mit der Lazy Evaluation-Strategie von SASL vereinbar wäre es, die Instantiierung so lange zu verschieben, bis der Wert wirklich benötigt wird. Dieser Aufschub erfordert, daß mit jeder Variablen die gesamte Bindungsumgebung mitgeschleppt würde. Das wäre aber sehr aufwendig, vor allem, wenn Lösungen verschiedener PROLOG-Aufrufe in einem Ausdruck vorkämen. In diesem speziellen Fall gibt man zwar Laziness auf, spart aber auf der anderen Seite Speicherplatz. Es handelt sich also um einen Kompromiß zwischen Speicher und Laufzeit. Die Situation, daß auf Variablenwerte später nie zugegriffen wird, dürfte aber nicht sehr häufig auftreten. Außerdem konnte durch Einführung von graph-Ausdrücken (die das Durchsuchen beliebiger Kombinatorausdrücke verhindern) der Aufwand für die Instantiierung erheblich gesenkt werden.

Auf der anderen Seite hat man aber auch einen Effizienzvorteil, weil durch die Ausnutzung von Structure Sharing bei der Instantiierung (Abschnitt 7.2.4) die mehrfache Darstellung gleicher Ausdrücke vermieden wird. Durch die call-by-need-Auswertungstrategie wird dieser Ausdruck also auch nur einmal berechnet. Das ist aber nur möglich, wenn der ganze Term in einem Schritt instantiiert wird.

## Änderung der Datenbasis

Bei der Reduktion von goal und next wird jeweils nur eine Lösung berechnet und die weitere Berechnung verzögert. Dadurch können mehrere Beweise gleichzeitig aktiv sein. Hierbei ist es wichtig, daß die Datenbasis, die für alle PROLOG-Aufrufe global ist, nicht dynamisch durch eine Anfrage geändert wird. Dadurch könnten sich die Lösungen anderer Beweise ändern und vom Zeitpunkt des Beweises abhängen. Die

Ergebnisse wären also nicht reproduzierbar, was der Referential Transparency widerspricht.

Würde man Datenbasisänderungen erlauben, müßte man jedem aktiven PROLOG-Prozeß, der dies ausnutzt, eine eigene lokale Datenbasis zuordnen, die auch bei seiner Suspendierung in der Continuation enthalten wäre. Der Speicheraufwand dafür wäre unverhältnismäßig hoch. Außerdem müßte man festlegen, welche der verschiedenen Datenbasen für einen neuen PROLOG-Aufruf zur aktuellen Datenbasis würde.

Um diesen Schwierigkeiten aus dem Weg zu gehen, und weil PROLOG ja auch kein Verwaltungssystem für diese nicht-monotonen Operationen vorsieht, sind die Prädikate assert und retract nicht definiert.

## 7.2 Das Beweisverfahren

### 7.2.1 Eigenschaften des LISPLOG-Interpreters

Der iterative LISPLOG-Interpreter bildet die Basis für die logische Komponente von SASLOG. Er wurde aus dem in [Boley, Kammermeier u.a 1985] beschriebenen rekursiven Interpreter entwickelt. Er verwaltet die noch unbewiesenen Teilziele und die offenen Backtrackmöglichkeiten in zwei verschiedenen Stacks. Für die Bindungsumgebung wird eine zweidimensionale Array-Struktur benutzt. Beim Einlesen haben die Variablen die Form einer zweielementigen Liste, bestehend aus dem Fragezeichen und dem Variablennamen, z.B. (? x). Während des Beweises wird diese Liste um zwei ganzzahlige Werte ergänzt, die als Index für den Zugriff auf die Bindungsumgebung dienen, z.B. (? x 3 64). Diese Umbenennung der Variablen der Konklusion erfolgt im Laufe der Unifikation mit dem Goal, während die Umbenennung der Variablen im Klauselrumpf bis zum Beweis der jeweiligen Prämisse verzögert wird. Gleichzeitig mit der Umbennung werden die Variablen eines Ziels so weit wie möglich instantiiert. Einzelheiten hierzu findet man bei [Dahmen 1986].

## 7.2.2 Die Hauptfunktion des PROLOG-Interpreters

Die Haupfunktion des PROLOG-Interpreters *FIND-NEXT-PROVE* berechnet bei jedem Aufruf jeweils eine Lösung für eine Anfrage. Der Wert ist

- $\perp$, falls ein Fehler auftrat,
- *NIL*, falls der Beweis fehlschlägt,
- die Continuation bei erfolgreichem Beweis.

Die Continuation bestimmt den Status des Systems nach Abschluß des Beweises. Mit dieser als Argument kann die Funktion *FIND-NEXT-PROVE* beim nächsten Aufruf den alten Zustand restaurieren. Der Aufbau der Continuation hat sich gegenüber LISPLOG geändert. Sie besteht aus drei Komponenten:

- dem Schlüsselwort *next*, das beim nächsten Aufruf des Interpreters die Herstellung des alten Zustandes veranlaßt und Backtracking anstößt
- dem Choice-stack mit den offenen Backtrackmöglichkeiten
- der Bindungsumgebung

Der Resultatsterm ist in der Continuation nicht mehr enthalten. Die Instantiierung des Resultatsterms wird in der aufrufenden Funktion durchgeführt, weil sie je nach Art des Aufrufs verschieden ist:

- In den Reduktionsergebnissen von goal und next werden freie Variablen zu $\perp$ instantiiert.
- Bei PROLOG-Durchgriffen während der Reduktion von prove ist eine Instantiierung unnötig, da nur Erfolg oder Mißerfolg von Bedeutung ist.
- Bei Aufrufen aus LISP müssen SASL-Ausdrücke, die an Variable gebunden sind, vollständig reduziert werden (s.u.).
- Bei interaktiven Anfragen wird die Reduktion von Ausdrücken durch die Ausgabekomponente gesteuert.

Die Funktion *FIND-NEXT-PROVE* nimmt jeweils das erste der noch zu beweisenden Ziele vom Stack, benennt die Variablen durch Einfügen des Index um und führt dann die notwendigen Beweisschritte durch. Beim Beweis benutzerdefinierter Prädikate werden alle die Klauseln gesammelt, deren Konklusion mit dem Goal unifizierbar sein könnte (siehe Abschnitt 7.3). Bei erfolgreicher Unifikation des Goals mit der Konklusion einer Klausel werden die Prämissen dieser Klausel zu den noch zu beweisenden Zielen hinzugefügt. Die restlichen Klauseln werden als Backtrackmöglichkeiten auf den Choice-Stack gelegt.

Die Variableninstantiierung für Goals wird in zwei Phasen durchgeführt. Vor dem eigentlichen Beweis werden Variablen - wenn sie mit ihrem Index versehen werden - auch gleichzeitig soweit wie möglich instantiiert. Dabei werden die Variablen des zweiten Arguments von graph zwar instantiiert, die Auflösung des graph-Ausdrucks (z.B. (graph COMB [_u,_v])) in einen reduzierbaren Kombinatorausdruck (im Beispiel: (COMB u' v')) und die gleichzeitige Instantiierung freier Variablen zu $\perp$ erfolgt aber erst unmittelbar vor der Reduktion, weil während der Unifikation noch weitere Variablenbindungen erzeugt werden. Durch diese Zweiteilung sollen Mehrfachinstantiierungen der gleichen Variable vermieden werden, wenn die Unifikation mit mehreren Konklusionen versucht wird.

Im Unterschied zu LISPLOG werden die Variablen der Konklusion schon vor der Unifikation umbenannt, so daß der Unifikationsalgorithmus übersichtlicher wird.

### 7.2.3 Die semantische Unifikation

Die in PROLOG übliche und auch in LISPLOG durchgeführte syntaktische Unifikation wird in SASLOG durch die semantische Unifikation ersetzt. Die Unifikation von Literalen (Konklusion und Ziel) und Termen wird in zwei verschiedenen Funktionen durchgeführt.

Die Argumente eines Literals sind Terme. Bei der Unifikation von Konklusion und Goal werden zuerst die Prädikatnamen verglichen und dann die Unifikationsfunktion für Terme nacheinder für die Argumente aufgerufen. Für zwei Literale $p(t_1,...,t_n)$ und $p(s_1,...,s_n)$ sowie eine Bindungsumgebung $\sigma$ sieht das also folgendermaßen aus:

$$UNIFY(s_n,t_n,UNIFY(s_{n-1},t_{n-1},UNIFY(...,UNIFY(s_1,t_1,\sigma)..)$$

Die Funktion *UNIFY* realisiert genau die semantische Unifikationsprozedur für Terme wie sie in Abschnitt 6.2 zur Beschreibung der Semantik angegeben wurde.

Der einzige Konstruktor in SASLOG ist der Listenkonstruktor P mit 2 Argumenten (Anfang und Rest einer Liste). Sind beide zu unifizierenden Terme Listen, dann ruft sich die Funktion rekursiv mit den Argumenten wieder auf. Ist ein Term $t_i$ eine Liste und der andere Term $s_i$ nicht, dann ist eine Unifikation nur möglich, wenn der Term $s_i$ zu einer Liste reduziert werden kann. Ist der Wert von $s_i$ keine Liste, scheitert die Unifikation sofort. Nur wenn der Wert von $s_i$ eine Liste ist, kann der Unifikationsalgorithmus für die Argumente aufgerufen werden.

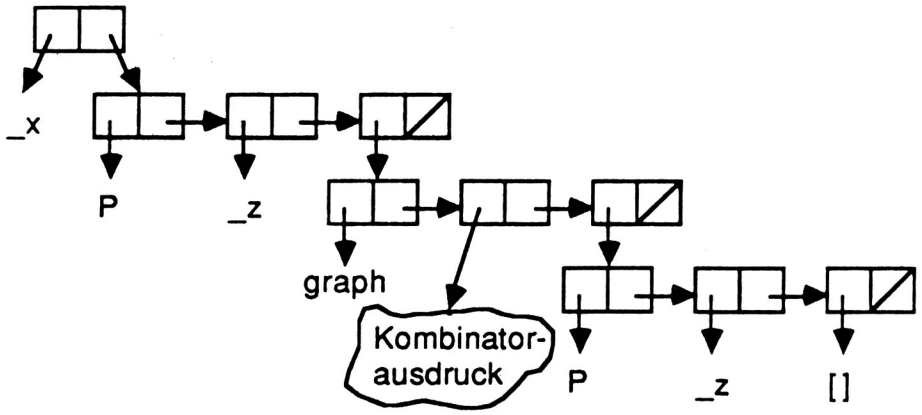
Konstanten, Zahlen und boolesche Werte werden direkt mit der SASL-Funktion "=" verglichen.

Wie schon mehrfach erwähnt, müssen die zu reduzierenden Terme variablenfrei sein. Deswegen werden freie Variablen mit ⊥ instantiiert. Wie die Instantiierungsfunktion das Structure Sharing - das für die Reduktion von Kombinatorgraphen eine wichtige Rolle spielt - unterstützt, wird im folgenden Abschnitt beschrieben.

### 7.2.4 Variableninstantiierung

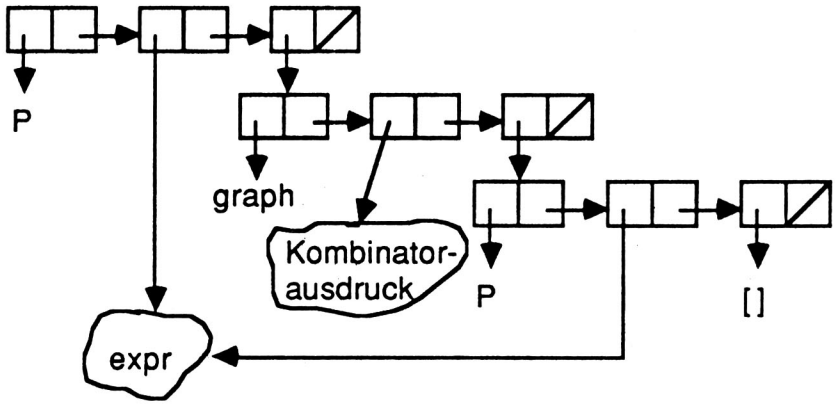
Der SASL-Reduktionsprozeß arbeitet auf einem Kombinatorgraphen, der durch Ausnutzung von Structure Sharing gleiche Ausdrücke nur einmal enthält, so daß diese auch nur einmal reduziert werden müssen. Beim Übergang von PROLOG nach SASL (d.h. bei der Reduktion von Termen während der Unifikation und beim Verlassen des PROLOG-Interpreters nach erolgreichem Beweis während der Reduktion von goal und next) sollte diese Structure Sharing-Eigenschaft durch die Variableninstantiierung unterstützt werden.

Angenommen, eine Variable _x hat die Bindung [_z | (graph ... [_z])]:




Die seperate Instantiierung beider Vorkommen von _z würde nicht nur den Aufwand verdoppeln (zumal wenn _z ein komplizierter, verschachtelter Ausdruck ist), sondern _z würde durch zwei Kopien des gleichen Ausdrucks ersetzt. Die Structure Sharing Eigenschaft wäre verloren. Als Lösung dieser Situation wird beim Aufruf der Instantiierungsfunktion *ULTIMATE-INST* eine Assoziationsliste **VAR-VALUES** angelegt. Sobald eine Variable instantiiert wurde, wird das Paar bestehend aus der Variablen und ihrem Wert in diese Liste eingefügt. Bevor nun für eine Variable in der Bindungsumgebung nach ihrem Wert gesucht wird, wird in dieser lokalen Liste **VAR-VALUES** nachgeschaut, ob die Variable während des gerade laufenden Prozesses schon instantiiert wurde. War dies der Fall, hat die Variable genau den in **VAR-VALUES** gefundenen Wert.

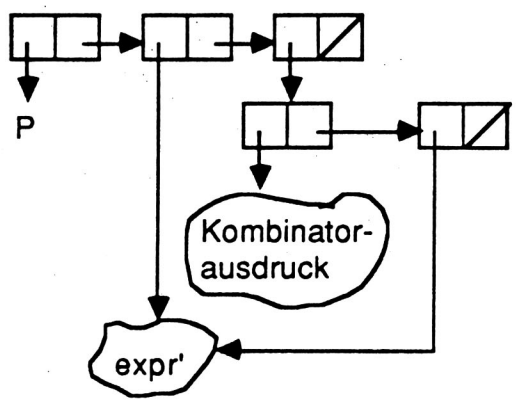
Angenommen, die Bindung von _z sei ein Ausdruck expr. Dann enthält die Liste *VAR-VALUES* nach der Instantiierung des ersten Vorkommens von _z das Paar (_z expr). Die Instantiierung von _x ergibt also:




Man beachte, daß die einzelnen Zellen bei der Instantiierung neu angelegt werden, während der Kombinatorausdruck noch der ursprüngliche ist.

Wie in Abschnitt 7.2.2 erläutert, werden Anfragen bei der Indizierung ihrer Variablen schon vor der Unifikation so weit wie möglich instantiiert. Dadurch können Terme die Structure Sharing-Eigenschaft erfüllen, indem gleiche Unterterme nur einmal angelegt wurden. Um diese Eigenschaft auch weiterhin zu erhalten, wird gleichzeitig mit *VAR-VALUES* eine zweite Liste *TERM-VALUES* angelegt. Wurde nun ein Term instantiiert, wird ein Paar mit je einem Zeiger auf den nicht-instantiierten und den instantiierten Term in diese Liste *TERM-VALUES* eingefügt, so daß der gleiche Unterterm nicht noch einmal für die Instantiierung durchlaufen werden muß.

Soll nun der obige Ausdruck (der Wert von _x) vollständig instantiiert werden, so enthält nach der ersten Instantiierung von expr zu expr' die lokale Assoziationsliste *TERM-VALUES* u.a. das Paar (expr expr'). Die Instantiierung des zweiten Vorkommens des Ausdrucks expr ergibt denselben Wert expr'; der graph-Ausdruck wird aufgelöst:



## 7.3 Organisation der Datenbasis

Für die Bearbeitung durch den Interpreter werden funktionale Terme in Klauseln durch Elimination von where- und ZF-Ausdrücken in Kombinatorausdrücke übersetzt (Abschnitt 7.1). Um dem Benutzer die Möglichkeit zu geben, sich die Datenbasis interaktiv anzusehen (durch das bekannte listing-Kommando), wird für jede Klausel neben der übersetzten Version auch die Originalform abgespeichert. Auch die Definitionen von SASL-Funktionen werden in ihrer Originalversion und in ihrer compilierten Version (als Kombinatorgraphen) abgelegt.

### Indexierung von PROLOG-Klauseln

Die Indexierung von Klauseln hat das Ziel, eine Vorauswahl der Klauseln zu treffen, die zum Beweis eines Goals herangezogen werden. Kriterium ist die potentielle Unifizierbarkeit der Konklusion einer Klausel mit der Anfrage. Dabei soll die Zahl der ausgewählten Klauseln möglichst gering sein, ohne aber eine unifizierbare Klausel auszuschließen.

Das Indexierungskonzept für LISPLOG-Datenbasen ([Bernardi 1986]) wurde für SASLOG nicht übernommen. In LISPLOG erfolgt die Indexierung nach dem Prädikatnamen und einer vom Programmierer festzulegenden Koordinate. Die Koordinatenwerte werden in vier Typen (*konstant-atomar*, *Variable*, *Term*, *undefiniert*) gegliedert und festgelegt, welche untereinander potentiell unifizierbar sind. Eine konstant-atomare Koordinate ist z.B. mit einer atomaren Konstante gleichen Wertes unifizierbar, mit einer Variablen möglicherweise und mit einem Term nie unifizierbar.

Hier ergibt sich der erste Unterschied zu SASLOG. In SASLOG ist eine konstant-atomare Koordinate auch mit einem Term potentiell unifizierbar. Dies ist nämlich der Fall, wenn es sich bei dem Term um einen Kombinatorausdruck handelt. Das bedeutet, daß das Indexierungsverfahren zu jeder Anfrage mit einer Konstanten an der Indexkoordinate auch alle Klauseln mit Termen an dieser Position auswählen muß und umgekehrt. Dies würde die Effektivität des Indexverfahrens in Frage stellen. Man kann das Verfahren verbessern, indem man den Typ *Term* in die Typen *Liste* und *Ausdruck* aufteilt, denn Listen und Konstanten sind nicht unifizierbar. Hierbei sind jedoch noch einige Fragen zu klären: Wie soll das so geänderte Indexierungsverfahren reagieren, wenn ein Ausdruck an der Indexkoordinate zu einer Konstante reduziert wurde? Soll die Organisation der Datenbasis dadurch geändert werden?

In LISPLOG gibt es die Möglichkeit, ein Prädikat zu definieren, das mit unterschiedlicher Zahl von Argumenten anwendbar ist (das Dot-Konstrukt). Darauf muß das Indexierungsverfahren korrekt reagieren. In SASLOG dagegen haben Prädikate feste Stelligkeit. Ein Indexierungsverfahren für SASLOG könnte deshalb die Klauseln, deren Konklusion in der Zahl der Argumente mit der Anfrage übereinstimmen, durch einen zusätzlichen Index für die Stelligkeit auswählen.

Im jetzigen Stadium der Implementierung werden die Klauseln nur nach dem Prädikatnamen indexiert, da sie in einer Propertyliste unter diesem Namen abgelegt sind. Ein noch zu erstellendes Indexierungsverfahren, das die obigen Bemerkungen berücksichtigt, kann einfach integriert werden. Die Schnittstelle ist bestimmt durch die Funktionen *INX-ASS-TOP* und *INX-ASS-END* zum Einfügen, *INX-REX-1* zum Löschen und *INX-GET* zur Auswahl von Klauseln.

# 8 Einbettung in das LISP-System

Die Implementierungssprache LISP ist für den Benutzer von SASLOG nicht unsichtbar, SASLOG ist vielmehr in das LISP-System eingebettet. Diese Verbindung ist aber derjenigen zwischen SASL und PROLOG, die ja auf der Vereinheitlichung der Konzepte beruht, nicht gleichwertig. Zwei Ziele wurden mit der Einbettung in das LISP-System verfolgt:

- Die Ausführung von Ein- und Ausgabeoperationen ist in der Implementierungssprache LISP am einfachsten zu realisieren. Gerade auf den LISP-Maschinen erlaubt dies auch die Nutzung der graphischen Fähigkeiten und der komfortablen Programmierumgebung.

- In SASLOG geschriebene Programmteile können als Unterprogramme für LISP zur Verfügung gestellt werden.

In einer späteren Phase des Projektes kann diese Einbettung zu einer Programmierumgebung für SASLOG ausgebaut werden.

## 8.1 LISP-Ausdrücke in SASLOG

Es gibt zwei natürliche Ansatzpunkte, um in der PROLOG-Komponente LISP-Ausdrücke zu evaluieren, auf denen schon die LISP/PROLOG-Vereinheitlichung LISPLOG aufbaute (vgl. [Boley, Kammermeier u.a. 1985]):

1. Das in PROLOG bekannte, allerdings logisch impure Primitiv is wird für die Auswertung von LISP-Funktionen beibehalten:

   ```
   ?- _x is (read).
   ```

   Für die Reduktion von SASL-Ausdrücken ist es, wie in Abschnitt 5.3 erwähnt, nicht mehr notwendig.

2. LISP-Funktionsausdrücke können als Prämissen aufgerufen werden, müssen allerdings durch das Primitiv lispeval als LISP-Durchgriffe gekennzeichnet sein:

   ```
   ?- lispeval((streamp '_x)).
   ```

Die einzigen SASLOG-Terme, die als Argumente von LISP-Ausdrücken erlaubt sind, sind logische Variablen. Alle anderen Terme (Konstanten, Listen und SASL-Ausdrücke) müssen vorher an eine Variable gebunden werden. SASL-Ausdrücke werden bei der Instantiierung der Variablen vor der Übergabe an LISP vollständig reduziert. Die Listen in SASLOG und LISP sind typkonform. Dies wird dadurch erreicht, daß bei der Reduktion die durch P-Kombinatoren dargestellten Listen in LISP-Listen umgewandelt werden. Umgekehrt werden Listen als Werte von LISP-Funktionsaufrufen vor der Unifikation mit einem SASLOG-Term (im Prädikat is) in Listen mit dem P-Kombinator umgewandelt.

Beispiel: `?- _x == tl [1,2|(tl [3,4])], lispeval((print '_x))`

Die Instantiierung von _x erfordert die Reduktion von tl [1,2|(tl [3,4])]; dies ergibt die LISP-Liste (2 4), die als Argument an print übergeben wird. Das Quote im LISP-Ausdruck quotiert nicht die Variable, sondern deren Wert.

## 8.2 SASLOG als Untersystem in LISP-Programmen

Analog zur LISP/PROLOG-Schnittstelle in LISPLOG wird die umgekehrte Verwendung von in SASLOG formulierten Programmteilen als Teile eines LISP-Programms durch die Funktion PROVE bzw. durch vom Programmierer zu verwaltende Streams ermöglicht (vgl. [Dahmen 1987]).

Die Funktion PROVE muß mit einem numerischen Argument, das die Maximalzahl der zu berechnenden Lösungen angibt, aufgerufen werden. Da aber oft nicht von vornherein klar ist, wieviele Lösungen man haben will, sollte nach der Berechnung einer Lösung die Kontrolle an die aufrufende LISP-Funktion zurückgehen. Diese muß dann die Möglichkeit haben, weitere Lösungen anzufordern. Das ist durch Anlegen eines Streams möglich. Dazu wird die Funktion OPEN-SASLOG-STREAM mit der Liste der Goals, dem zu instantiierenden Ausdruck und dem Namen des Streams aufgerufen. Jeder Funktionsaufruf (GET-SASLOG-STREAM <name>) berechnet nun eine weitere Lösung des unter <name> definierten Streams. Soll keine weitere Lösung berechnet werden, kann man den Stream durch den Funktionsaufruf (CLOSE-SASLOG-STREAM <name>) abschließen. Dieser ganze Aufwand ist notwendig, weil LISP im Gegensatz zu SASL eben keine Möglichkeiten des Lazy Evaluation bietet.

Aufrufe von PROVE oder OPEN-SASLOG-STREAM, die wieder PROLOG-Anfragen als Argumente haben, dürfen nicht in LISP-Ausdrücken von is oder lispeval auftreten.

# 9 Benutzeroberfläche

## 9.1 L-Notation

Der SASLOG-Interpreter ist in LISP implementiert. Er erwartet Ausdrücke in LISP-Notation. Zur Umwandlung der original SASLOG-Syntax in diese Darstellung wird ein Parser benötigt, der die Transformation vor der Übergabe der Ausdrücke an den Interpreter durchführt. Der Parser wurde allerdings im Rahmen dieser Arbeit nicht realisiert. Die zur Verfügung stehende Syntax bietet dem Programmierer jedoch einige Erleichterungen gegenüber der internen Darstellung. So brauchen Listen nicht mit Hilfe des Kombinators P geschrieben zu werden. Für (P a b) schreibt man wie in der Originalsyntax [a | b]. Das Trennzeichen zwischen den Listenelementen ist das Leerzeichen. Die Liste [1 2 3] steht für (P 1 (P 2 (P 3 [ ]))). Variablen werden wie in der Originalsyntax durch einen Unterstrich gekennzeichnet, Konstanten durch Apostroph.

Die Originalsyntax und die als L-Notation bezeichnete bisher realisierte Syntax sind in Anhang A und B beschrieben. Am Beispiel des SASLOG-Prelude werden beide Notationen gegenübergestellt (Anhang C).

## 9.2 Der SASLOG-Toplevel

Durch den LISP-Funktionsaufruf (SASLOG) befindet man sich im SASLOG-Toplevel, der als read-eval-print-Schleife realisiert ist. Solange man sich auf dem SASLOG-Toplevel befindet, wird die Readtable lokal umgesetzt, so daß in L-Notation geschriebene Ausdrücke in die interne Darstellung transformiert werden. Die Ausgabe von Variablenbindungen und Funktionswerten erfolgt ebenfalls in der als L-Notation bezeichneten Syntax.

Als Eingabe sind Kommandos, PROLOG-Anfragen, SASL- und LISP-Ausdrücke erlaubt. Woran erkennt der Interpreter nun, wie die Eingabe zu interpretieren ist und wie werden die verschiedenen Ausdrücke bearbeitet?

### Kommandos

Die Kommandos entsprechen im wesentlichen den Toplevel-Kommandos von LISPLOG, die bei [Bernardi, Dahmen, Meyer 1987] erläutert werden. Es gibt Kommandos, die immer ein Argument haben (z.B. ass), solche, die nie ein Argument haben (z.B. help) und solche, die mit und ohne Argument aufgerufen

werden können (z.B. listing). Kommandos ohne Argument können ungeklammert eingegeben werden, Kommandos mit Argumenten müssen geklammert sein. Wird eine Eingabe als Kommando erkannt, wird die entsprechende Funktion ausgeführt.

Beispiele: listing
(listing append-r)

## PROLOG-Anfragen

Es kann sowohl jeweils eine einzelne als auch eine Konjunktion von Anfragen (dargestellt als LISP-Liste von Einzelanfragen) an das PROLOG-System gestellt werden. Einzelanfragen sind Strukturen, deren erstes Element ein Atom ist, das entweder eine Prädikatsdefinition hat oder ein PROLOG-Primitiv ist (not, call, var, nonvar).

Ist die Eingabe eine LISP-Liste deren erstes Element nicht atomar ist, so handelt es sich um eine Konjunktion von PROLOG-Anfragen. Die einzigen Ausnahmen von dieser Regel sind die where-Ausdrücke von SASL.

Beispiele: (member-r _x [1 2 3])
((member-r _x [1 2 3]) (== (> _x 1) TRUE))

## SASL-Ausdrücke

SASL-Ausdrücke sind entweder ZF-Ausdrücke, where-Ausdrücke, Funktionsanwendungen oder Variablen. Auf dem Toplevel eingegebene Variablen müssen durch s-def definiert sein. Funktionen sind entweder Kombinatoren (Listen sind durch den P-Kombinator repräsentiert) oder wurden durch s-def definiert. Where-Ausdrücke haben die Form (<Ausdruck> where <Definitionsliste>). Jede Struktur, deren zweites Element das Schlüsselwort where ist, wird als SASL-Ausdruck interpretiert.

Ist die Eingabe als SASL-Ausdruck erkannt, werden where- und ZF-Ausdrücke durch die Abstraktion eliminiert und der entstandene Kombinatorausdruck wird reduziert. Die Ausgabekomponente gibt das Reduktionsergebnis in der L-Notation aus, wobei die Reduktion rekursiv auf die Komponenten von Listen angewandt wird (vgl. *SASL-EVAL* in [Nökel, Rehbold 1986]).

## LISP-Ausdrücke

Als Eingabe auf dem SASLOG-Toplevel sind alle S-Ausdrücke erlaubt.

Alle angegebenen Bedingungen für die Zuordnung der Eingabe zu einer der vier Gruppen können getestet werden. Die oben angegebene Reihenfolge bestimmt auch die Priorität der Interpretation, wenn die Zuordnung nicht eindeutig ist. Es wird also zuerst getestet, ob die Eingabe ein Kommando ist, danach ob sie eine PROLOG-Anfrage und schließlich ob sie ein SASL-Ausdruck ist. Trifft keiner der Fälle zu, wird die Eingabe als LISP-Ausdruck evaluiert.

# 10 Ausblick

## 10.1 Ein Modulkonzept

Das Modulsystem von LISPLOG ([Dahmen 1987]) unterstützt die Entwicklung größerer Programmsysteme durch die Möglichkeit, die Klauseln in verschiedene Einheiten aufzuteilen. Jedes Modul bildet einen eigenen Namensraum für die Prädikatnamen, d.h. ein Name referiert in verschiedenen Moduln auf verschiedene Prädikatsdefinitionen (Klauseln). Durch explizite Import- und Exportdeklarationen kann sich ein Prädikatsname in verschiedenen Moduln auf dieselbe Definition beziehen. Zum Beweis einer Anfrage mit importiertem Prädikatnamen wird das aktuelle Modul gewechselt. Das Modul, das die Definition des Prädikats enthält, wird jetzt zum aktuellen Modul.

Überlegungen, dieses Modulsystem auf SASLOG zu übertragen, ergaben verschiedene Probleme:

- Ein integriertes System für SASLOG erfordert die Verteilung sowohl von Funktionen als auch von Klauseln auf verschiedene Moduln. Durch die Lazy Evaluation-Strategie ist es nicht nachvollziehbar, in welchem Modul ein Ausdruck reduziert wird. Ein einfaches Beispiel soll dies verdeutlichen:

```
Modul A:   import(B,[q]).          ;Importierung von q aus Modul B
           def f x = x + 3
           p(_z) :- q(f _z).

Modul B:   export([q]).            ;Exportierung von q
           q(5).
           q(6).
```

Die Funktion f wird im Modul A definiert. Zum Beweis der importierten Relation q wird das Modul gewechselt, wobei f _z aber erst während der Unifikation mit der Konstante 5, also in Modul B, reduziert wird. Aber f ist in Modul B nicht definiert. Das Problem ist also aus diesem Beispiel schon ersichtlich. Die Verzögerung kann sich aber über mehrere Modulwechsel hinziehen, so daß jede Funktion in jedes Modul exportiert werden müßte. Dies würde einer Speicherung der Funktionsdefinitionen an zentraler Stelle - wie in LISPLOG, wo LISP-Funktionen ja auch für alle Moduln global sind - entsprechen.

- Selbst globale Funktionsdefinitionen sind nicht unproblematisch. Enthält ein SASL-Term (explizit oder in der Definition einer aufgerufenen Funktion) einen PROLOG-Durchgriff in Form eines prove- oder ZF-Ausdrucks, so ist wieder unklar, in welchem Modul der Term reduziert wird. Also ist auch nicht nachvollziehbar, in welchem Modul das Goal zu beweisen ist. Als Konsequenz müßten auch die Prädikate in jedes Modul exportiert werden.

Man sieht, daß das Modulkonzept von LISPLOG nicht auf SASLOG übertragbar ist. Der Grund liegt darin, daß die Zuordnung einer Definition zu einem Funktions- bzw. Prädikatnamen vom Ort des Aufrufs abhängt, also statisch ist. In LISPLOG wird die Definition aber zum Zeitpunkt des Beweises gesucht. Ein Modulsystem für SASLOG muß eine statische Zuordnung von Funktionsdefinitionen zum Funktionsaufruf vornehmen. Zwei Vorschläge sollen dieses Prinzip verdeutlichen.

1. Jeder Bezeichner in einem Modul erhält beim Einlesen einen eindeutigen modulbezogenen Namen. So kann man z.B. den Modulnamen an jeden Bezeichner, getrennt durch ein Sonderzeichen, konkatenieren. Konstanten und Namen von logischen Variablen sind von dieser Konvention ausgenommen. Import- und Exportdeklarationen legen globale Zuordnungen fest. Modulwechsel sind für den Beweis nicht mehr notwendig.

   Im obigen Beispiel wird die Funktion f zu f-A, p zu p-A, x zu x-A. In Modul A wird q zu q-A und in Modul B zu q-B:

   ```
   def f-A x-A = x-A + 3
   p-A(_z) :- q-A(f-A _z).

   q-B(5).
   q-B(6).
   ```

   Die Import-Exportdeklarationen legen fest, daß zum Beweis von q-A die Definition von q-B anzuwenden ist. Die Definition von f-A ist nun eindeutig, so daß kein Fehler auftritt.

2. Der zweite Vorschlag entspricht den Packages in COMMON LISP. Die Umbenennung für die Bezeichner erfolgt analog zu Vorschlag 1. Statt expliziter Import-Exportdeklarationen wird an der Aufrufstelle einer importierten Funktion bzw. Relation gleich der entsprechend geänderte Name eingesetzt. Eine Umbenennung wird in diesem Fall nicht durchgeführt. Die obige Datenbasis sieht dann wie folgt aus:

```
Modul A:   def f x = x + 3
           p(_z) :- q-B(f _z).

Modul B:   q(5).
           q(6).
```

Nach dem Einlesen hat sie intern folgende Namensgebung:

```
def f-A x-A = x-A + 3
p-A(_z) :- q-B(f-A _z).

q-B(5).
q-B(6).
```

Der Vorteil dieser Realisierung ist, daß man bei der Namensgebung in einem Modul völlig frei ist, unabhängig von importierten Funktionen und Prädikaten. Im Beispiel würde eine Relation q in Modul A umbenannt zu q-A und somit verschieden von q-B. Außerdem wäre keine globale Zuordnungsliste notwendig und somit Aufwand zur Laufzeit gespart.

Das Modulkonzept von LISPLOG - d.h. die Aufteilung der Klauseln in verschiedene Einheiten - wurde zwar in der jetzigen Implementierung beibehalten, aber nur, weil es als Ausgangspunkt für das geplante Modulkonzept von SASLOG dienen kann. Zur Zeit ist es jedoch lediglich für SASLOG-Programme ohne ZF- und prove-Ausdrücke gefahrlos anwendbar.

## 10.2 Interaktionsumgebung

Eine Programmierumgebung, die die Entwicklung von SASLOG-Programmen unterstützt, sollte in das System integriert werden. Zur Zeit existiert eine Interaktionsumgebung, die von LISPLOG auf SASLOG übertragen und in einigen Punkten angepaßt wurde ([Meyer 1987]). Der Tracer basiert auf dem Box-Modell von Byrd ([Clocksin, Mellish 1981/84]), das zu einem erweiterten Box-Modell ausgebaut wurde. Mit dem Tracer können Informationen über Beweisabläufe ausgegeben werden. Daneben bietet das Break-Paket Möglichkeiten, den Programmablauf beim Erreichen vorher angegebener Breakpoints zu unterbrechen. Die interaktive Programmunterbrechnung wird in SASLOG nicht unterstützt. Ein drittes Werkzeug ist der Handschneider, der es dem Programmentwickler erlaubt, während des Programmlaufs initiale Cuts einzufügen.

All diese Konzepte unterstützen nur die Programmentwicklung in der logischen Komponente. Wünschenswert wäre jedoch eine kombinierte Interaktionsumgebung für Funktions- und Hornklauselprogramme.

## 10.3 Ein Typkonzept

In Miranda erweiterte Turner KRC (eine Weiterentwicklung von SASL) um ein polymorphes Typkonzept ([Turner 1985]). Mycroft und O'Keefe haben für PROLOG ebenfalls ein polymorphes Typkonzept entwickelt ([Mycroft, O'Keefe 1984]). TEL ist eine typisierte funktional-logische Sprache erster Ordnung. Sie unterstützt sowohl Subtypes als auch polymorphe Typkonstruktoren ([Smolka 1987]).

Aufbauend auf Erfahrungen mit diesen Systemen könnte ein Typkonzept für SASLOG entworfen werden, das es erlaubt, viele Programmierfehler, die auf Verletzungen von Typkonventionen beruhen, schon zur Compile-Zeit zu erkennen.

# 11 Zusammenfassung

SASLOG ist als rein funktional-logische Sprache ohne impure Eigenschaften konzipiert. Dadurch soll die Entwicklung einfacher, leicht verständlicher und verifizierbarer Programme ermöglicht werden. So wurde mit SASL eine reine funktionale Sprache gewählt. In PROLOG - der logischen Komponente - sind dynamische Datenbasisänderungen verboten und der Cut kann nur in eingeschränkter Form verwendet werden.

Auf der anderen Seite soll SASLOG aber für die Programmierung von Expertensystemen und Werkzeugen verwendet werden. Deshalb wurde die Erfüllung theoretischer Anforderungen, wie die Vollständigkeit des Beweisverfahrens, zurückgestellt. Um diese zu erreichen, hätte man zum Beispiel die Reduktion zu Narrowing verallgemeinern müssen. Dies hätte aber die Effizienz wahrscheinlich so stark verringert, daß SASLOG für den praktischen Einsatz in der Expertensystementwicklung wohl kaum akzeptiert worden wäre.

SASLOG ist eine symmetrische Sprache in dem Sinn, daß sowohl funktionale Terme im logischen Teil berechnet und umgekehrt logische Beweise von der funktionalen Seite aktiviert werden können, wobei beide Komponenten auf den gleichen Objekten operieren. Beide Richtungen der Integration nutzen die Verwandtschaft der Reduktion mit Lazy Evaluation und des Tiefensuchverfahrens mit Backtracking aus. So kann eine jeweils dem Teilproblem angepaßte Repräsentation gewählt werden, ohne Rücksichten auf die Schnittstellen zwischen unterschiedlichen Darstellungs- und Bearbeitungsformen nehmen zu müssen. Die Einbindung der Mengendarstellung von PROLOG-Anfragen in das bestehende ZF-Konstrukt von SASL ermöglicht eine elegante Verbindung funktionaler und relationaler Ausdrücke. Durch Einführung funktionaler Ausdrücke als Terme in Relationen erreicht man eine starke Verschmelzung auf Termebene. Diese erfordert eine Modifikation des Unifikationsalgorithmus hin zu einer semantischen Unifikation.

Die natürliche Verbindung zwischen funktionaler und relationaler Darstellung macht SASLOG für die Anwendung im Bereich der Expertensysteme interessant. Zur Modellierung eines Problemgebietes müssen meist prozedurale und deklarative Darstellungen verwendet werden. Funktionale Terme als Objekte erlauben eine natürliche Einbindung von Prozeduren in Wissensstrukturen (procedural attachment).

Für die Entwicklung großer Systeme benötigt man oft Werkzeuge, die die Modellierung von Teilaspekten unterstützen. Eine funktional-logische Sprache ist auch für die Implementierung solcher Werkzeuge, die Wissen darstellen und verarbeiten, geeignet.

## Literatur

[Abramson 1984]
H. Abramson:
A Prological Definition of HASL: A Purely Functional Language with unification-based conditional Binding Expressions.
In: New Generation Computing, Vol. 2, No. 1, 1984, S. 3-35
Auch in: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986

[Backus 1978]
J. Backus:
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
In: Communications of the ACM, Vol. 21, No, 8, August 1978, S. 613-641

[Barbuti, Bellia, Levi 1986]
R. Barbuti, M. Bellia, G. Levi:
LEAF: A Language which integrates Logic, Equations and Functions.
In: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986, S. 201-238

[Bellia, Levi 1986]
M. Bellia, G. Levi:
The Relation between Logic and Functional Languages: A Survey.
In: Journal of Logic Programming, Vol. 3, No. 3, 1986, S. 217-236

[Bellia u.a. 1982]
M. Bellia, E. Dameri, P. Degano, G. Levi, M. Martelli:
Applicative Communicating Processes in First Order Logic.
In: Proc. 5th Int. Symp. on Programming, LNCS 137, Springer Verlag, 1982, S. 1-14

[Bellia, Degano, Levi 1982]
M. Bellia, P. Degano, G. Levi:
The Call-by-Name Semantics of a Clause Language with Functions
In: K. Clark, S. A. Tärnlund (Eds.):
Logic Programming.
Academic Press, London, 1982, S. 281-296

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

[Bernardi, Dahmen, Meyer 1987]
A. Bernardi, M. Dahmen, M. Meyer:
LISPLOG Benutzerhandbuch.
Universität Kaiserslautern, FB Informatik, SEKI Working Paper SWP 87-01, Januar 1987

[Boley 1986]
H. Boley (Ed.):
A Bird's-Eye View of LISPLOG: The LISP/PROLOG Integration with Initial-Cut Tools.
Universität Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-08, Dezember 1986, SWP-87-04, Juni 1987

[Boley, Kammermeier u.a. 1985]
H. Boley, F. Kammermeier u. die LISPLOG-Gruppe:
LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung.
Universität Kaiserslautern, FB Informatik, MEMO SEKI-85-03, August 1985
Kurzfassung in: B. Nebel (Ed.):
Papiere zum Workshop Logisches Programmieren und Lisp.
TU Berlin, FB Informatik, KIT-REPORT 31,
Dezember 1985, S. 36-53

[Bowen, Kowalski 1982]
K.A. Bowen, R.A. Kowalski:
Amalgamating Language and Metalanguage in Logic Programming.
In: K. Clark, S. A. Tärnlund (Eds.):
Logic Programming.
Academic Press, London, 1982, S. 153-172

[Colmerauer u.a. 1972]
A. Colmerauer, H. Kanoui, R. Pasero, P. Roussel:
Un Systeme de Communication Homme-Machine on Francais.
Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Luminy, 1972

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

[Curry, Feys 1958]
H.B. Curry, R. Feys:
Cominatory Logic, Volume I.
North Holland, Amsterdam, 1958

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

[Dahmen 1987]
M. Dahmen:
Module und Streams in LISPLOG.
Universität Kaiserslautern, FB Informatik, SEKI Working Paper SWP-87-06,
Juni 1987

[Darlington, Field, Pull 1986]
J. Darlington, A. J. Field, H. Pull:
The Unifcation of Functional and Logic Languages.
In: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986, S. 37-70

[Dincbas, van Hentenryck 1987]
M. Dincbas, P. van Hentenryck:
Extended Unification Algorithms for the Integration of Functional Programming
into Logic Prpgramming.
In: Journal of Logic Programming, Vol. 4, No. 3, 1987, S. 199-227

[Fribourg 1984]
L. Fribourg:
Oriented Equational Clauses as a Programming Language.
In: Journal of Logic Programming, Vol. 1, No. 2, 1984, S. 165-177

[Goguen, Meseguer 1984/86]
J. A. Goguen, J. Meseguer:
Equality, Types, Modules, and (Why not?) Generics for Logic Programming.
In: Journal of Logic Programming, Vol. 1, No. 2, 1984, S. 179-210
Erweiterte Fassung in: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986, S. 295-363

[Hansson, Haridi, Tärnlund 1982]
A. Hansson, S. Haridi, S. A. Tärnlund:
Properties of a Logic Programming Language.
In: K. Clark, S. A. Tärnlund (Eds.):
Logic Programming.
Academic Press, London, 1982, S. 267-280

[Hinkelmann, Nökel, Rehbold 1988]
K. Hinkelmann, K. Nökel, R. Rehbold:
SASLOG: Lazy Evaluation Meets Backtracking.
Universität Kaiserslautern, FB Informatik, SEKI Report SR-88-01, erscheint

[Hölldobler 1984]
S. Hölldobler:
Functional and Logic Programming.
Universität der Bundeswehr, FB Informatik, München, Bericht Nr. 8408, Juni 1984

[Hölldobler, Furbach, Laußermair 1985]
S. Hölldobler, U. Furbach, T. Laußermair:
Extended Unification and its Implementation.
Universität der Bundeswehr, FB Informatik, München, Bericht Nr. 8504, Juni 1985

[Kammermeier 1986]
F. Kammermeier:
LISPLOG im Kontext anderer LISP/PROLOG-Vereinheitlichungen.
Universität Kaiserslautern, FB Informatik, SEKI Working Paper SWP-86-09, Dezember 1986

[Kornfeld 1983]
W. Kornfeld:
Equality for Prolog.
In: Proc. 8th IJCAI-83, Karlsruhe, August 1983, S. 514-519
Auch in: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986, S. 279-293

[Meyer 1987]
M. Meyer:
Entwurf und Implementierung einer Interaktionsumgebung für LISPLOG.
Universität Kaiserslautern, FB Informatik, SEKI Working Paper SWP-87-02, Januar 1987, zweite Auflage Dezember 1987

[Mycroft, O'Keefe 1984]
A. Mycroft, R.A. O'Keefe:
A Polymorphic Type System for Prolog.
In: Artificial Intelligence, Vol. 23, 1984, S. 295-307

[Narain 1986]
S. Narain:
A Technique for Doing Lazy Evaluation in Logic.
In: Journal of Logic Programming, Vol. 3, No. 3, 1986, S. 259-276

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

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

[Reddy 1986]
U. S. Reddy:
On the Relationship between Logic and Functional Languages.
In: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986, S. 3-36

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

[Richards 1984]
H. Richards:
Programming in SASL.
Burroughs Austin Research Center, ARC 84-21, Austin, 1984

[Robinson 1965]
J.A. Robinson:
A Macine-Oriented Logic Based on the Resolution Principle.
In: Journal of the ACM, Vol. 12, No. 1, 1965, S. 23-41

[Robinson 1983]
J.A. Robinson:
A Proposal to develop a "Fifth Generation" Programming System Based on Logic Programming and Highly Parallel Reduction Machine
Logic Programming Research, School of Computer and Information Science, Syracuse University, New York, February 1983

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

[Robinson, Sibert 1982b]
J.A. Robinson, E. Sibert:
LOGLISP: An Alternative to PROLOG.
In: Machine Intelligence 10, Chichester, 1982, S. 399-419

[Schönfinkel 1924]
M. Schönfinkel:
Über die Bausteine der mathematischen Logik.
In: Mathematische Annalen, Vol. 92, 1924, S. 305-316

[Stoy 1977]
J. Stoy:
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
The MIT Press, Cambridge Massachusetts, 1977

[Smolka 1985]
G. Smolka:
FRESH: A Higher-order Language with Unification and Multiple Results.
In: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986,S. 469-524

[Smolka 1987]
G. Smolka:
TEL, Version 0.9, Report and User Manual
Universität Kaiserslautern, FB Informatik, SEKI Report SR-87-11,
Oktober 1987

[Subrahmanyam, You 1986]
P. A. Subrahmanyam, J.-H. You:
FUNLOG: A Computational Model integrating Logic Programming and Functional Programming.
In: D. DeGroot, G. Lindstrom (Eds.):
Logic Programming - Functions, Relations, and Equations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1986, S. 157-198

[Turner 1979]
D.A. Turner:
A New Implementation Technique for Applicative Languages.
In: Software - Practice and Experience, Vol. 9(1), 1979, S. 31-49

[Turner 1981]
D.A. Turner:
Recursion Equations as a Programming Language.
In: J. Darlington, P. Henderson, D.A. Turner (Eds.):
Functional Programming and its Applications.
Cambridge University Press, Cambridge, 1981

[Turner 1983]
D.A. Turner:
SASL Language Manual (Revised Version). Canterbury 1983

[Turner 1985]
D.A. Turner:
Miranda: A non-strict functional language with polymorphic types
In: 1985 Conference on Functional Programming and Computer Architecture
Springer Verlag, Berlin, 1985, S. 1-16

[van Emdem, Kowalski 1976]
M.H. van Emden, R.A. Kowalski:
The Semantics of Predicate Logic as a Programming Language.
In: Journal of the ACM, Vol. 23, No. 4, Oktober 1976, S. 733-742

[van Emden, Yukawa 1986]
M.H. van Emden, K. Yukawa:
Equational Logic Programming.
Technical Report CS-86-05, Dept. of Computer Science,
University of Waterloo, Ontario, Canada, March 1985

[Voda, Yu 1984]
P. J. Voda, B. Yu: RF-Maple:
A Logic Programming Language with Functions, Types, and Concurrency.
Proc. International Conference on Fifth Generation Computer Systems 1984,
ICOT 1984, S. 341-347

[Wadler 1985]
P. Wadler:
How to Replace Failure by a List of Successes
In: 1985 Conference on Functional Programming and Computer Architecture
Springer Verlag, Berlin, 1985, S. 113-128

[Wadsworth 1971]
C.P. Wadsworth:
Semantics and Pragmatics of the λ-calculus.
DPhil Thesis, Oxford 1971

# Anhang A: Die Originalsyntax von SASLOG

Dieser Abschnitt enthält eine Beschreibung der SASLOG-Syntax in einer Variante der Backus-Naur-Form. Es gelten folgende Konventionen:

1. Die Definition wird von dem zu definierenden Objekt durch ::= getrennt.
2. Alternativen sind durch senkrechte Striche getrennt. In Apostroph eingeschlossene Striche stehen für sich selbst ('|').
3. Reservierte Namen sind unterstrichen.
4. Geschweifte Klammern schließen optionale Ausdrücke ein, z.B. {Ausdruck}. {Ausdruck}* steht für null, ein oder mehrere Vorkommen von Ausdruck. {Ausdruck}+ steht für mindestens ein Vorkommen von Ausdruck.
5. Alle anderen Zeichen stehen für sich selbst.

| | | |
|---|---|---|
| SASLOG-Programm | ::= | {Klausel \| Definition \| where-Ausdruck } |
| Definition | ::= | def Parameter-def |
| Klausel | ::= | Fakt \| Regel \| Ziel |
| Fakt | ::= | Konklusion . |
| Regel | ::= | Konklusion :- Prämisse {, Prämisse}*. \| |
| | | Konklusion :- Cut {, Prämisse}*. |
| Ziel | ::= | ?- Prämisse {, Prämisse}*. |
| Konklusion | ::= | Struktur |
| Prämisse | ::= | Struktur \| unify-Prämisse \| not-Prämisse \| call-Prämisse \| |
| | | is-Prämisse \| lispeval-Prämisse \| fail |
| Cut | ::= | ! |
| Struktur | ::= | Bezeichner \| Bezeichner({Term}+) |
| unify-Prämisse | ::= | Term == Term |
| not-Prämisse | ::= | not(Prämisse {, Prämisse}*) |
| call-Prämisse | ::= | call(Prämisse) \| call(Term) |
| is-Prämisse | ::= | Term is LISP-Ausdruck |
| lispeval-Prämisse | ::= | lispeval(LISP-Ausdruck) |
| Term | ::= | Bezeichner \| Zahl \| Boolean \| Konstante \| Variable \| |
| | | Liste \| (where-Ausdruck) |
| Boolean | ::= | true \| false |
| Konstante | ::= | 'Bezeichner |
| Variable | ::= | einfache-Variable \| anonyme-Variable |
| einfache-Variable | ::= | _Bezeichner |
| anonyme-Variable | ::= | _? |

| | | |
|---|---|---|
| where-Ausdruck | ::= | Ausdruck {where {Def}+} |
| Ausdruck | ::= | op-Ausdruck {-> true-Zweig {; false-Zweig}} |
| true-Zweig | ::= | Ausdruck |
| false-Zweig | ::= | Ausdruck |
| op-Ausdruck | ::= | {Präfix-op} Komb {Infix-op Komb}* {Postfix-op} \| prove-Ausdruck |
| prove-Ausdruck | ::= | prove (Prämisse {,Prämisse}*) |
| Komb | ::= | {Term}+ |
| Liste | ::= | Aufzählungsliste \| ZF-Ausdruck |
| Aufzählungsliste | ::= | [ {op-Ausdruck {, op-Ausdruck}* {'\|' op-Ausdruck}} ] |
| ZF-Ausdruck | ::= | [ op-Ausdruck {Qualifier}+ ] |
| Qualifier | ::= | Generator \| PROLOG-Generator |
| Generator | ::= | ; Bezeichner <- Bereich { ; Filter}* |
| PROLOG-Generator | ::= | ; Namelist <- Prämisse { , Prämisse}* { ; Filter}* |
| Filter | ::= | op-Ausdruck |
| Bereich | ::= | op-Ausdruck |
| Namelist | ::= | [ Bezeichner { , Bezeichner}* ] |
| Def | ::= | Namelist-def \| Parameter-def |
| Namelist-def | ::= | Namelist-Struktur = where-Ausdruck |
| Namelist-Struktur | ::= | [ Bezeichner {, Bezeichner}* {'\|' Bezeichner} ] |
| Parameter-def | ::= | Bezeichner {Parameter}* = where-Ausdruck |
| Parameter | ::= | Bezeichner \| Boolean \| Zahl \| Konstante \| Parameter-list |
| Parameter-list | ::= | [ {Parameter {, Parameter}* {'\|' Parameter}} ] |
| Präfix-op | ::= | - \| ~ |
| Infix-op | ::= | + \| - \| * \| / \| = \| < \| > \| >= \| <= \| ++ \| -- \| .. \| & \| REM \| DIV |
| Postfix-op | ::= | ... |

Der Aufbau der Bezeichner, also welche Zeichen darin erlaubt sind, und der Bereich der Zahlen hängt von der konkreten Implementierung ab.

## Anhang B: Die L-Notation für SASLOG

Dieser Abschnitt enthält eine Beschreibung der L-Notation für SASLOG in einer Variante der Backus-Naur-Form. Es gelten die gleichen Konventionen wie in Anhang A für die Beschreibung der Originalsyntax.

| | | |
|---|---|---|
| SASLOG-Programm | ::= | {Klausel \| Definition \| Ausdruck } |
| Definition | ::= | (s-def Parameter-def) |
| Klausel | ::= | (Konklusion { Prämisse}*) |
| Konklusion | ::= | {Cut}Struktur |
| Prämisse | ::= | Struktur \| Atom-Prämisse \| not-Prämisse \| call-Prämisse \| is-Prämisse \| lispeval-Prämisse |
| Cut | ::= | ! |
| Struktur | ::= | (Bezeichner {Term}*) |
| not-Prämisse | ::= | (not { Prämisse}*) |
| call-Prämisse | ::= | (call Prämisse) \| (call Term) |
| Atom-Prämisse | ::= | Bezeichner \| fail |
| is-Prämisse | ::= | (is Term LISP-Ausdruck) |
| lispeval-Prämisse | ::= | (lispeval LISP-Ausdruck) |
| Term | ::= | Bezeichner \| Zahl \| Boolean \| Konstante \| Variable \| Liste \| Ausdruck |
| Boolean | ::= | true \| false |
| Konstante | ::= | 'Bezeichner |
| Variable | ::= | einfache-Variable \| anonyme-Variable |
| einfache-Variable | ::= | _Bezeichner |
| anonyme-Variable | ::= | _? |
| Ausdruck | ::= | where-Ausdruck \| bedingter-Ausdruck \| prove-Ausdruck \| Kombination |
| where-Ausdruck | ::= | (Ausdruck where {Def}+) |
| bedingter-Ausdruck | ::= | (if Ausdruck true-Zweig false-Zweig) |
| true-Zweig | ::= | Ausdruck |
| false-Zweig | ::= | Ausdruck |
| Kombination | ::= | ({Term}+) |
| prove-Ausdruck | ::= | (prove {Prämisse}+) |
| Liste | ::= | Aufzählungsliste \| ZF-Ausdruck |
| Aufzählungsliste | ::= | [ {{Ausdruck}+ {'\|' Ausdruck}} ] |
| ZF-Ausdruck | ::= | [ ZF Ausdruck {Qualifier}+ ] |
| Qualifier | ::= | Generator \| PROLOG-Generator |
| Generator | ::= | (Bezeichner <- Bereich) {Filter}* |

| | | |
|---|---|---|
| PROLOG-Generator | ::= | (Namelist <- { Prämisse}+) {Filter}* |
| Filter | ::= | Ausdruck |
| Bereich | ::= | Ausdruck |
| Namelist | ::= | [ {Bezeichner }+ ] |
| Def | ::= | Namelist-def \| Parameter-def |
| Namelist-def | ::= | (Namelist-Struktur Ausdruck) |
| Namelist-Struktur | ::= | [ {Bezeichner}+ {'\|' Bezeichner} ] |
| Parameter-def | ::= | ((Bezeichner {Parameter}*) Ausdruck) \| |
| | | (Bezeichner Ausdruck) |
| Parameter | ::= | Bezeichner \| Boolean \| Zahl \| Konstante \| Parameter-list |
| Parameter-list | ::= | [ {{Parameter}+ {'\|' Parameter}} ] |

Die Bezeichner sind LISP-Atome. Ihr genauer Aufbau hängt von der konkreten Implementierung ab. Sie dürfen jedoch keines der Zeichen [, ], |, _, ', ! enthalten.

## Anhang C: Das SASLOG-Prelude

Dieser Abschnitt enthält das SASLOG-Prelude, also die vordefinierten Funktionen und Relationen. Jede Definition und jede Klausel sind sowohl in der `Originalsyntax` als auch in der L-Notation angegeben.

1. Als primitive Funktionen werden solche Funktionen bezeichnet, die sich nicht aus einfacheren Funktionen zusammensetzen lassen und die deshalb direkt als Kombinatoren reduziert werden. Für diese wird hier die Schreibweise angegeben:

   Gleichheit:

   ```
   x = y     oder   eq x y
   (= x y)
   ```

   bedingter Ausdruck:

   ```
   b -> x; y
   (if b x y)
   ```

   Arithmetik

   - op ∈ {+, -, /, *, >, >=, <, <=}: `x op y`
     `(op x y)`
   - op ∈ {plus, times, rem}: `op x y`
     `(op x y)`
   - op ∈ {exp, log, sqrt, sin, cos, arctan, entier}: `op x`
     `(op x)`

   Typzugehörigkeitsprädikate

   op ∈ {isboolean, isnumber, islist, isconstant, ischaracter}: `op x`
   `(op x)`

2. Die folgenden Funktionen können mit Hilfe anderer Funktionen definiert werden, sind aber aus Effizienzgründen direkt als Kombinatoren implementiert. Hier werden ihre SASL-Definitionen angegeben:

   Absolutbetrag von n:

   ```
   def abs n = n < 0 -> -n ; n
   (s-def ((abs n) (if (< n 0) (- 0 n) n)))
   ```

Konjunktion:

```
def and x y = x -> y ; FALSE
(s-def ((and x y) (if x y FALSE)))
```

Listenkonkatenation:

```
def append []     y = y
    append [a|x] y = [a|(append x y)]
(s-def  ((append [] y)  y)
        ((append [a|x] y)  [a | (append x y)])))
```

andere Schreibweise: `x ++ y`, `(++ x y)`

Vertauschen der Argumente:

```
def converse f x y = f y x
(s-def ((converse f x y) (f y x)))
```

Komposition:

```
def dot f g = h where h x = f ( g x)
(s-def ((dot  f g) (h where ((h x) (f (g x))))))
```

Listenfilterfunktion:

```
def filter p [a|x] = (p a) -> [a|(filter p x)] ;
                              filter p x
    filter p []    = []
(s-def  ((filter p [a | x]) (if (p a) [a | (filter p x)] (filter p x)))
        ((filter p [ ]) [ ]))
```

Listenfaltung nach links:

```
def foldl f r [a|x] = foldl f (f r a) x
    foldl f r []    = r
(s-def  ((foldl f [a | x]) (foldl f (f r a) x))
        ((foldl f r) r))
```

Listenfaltung nach rechts:

```
def foldr f r [a|x] = f a (foldr f r x)
    foldr f r []    = r
(s-def  ((foldr f [a | x]) (f a (foldr f r x)))
        ((foldr f r) r))
```

Länge einer Liste:

```
def length []     = 0
    length [a|x] = 1 + length x
```

```
(s-def  ((length [ ]) 0)
        ((length [a | x]) (+ 1 (length x))))
```

Anwendung einer Funktion f auf jedes Element der Liste l:

```
def map f []     = []
    map f [a|x] = [(f a)  |  (map f x)]
```

```
(s-def  ((map f [ ]) [ ])
        ((map f [a | x]) [ (f a) | (map f x)] ))
```

Element einer Liste:

```
def member l a = any (map (eq a) l)
```

```
(s-def ((member l a) (any (map (eq a) l))))
```

Ungleichheit:

```
def neq x y = not (x = y)
```

```
(s-def ((neq x y) (not (= x y))))
```

Negation:

```
def not TRUE  = FALSE
    not FALSE = TRUE
```

```
(s-def  ((not TRUE) FALSE)
        ((not FALSE) TRUE))
```

Disjunktion:

```
def or x y = x -> TRUE ; y
```

```
(s-def ((or x y) (if x TRUE y)))
```

Rest einer Liste:

```
def tl [a|x] = x
```

```
(s-def ((tl [a | x]) x))
```

Kopf einer Liste:

```
def hd [a|x] = a
```

```
(s-def ((hd [a | x]) a))
```

3. Die Definitionen folgender nützlicher Funktionen werden zusammen mit dem SASLOG-System geladen:

Konjunktion über eine Liste boolescher Werte:

```
def all = foldr and TRUE
(s-def ((all p) (foldr and TRUE)))
```

Disjunktion über eine Liste boolescher Werte:

```
def any = foldr or FALSE
(s-def ((any p) (foldr or FALSE)))
```

Verkettung aller Elemente einer Liste:

```
def concat = foldr append []
(s-def (concat (foldr ++ [] )))
```

Paarbildung:

```
def cons = P
(s-def (cons P))
```

Liste der ganzen Zahlen zwischen m und n:

```
def count m n = m > n -> [] ; [m|(count (m+1) n)]
(s-def ((count m n) (if (> m n) [] [m | (count (+ 1 m) n) ] )))
```

Rest einer Liste nach n Elementen:

```
def drop n []     = []
    drop n [a|x] = n > 0 -> drop (n-1) x ; [a|x]
(s-def ((drop n [ ]) [ ])
       ((drop n [a | x]) (if (> n 0) (drop (- n 1) x) [a | x])))
```

Anwendung einer Funktion f auf die Zahlen zwischen a und b:

```
def for a b f = map f (count a b)
(s-def ((for a b f) (map f (count a b))))
```

Liste der ganzen Zahlen ab n:

```
def from n = [b|(from (n+1))]
(s-def ((from n) [n | (from (+ 1 n))] ))
```

Mengendurchschnitt:

```
def intersection l m = filter (member l) m
(s-def ((intersection l m) (filter (member l) m)))
```

Liste [x, f x, f (f x), ...]:

```
def iterate f = [x| (iterate f (f x))]
```
```
(s-def ((iterate f x) [x | (iterate f (f x))] ))
```

Listendifferenz:

```
def listdiff []     m      = []
    listdiff l      []     = l
    listdiff [a|x] [a|y] = listdiff x y
    listdiff [a|x] [b|y] =
                          listdiff [a|(listdiff x [b])] y
```
```
(s-def ((listdiff [ ] m) [ ])
       ((listdiff l [ ]) l)
       ((listdiff [a | x] [a | y]) (listdiff x y))
       ((listdiff [a | x] [b | y]) (listdiff [a | (listdiff x [b])] y)))
```

Mehrfachvorkommen löschen:

```
def mkset []     = []
    mkset [a|x] = [a|(mkset (filter (neq a) x))]
```
```
(s-def ((mkset [ ]) [ ])
       ((mkset [a | x]) [a | (mkset (filter (neq a) x))] ))
```

Produkt über alle Listenelemente:

```
def product = foldr times 1
```
```
(s-def (product (foldr times 1)))
```

Umkehrung einer Liste:

```
def reverse = foldl cons []
```
```
(s-def (reverse (foldl cons [ ] )))
```

Summe über alle Listenelemente:

```
def sum = foldr plus 0
```
```
(s-def (sum (foldr plus 0)))
```

Erste n Elemente einer Liste:

```
def take n []     = []
    take n [a|x] = n > 0 -> [a|(take (n-1) x)] ; []
```
```
(s-def ((take n [ ]) [ ])
       ((take n [a | x]) (if (> n 0) [a | (take (- n 1) x)] [ ])))
```

Mengenvereinigung:

```
def union x y = filter (dot not (member y)) x ++ y
```
```
(s-def ((union x y) (++ (filter (dot not (member y)) x) y)))
```

Matrixtransposition:

```
def zip x =  hd x = [] -> [] ;
                              [(map hd x)|(zip (map tl x)]
(s-def ((zip x)  (if (= [] (hd x)) [] [(map hd x) | (zip (map tl x))] )))
```

4. Die folgenden Relationen sind primitiv, d.h. sie sind in den Interpreter integriert:

Test auf freie Variable:

```
var(_x)
nonvar(_x)
(var _x)
(nonvar _x)
```

Negation:

```
not(_x1, _x2,...,_xn)
(not _x1 _x2 ... _xn)
```

Aufruf des Beweisers:

```
call(_x)
(call _x)
```

5. Die folgenden Relationen werden zusammen mit dem SASLOG-System in die Datenbasis geladen:

Unifizierbarkeit von Termen:

```
_x == _x
(ass (== _x _x))
```

Unendlich viele Backtracking Punkte:

```
repeat.
repeat :- repeat.
(ass (repeat))
(ass (repeat) (repeat))
```

Enthaltensein in einer Liste:

```
member-r(_e,[_e|_?]).
member-r(_e,[_?|_rest]) :- member-r(_e,_rest).
(ass (member-r _e [_e | _?]))
(ass (member-r _e [_? | _rest])  (member-r _e _rest))
```

Zusammensetzen von Listen:

```
append-r([],_l,_l).
append-r([_a|_l1],_l2,[_a|_l3]) :-
                              append-r(_l1,_l2,_l3).
(ass (append-r [ ] _l _l))
(ass (append-r [_a|_l1] _l2 [_a|_l3]) (append-r _l1 _l2 _l3))
```

Umkehrung von Listen:

```
reverse-r([],[]).
reverse-r([_a|_x],_y) :-  reverse-r(_x,_z),
                          append-r(_z,[_a],_y).
(ass (reverse-r [ ] [ ]))
(ass (reverse-r [_a|_x] _y) (reverse-r _x _z) (append-r _z [_a] _y))
```

Invertierbare Arithmetik:

```
addit(_x,_y,_sum) :-   TRUE == (isnumber _x),
                       TRUE == (isnumber _y),
                       _sum == (_x + _y).
addit(_x,_y,_sum) :-   TRUE == (isnumber _x),
                       TRUE == (isnumber _sum),
                       var(_y),
                       _y == (_sum - _x).
addit(_x,_y,_sum) :-   TRUE == (isnumber _y),
                       TRUE == (isnumber _sum),
                       var(_x),
                       _x == (_sum - _y).
subit(_x,_y,_diff) :-  addit(_diff,_y,_x).
multit(_x,_y,_prod) :- TRUE == (isnumber _x),
                       TRUE == (isnumber _y),
                       _prod == (_x * _y).
multit(_x,_y,_prod) :- TRUE == (isnumber _x),
                       TRUE == (isnumber _prod),
                       var(_y),
                       _y == (_prod / _x).
multit(_x,_y,_prod) :- TRUE == (isnumber _y),
                       TRUE == (isnumber _prod),
                       var(_x),
                       _x == (_prod / _y).
divit(_x,_y,_quot) :-  multit(_quot,_y,_x).
```

```
(ass   (addit _x _y _sum)      (== TRUE (isnumber _x))
                               (== TRUE (isnumber _y))
                               (== _sum (+ _x _y)))
(ass  (addit _x _y _sum)       (== TRUE (isnumber _x))
                               (== TRUE (isnumber _sum))
                               (var _y)
                               (== _y (- _sum _x)))
(ass  (addit _x _y _sum)       (== TRUE (isnumber _y))
                               (== TRUE (isnumber _sum))
                               (var _x)
                               (== _x (- _sum _y)))
(ass  (subit _x _y _diff)      (addit _diff _y _x))
(ass  (multit _x _y _prod)     (== TRUE (isnumber _x))
                               (== TRUE (isnumber _y))
                               (== _prod (* _x _y)))
(ass  (multit _x _y _prod)     (== TRUE (isnumber _x))
                               (== TRUE (isnumber _prod))
                               (var _y)
                               (== _y (div _prod _x)))
(ass  (multit _x _y _prod)     (== TRUE (isnumber _y))
                               (== TRUE (isnumber _prod))
                               (var _x)
                               (== _x (div _prod _y)))
(ass  (divit _x _y _quot)      (multit _quot _y _x))
```

## Anhang D: Der SASLOG-Interpreter

Das Programm-Listing des Interpreters wurde aus Platzgründen weggelassen, es ist aber auf Anfrage erhältlich.