Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

Patrik Koch
Lehrstuhl für Informatik II
Rheinisch-Westfälische Technische Hochschule Aachen
patrik@zeus.informatik.rwth-aachen.de

Zusammenfassung

Deklarative Programmiersprachen ermöglichen eine problemorientierte Beschreibung von Sachverhalten. Andererseits verursacht die große Distanz zur realen Maschine Effizienzprobleme bei der Auswertung solcher Programme.
In dieser Arbeit wird ein Intelligentes Backtracking – Verfahren für die Auswertung von Programmen der funktional-logischen Sprache Babel vorgestellt. Kern des Verfahrens ist eine dynamische Verkleinerung des Suchraums zum Auswertungszeitpunkt aufgrund einer Analyse der bisher erfolgten Berechnung.

1 Einleitung

Die deklarative Programmierung erfreut sich insbesondere in den letzten Jahren eines steigenden Interesses der Informatik-Fachwelt. Die zwei wichtigsten Vertreter der deklarativen Programmierparadigmen sind die logische und die funktionale Programmierung. Gründe für das steigende Interesse sind zum einen der Wunsch nach einer möglichst problemorientierten Programmierung („vertical migration“) und zum anderen die Verfügbarkeit immer leistungsfähigerer Implementierungstechniken und Hardware (u.a. Parallelrechner).

Der bekannteste Vertreter der logischen Programmiersprachen ist Prolog mit der Relation als wesentlichstem Ausdrucksmittel. Der der Relation innewohnende Nichtdeterminismus (Invertibilität von Ein- bzw. Ausgabeparametern) bewirkt eine effiziente Beschreibung des Problems ebenso wie Effizienzprobleme bei der Berechnung einer Anfrage an ein Programm (mit Unifikation, Resolution und Backtracking).
Grundlage der meisten Programmsysteme für die Sprache Prolog ist die WAM[1] bzw. leichte Abwandlungen von ihr. Versuche, die Effizienz eines Logik-Programms zum Auswertungszeitpunkt zu verbessern, beziehen sich meistens auf das *Indexing* (eine Vorabauswahl möglicher Programmklauseln anhand des Aufbaus der Argumente) und auf das in dieser Arbeit behandelte *Intelligente Backtracking*[2] (eine bessere Bestimmung von Zuständen, auf die die Berechnung beim Backtracking zurückgesetzt wird).

Funktionale Programmiersprachen wie Miranda oder Lisp besitzen Funktionen als atomare Objekte. Ein mit Hilfe von Abstraktion, Applikation und Rekursion definiertes Programm wird durch den

[1] **W**arren **A**bstract **M**achine, vgl. [Aït-Kaci 91]

[2] *Intelligentes Backtracking* ist in der einschlägigen Literatur ein feststehender Begriff. Deshalb wird er in dieser Arbeit ebenso wie der später auftretende Begriff *Naives Backtracking* groß geschrieben.

effizienten Mechanismus der Reduktion ausgewertet. Das Paradigma der funktionalen Programmierung steht auf einer sauberen mathematischen Basis. Dadurch ergeben sich für die Programme Eigenschaften wie Sicherheit und Verifizierbarkeit. Mit weitergehenden Konzepten wie Funktionen höherer Ordnung, polymorphen und abstrakten Datentypen erhält man kürzere und modularere Programme. Durch die Seiteneffektfreiheit („referential transparancy") sind bei der Auswertung funktionaler Programme automatisch Parallelisierungsmöglichkeiten gegeben.

Bei einer dritten Sparte der deklarativen Programmiersprachen handelt es sich um die Integration logischer und funktionaler Aspekte. Es wird versucht, die Ausdrucksstärke der logischen Programmierung mit der Auswertungseffizienz der funktionalen Programmierung zu verbinden. Man gibt dem Programmierer die Möglichkeit, an gewünschten Stellen (offensichtlichen) Determinismus zu formulieren, und an anderen Stellen den Nichtdeterminismus auszunutzen. Vertreter dieser Sparte sind z.B. K-Leaf, FunLog oder SLog. Ein weiterer Vertreter ist die in dieser Arbeit behandelte funktional-logische Sprache Babel, in der Funktionen mit logischen Bedingungen (*Guards*) definiert werden.
Das unidirektionale „Pattern-Matching" funktionaler Sprachen wird in Babel durch die bidirektionale Unifikation ersetzt. Unter anderem hat ein (lokaler) Berechnungsschritt Auswirkungen auf den gesamten Kontext der Berechnung. Bei der Auswertung kommt es wegen des in der Relation innewohnenden Nichtdeterminismus bei üblichen Suchstrategien immer wieder zum Zurücksetzen der Berechnung auf einen früheren Zustand. Dieses sogenannte Backtracking erfolgte bisher derart, daß die Berechnung immer bis zur letzten Auswahlstelle zurückgesetzt wird. Dieser so gefundene Berechnungspunkt muß aber nichts mit dem Scheitern der Berechnung zu tun haben, so daß eine alternative Regelauswahl an dieser Stelle zum wiederholten Scheitern führt. Erst nach dem erschöpfenden Versuch aller Alternativen einer Auswahlstelle kann die Berechnung zur nächsten tieferen Auswahlstelle zurückgesetzt werden. Dieses Verhalten bezeichnet man als Naives Backtracking.

In dieser Arbeit wird ein besseres Backtracking-Verfahren für die Auswertung von Programmen der funktional-logischen Sprache Babel entwickelt. Das Ziel ist eine Verringerung der Anzahl der Berechnungsschritte durch Einschränkung des Suchraumes. Eine Verbesserung der Auswertungseffizienz (Platz und Zeit) wird ebenfalls angestrebt, doch ist dies nicht immer zu erreichen. Denn die einzelnen Schritte einer Auswertung mit Intelligentem Backtracking sind aufwendiger als bei einer Auswertung mit Naivem Backtracking. Eine Steigerung der Ausführungseffizienz hängt vom Verhältnis zwischen eingespartem Suchraum und Mehraufwand der einzelnen Schritte ab.

2 Motivation und Literatur

Wie schon in der Einleitung angesprochen basiert die Auswertung eines Prolog-Programms auf der Unifikation (Parameterübergabemechanismus) und auf der Resolution als (vorwärtsgerichtete) Auswertungsmechanismen (vgl. [Schöning 89]). Aus Effizienzgründen wird üblicherweise als Auswertungsstrategie die Tiefensuche gewählt. Dabei werden die Klauseln und Literale in ihrer textuellen Reihenfolge im Programm (von oben nach unten bzw. von links nach rechts) probiert bzw. bearbeitet. Diese Auswertungsstrategie erfordert bei einem Fehlschlag der Berechnung (der Unifikation zwischen einem Zielliteral und dem Kopfliteral einer passenden Programmklausel), daß der Berechnungszustand zurückgesetzt und eine neue Klausel probiert wird. Diesen (rückwärtsgerichteten) Auswertungsmechanismus nennt man Backtracking. Genauer wird die Berechnung bis zur letzten (d.h. jüngsten) Stelle mit Alternativen für die Klauselauswahl zurückgesetzt. Doch diese so gefundene Auswahlstelle muß nichts mit dem oben angesprochenen Scheitern der Unifikation zu tun haben. Trotzdem fährt die (vorwärtsgerichtete) Berechnung an dieser Stelle fort und es kann wieder der gleiche Berechnungsfehler auftreten. „Generate-and-Test"–Programme veranschaulichen diese Tatsache besonders gut, wenn in einer frühen Generierungsphase der Grund für das mehrmalige Scheitern der Testphase liegt.
Hier setzt das Intelligente Backtracking an. Die Berechnung wird nicht „blind" bis zur jüngsten Auswahlstelle mit Alternativen zurückgesetzt (man bezeichnet dies in der Literatur als Naives Backtracking), sondern man versucht zu der Auswahlstelle zurückzugehen, die am Scheitern der Unifikation „eine Mit-

schuld trägt“. D.h. diese Auswahlstelle ist die jüngste in der Berechnung, die das Scheitern der Unifikation evtl. beheben könnte, würde eine alternative Programmklausel an dieser Stelle der Berechnung versucht.

Der nächste Abschnitt motiviert und erläutert kurz anhand von zwei Beispielen das Intelligente Backtracking bei der Auswertung von Prolog-Programmen.

2.1 Motivation des Intelligenten Backtrackings (IB)

Wie der Anfang des Abschnitts es schon andeutet, orientiert sich die Arbeit sehr stark am prozeduralen Verhalten von Prolog-Programmen (später auch Babel-Programme). Der Zustand einer Berechnung eines Prolog-Programms (genauer einer Anfrage an ein Prolog-Programm) wird durch einen Berechnungsplan und einen Unifikationsgraphen dargestellt, die beide im Abschnitt 3 definiert werden. Der Berechnungsplan stellt die bisher ausgeführte Berechnung mit Klauselauswahlen und noch vorhanden Alternativen dar, während die Substitution, die durch die bisherigen Unifikationen aufgebaut wurde, im Unifikationsgraphen notiert wird.

Ein Beispiel für die graphische Darstellung eines Berechnungsplanes und eines Unifikationsgraphen gibt Abbildung 1. Ein Berechnungsplan ist ein Baum, dessen Knoten im offenen (dies bedeutet unbearbeiteten) Zustand graphisch als Halbovale und im geschlossenen (dies bedeutet bearbeiteten) Zustand als volle Ovale dargestellt sind. Innerhalb des Ovals befindet sich im oberen Teil ein zu bearbeitendes Ziellliteral (Anfrage oder Literal einer rechten Klauselseite), während bei geschlossenen Knoten im unteren Teil der Klauselkopf der angewandten Regel enthalten ist. Weitere Komponenten eines Knotens sind seine Nummer (links oben), die Menge der berechneten Backtracking-Punkte (rechts oben,) und die Menge der noch vorhandenen Alternativklauseln (rechts unten). Die Menge der Backtracking-Punkte enthält Nummern von geschlossenen Knoten (fett geschrieben), wohingegen die Menge von Alternativklauseln Nummern der implizit nummerierten Programmregeln enthält.
Die vorgenommenen Bindungen während der Berechnung einer Anfrage an ein Prolog-Programm werden außerhalb des Berechnungsplanes in einem Unifikationsgraphen notiert. Ein Unifikationsgraph besitzt als Knotenmenge eine Menge von Klassen, deren funktionale Abhängigkeiten durch die Kanten des Unifikationsgraphen festgehalten werden. In der graphischen Darstellung werden die Kanten durch die entsprechende Position des Argumentes des Ursprungs markiert. Die Klassen selber bestehen aus Graphen (mit Kanten ohne Markierung) und aus einer Label-Menge. Diese Label-Menge enthält die Nummern derjenigen Knoten des zugehörigen Berechnungsplanes, die an den vorgenommenen Bindungen innerhalb der Klasse beteiligt sind. Graphisch wird die Label-Menge einer Klasse neben dem Repräsentanten dieser Klasse (eindeutiger Knoten, der von allen Elementen der Klasse erreicht werden kann bzw. zu dem alle Knoten der Klasse dereferenziert werden) notiert. Genauere Erläuterungen und Berechnungsvorschriften werden im Abschnitt 3 angegeben.

Zur Motivation des Intelligenten Backtrackings folgen zwei Beispiele. Das erste Beispiel beschreibt die Durchführung unnötiger Berechnungsschritte bei der Anwendung von Naivem Backtracking: Der durch das Paar $\langle P_4, UG_4 \rangle$ der Abbildung 1 dargestellte (vierte) Berechnungszustand wird durch sukzessive Anwendung der Programmregeln `5),1),2)` und `3)` erreicht. Der zugehörige Unifikationsgraph enthält vier Knoten (Klassen) und zwei Kanten. Das nächste zu bearbeitende Zielliteral ist `n(Y')`, wobei die Programmregel mit der Nummer `4)` die einzige Regel ist, die das Prädikat `n/1` definiert. Die Anwendung der Regel `4)` auf das Zielliteral `n(Y')` scheitert, da (wie im Unifikationsgraphen dargestellt) `Y'` an `f(a)` gebunden ist. Das Fehlen weiterer Alternativklauseln bewirkt ein Zurücksetzen der Berechnung durch Backtracking.
Üblicherweise wird in einem Prolog-System Naives Backtracking realisiert: Die Berechnung wird bis zur jüngsten Auswahlstelle mit Alternativen zurückgesetzt. Im Beispiel wäre das der Knoten mit der Nummer `3`. Doch dieser Knoten kann ein Scheitern der Berechnung an gleicher Stelle nicht verhindern, so daß es zu einem „oszillierenden“ Verhalten kommt. Erst wenn alle Alternativen des Knotens mit der Nummer `3` ausprobiert wurden erfolgt Backtracking zum Knoten mit der Nummer `2`.
Intelligentes Backtracking versucht nun den Suchraum einer Auswertung um gerade diese als sinnlos

zu erkennenden Bereiche zu verkleinern. Denn mit Hilfe des Unifikationsgraphen ist erkennbar, daß der Knoten mit der Nummer 3 nicht an der Bindung von `Y'` an `f(a)` beteiligt ist. Somit ist eine alternative Regelwahl an diesem Knoten für das darauf folgende Scheitern unerheblich.

Beispiel

1) q(f(X),Z,Z) :- r(X).
2) r(a).
3) p(Y,a).
1') q(a,b,c).
2') r(b).
3') p(Y,b).
4) n(f(b)).
3") p(V,V) :- r(V).
5) goal(X',Y',W') :- q(X',X',Y'), p(g(Y'),W'), n(Y').

?- goal(X",Y",W").

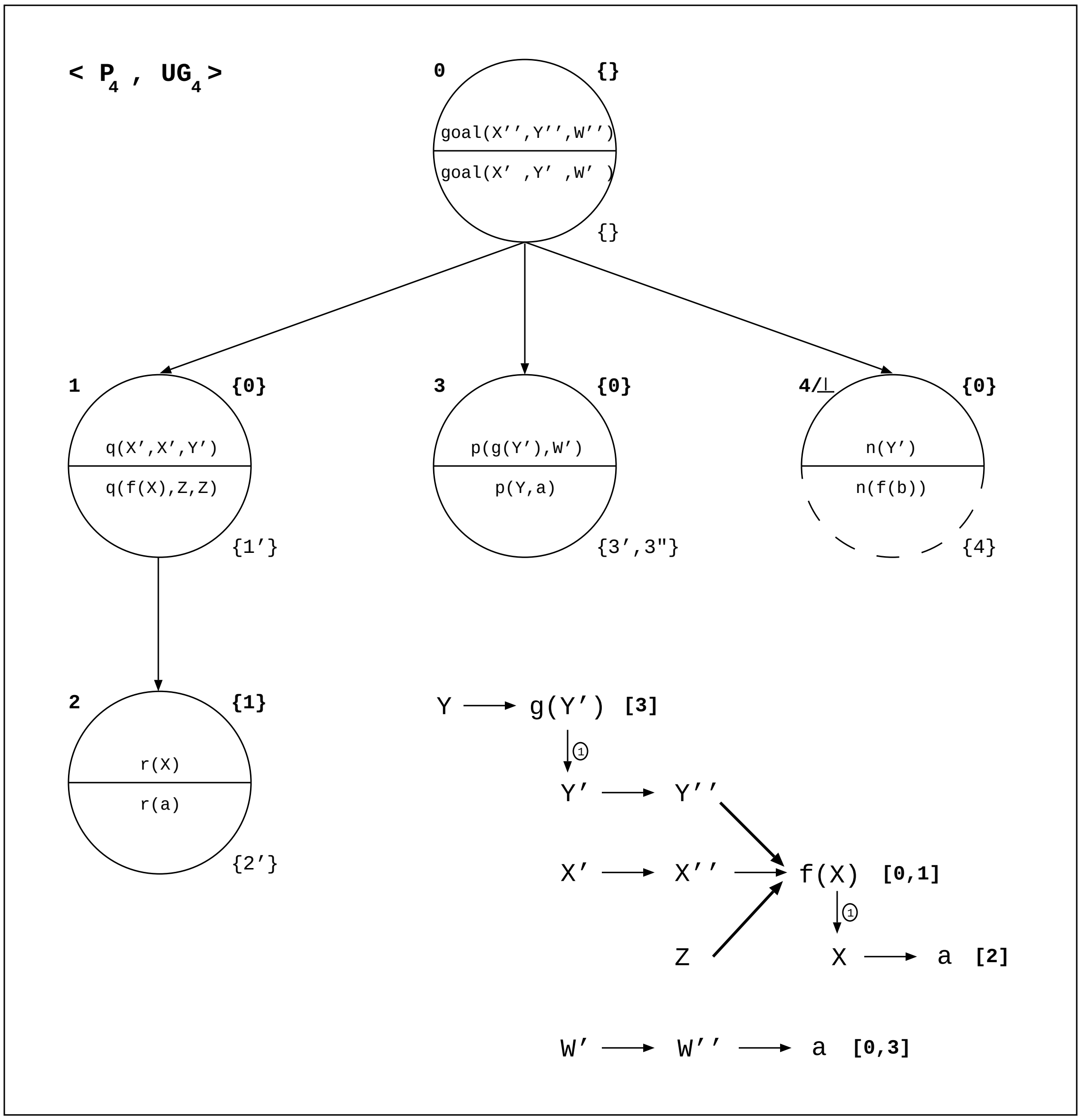


Abbildung 1: Ein Beispiel für unnötige Berechnungsschritte

Das zweite Beispiel in Abbildung 2 zeigt, daß durch die Einschränkung des Suchraumes um unendliche Bereiche eine mit Naivem Backtracking nicht terminierende Berechnung mit Intelligentem Backtracking terminieren kann. Das Beispiel zeigt den Versuch, eine mit dem Element `a` beginnende Liste um zwei weitere Listen `W` und `X` so zu verlängern, daß eine mit `b` beginnende Liste entsteht.

Die Abbildung deutet an, daß am Knoten mit der Nummer `3` beide Regeln für das Prädikat `app` nicht angewendet werden können. Naives Backtracking setzt die Berechnung bis zur jüngsten (und einzigen) Alternativstelle am Knoten mit der Nummer `2` zurück. Wieder kommt es zu einem „oszillierenden“ Verhalten, wobei anschaulich gesprochen immer neue Zwischenlisten eingeführt werden. Das im weiteren Verlauf dieses Abschnitts vorgestelle IB-Verfahren erkennt, daß am Scheitern der Berechnung am Knoten mit der Nummer `3` nur die Knoten mit den Nummern `0` und `1` beteiligt sind. Intelligentes Backtracking zu diesen Knoten bewirkt einen Abbruch der Berechnung mit einer negativen Erfüllbarkeitsaussage.

Beispiel

1) app([],L',L').
1') app([E|R],L,[E|RL]) :- app(R,L,RL).
2) goal(V,W,X,Y,Z) :- app([a|V],W,Z), app(Z,X,[b|Y]).

?- goal(V',W',X',Y',Z').

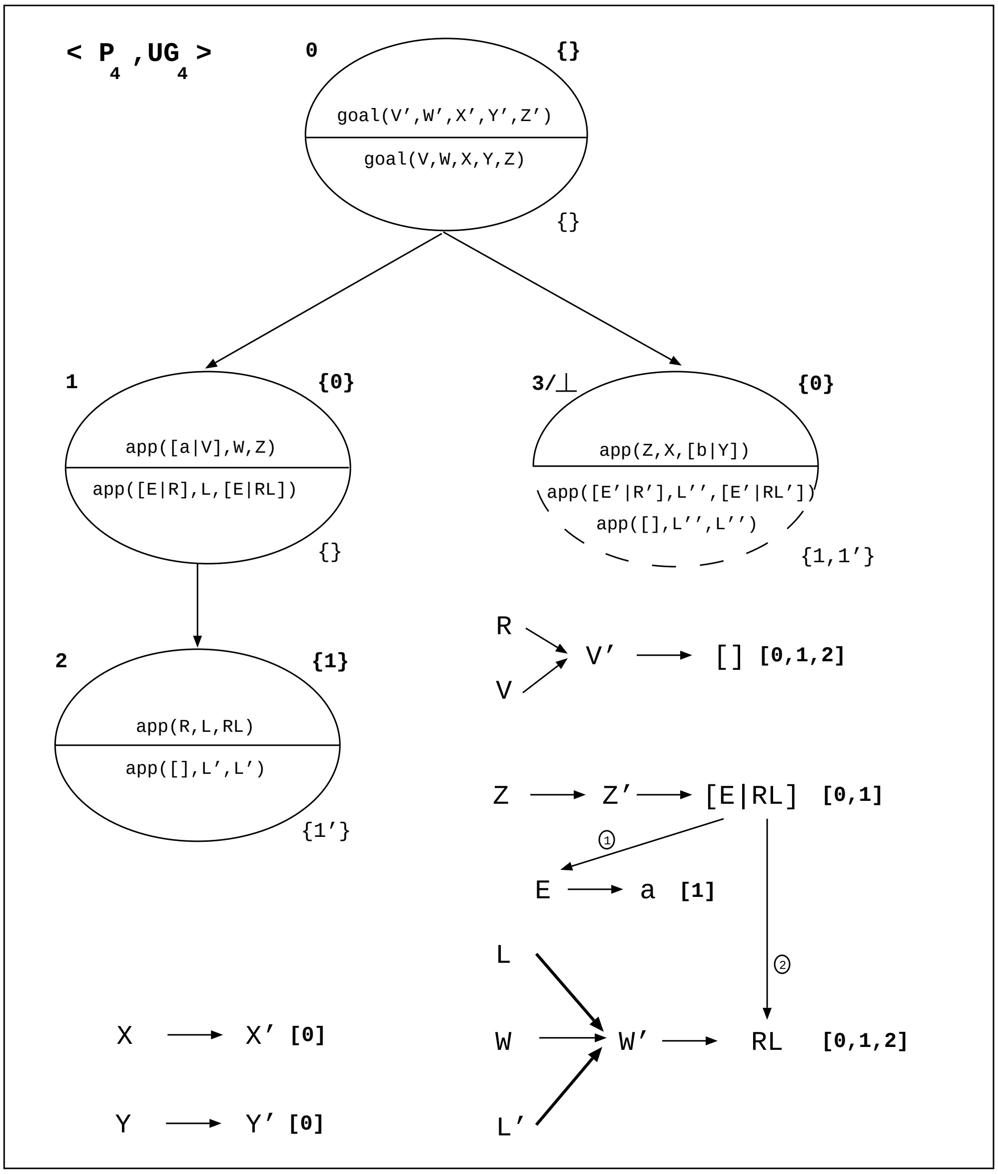


Abbildung 2: Ein Beispiel für nichtterminierende Berechnungen

Vor der Darstellung des IB-Verfahrens für Prolog gibt der nächste Abschnitt einen Überblick über die Entwicklung des Intelligenten Backtrackings (bzgl. der Logik-Programmierung) in der Literatur.

2.2 Entwicklung des Intelligenten Backtrackings in der Literatur

Einer Maschine, die ein Prolog-Programm ausführen und eine Anfrage berechnen kann, liegt meistens die WAM[3] zugrunde. Das blinde Vorgehen beim Zurücksetzen der Berechnung durch Naives Backtracking wird auf die WAM übertragen. Es wird nicht zwischen dem (lokalen) Versuch einer weiteren Alternative („shallow backtracking") und dem tieferen Zurücksetzen der Berechnung („deep backtracking") unterschieden: Dem Scheitern eines Berechnungsschrittes folgt immer das Zurücksetzen der Berechnung bis zur letzten (d.h. jüngsten) Auswahlstelle mit noch nicht versuchten Alternativen. Diese uniforme (und etwas „mechanische") Behandlung des Backtrackings wurde in der WAM in geeigneter Weise unterstützt. Doch schon Mitte der siebziger Jahre wurde nach Alternativen zur erschöpfenden Abarbeitung des Suchraumes gesucht. Denn die Anwendung der nächsten Alternative verhindert nicht immer ein wiederholtes Scheitern der Berechnung. Es könnten Berechnungsschritte vermieden werden, die aufgrund der bis zum Zeitpunkt des Scheiterns durchgeführten Berechnung als nicht erfolgreich zu erkennen sind.

Ein Versuch der Verbesserung des Auswertungsverhaltens ist das Indexing. Soll an einem Berechnungspunkt eine alternative Klausel versucht werden, so wird nicht einfach die (textuell) nächste Klausel probiert, sondern durch Indexing wird die nächste Klausel bestimmt, für die die Unifikation des Zielliterals mit ihrem Kopf aufgrund der bisher durchgeführten Unifikationsversuche noch nicht als vergeblich erkannt wurde. Indexing betrifft also die Vermeidung unnötiger Unifikationsversuche beim „shallow backtracking". Durch Indexing kann aber insbesondere nicht eine unendliche Berechnung verhindert werden.

Ein weiterer Versuch ist das in dieser Arbeit beschriebene Intelligente Backtracking, welches in gewisser Weise orthogonal zum Indexing ist: Beim „deep backtracking" wird die Berechnung nicht (nur) bis zur letzten Auswahlstelle mit Alternativen zurückgesetzt, sondern bis zu einer Stelle der Berechnung, bei der eine alternative Klauselwahl das (wiederholte) Scheitern eines Berechnungsschrittes verhindern könnte. Im Gegensatz zum eher lokalen Charakter des Indexings stellt das Intelligente Backtracking einen (bzgl. des Berechnungspunktes) globalen Versuch dar, den Suchraum zu verkleinern. Durch Intelligentes Backtracking lassen sich viele unendliche Berechnungen, die durch die mechanische Auswahl von Alternativen beim Naiven Backtracking begründet sind, vermeiden. Wie das Beispiel 2 zeigt, muß an der Unendlichkeit dieser Berechnungen nicht nur die unvollständige Suchstrategie (Tiefensuche) Schuld sein. Doch sicherlich wird meistens die Tiefensuche in Verbindung mit Naivem Backtracking als Ursache zu erkennen sein.

Ein Intelligentes Backtracking-Verfahren besteht immer aus zwei Komponenten :

1. Beschaffung und Verwaltung von Informationen insbesondere durch eine Erweiterung der Unifikation

2. Analyse des Scheiterns eines Berechnungsschrittes und Bestimmung geeignter Backtracking-Punkte (Punkte, auf die die Berechnung zurückgesetzt wird) mit Hilfe der gesammelten Informationen

Je genauer das IB-Verfahren ist, desto mehr Informationen werden benötigt und desto aufwendiger ist die Bestimmung geeigneter Backtracking-Punkte. D.h. es besteht allgemein ein Widerspruch zwischen eingespartem Suchraum und benötigtem Aufwand zur Laufzeit. Die Informationen betreffen nicht nur wie üblich die Kenntnis, an welchen Term eine Variable gebunden wurde, sondern z.B. ist auch der Zeitpunkt der Bindung für das Intelligente Backtracking sehr wichtig. Zu diesem Zweck wurden schon viele erweiterte Unifikationsalgorithmen vorgeschlagen, wie z.B. in [Bruynooghe, Pereira 84], [Kumar, Lin 87], [Cox 87], [Codognet et al. 88] und [Malhotra 90].
Das primäre Ziel dieser Arbeit ist die Einschränkung des Suchraumes durch Intelligentes Backtracking bei der Auswertung von Babel-Programmen. Daneben besteht aber auch das (sekundäre) Ziel der Verkürzung der Auswertungszeit, so daß auch in dieser Arbeit ein Kompromiß zwischen der Genauigkeit des zu entwickelten Verfahrens und der Einschränkung des dadurch verursachten Mehraufwandes gemacht werden muß.

[3] **W**arren **A**bstract **M**achine, vgl. [Aït-Kaci 91]

Der Rest dieses Abschnittes ist der Geschichte des Intelligenten Backtrackings gewidmet.
Mitte der siebziger Jahre gab es im wesentlichen zwei Gruppen, die sich mit Intelligentem Backtracking bei sequentiellen Strategien beschäftigten. Die Ansätze der beiden Gruppen sind in gewisser Weise orthogonal zueinander, aber dennoch eng verwandt.

Die belgisch-portugiesische Gruppe bestand aus Maurice Bruynooghe, Luís Pereira und António Porto. Sie untersuchten Intelligentes Backtracking für reine Hornklauselprogramme, wobei sie ähnlich wie in dieser Arbeit die Berechnung mit „Deduktionsbäumen" darstellten (vgl. [Bruynooghe, Pereira 84]). Ein Deduktionsbaum beschreibt eine Menge von Paaren bestehend aus einem (Ziel-) Literal und dem Kopf einer Klausel, die alle paarweise unifiziert werden sollen. Scheitert die Unifikation, so werden minimale, nicht unifizierbare Teilmengen betrachtet, die Teilbäumen des Deduktionsbaumes entsprechen.
[Pereira, Porto 80] enthält einen theoretischen und einen praktischen (nicht so präzisen, aber besser implementierbaren) Ansatz des Intelligenten Backtrackings. Porto erstellte 1980 auf der Grundlage dieses Ansatzes einen ersten intelligent backtrackenden Prolog-Interpreter (in Prolog).

Die zweite Gruppe der Pioniere des Intelligenten Backtrackings bestehend aus Cox, Matwin und Pietrzykowski schränkte ihren Arbeitsbereich nicht auf Hornklauselprogramme ein. Sie betrachteten Intelligentes Backtracking allgemein für die Logik-Programmierung. Hintergrund waren mechanische Theorembeweiser, die sich auf die Logik-Programmierung stützen. Ähnlich zur ersten Gruppe stellten sie Auswertungszustände in „Deduktionsplänen" dar. Aber sie betrachteten beim Scheitern eines Auswertungsschrittes maximale, unifizierbare Teilpläne, welche sie als Ausgangsbasis für weitere Auswertungsschritte gebrauchten. [Matwin, Pietrzykowski 82] und [Pietrzykowski, Matwin 82] basieren auf früheren Arbeiten von Cox und werden in [Matwin, Pietrzykowski 85] in einer verfeinerten Form dargestellt. Die Arbeiten von Cox sind in [Cox 84] und [Cox 87] beschrieben.

Beiden Ansätzen ist gleich, daß sie desto exakter sind, je kleiner die minimalen, nicht unifizierbaren Teilmengen bzw. je größer die maximalen, unifizierbaren Teilmengen sind. Z.B. betrachtet [Cox 87] alle maximalen, unifizierbaren Teilmengen, und [Matwin, Pietrzykowski 82] zeigt die Möglichkeit einer exponentiellen Einschränkung des Suchraumes.
Die praktische Relevanz dieser Ansätze wird durch [Wolfram 86] relativiert: Eine Menge mit n Paaren kann 2^n minimale, nicht unifizierbare bzw. maximale, unifizierbare Teilmengen besitzen, und die Suche nach der kleinsten minimalen, nicht unifizierbaren, bzw. der größten maximalen, unifizierbaren Teilmenge ist nicht immer mit polynomialem Aufwand möglich.

Neuere Arbeiten orientieren sich stärker an der praktischen Verwendbarkeit und schenken dem Aufwand des IB-Verfahrens mehr Aufmerksamkeit. Es folgt eine kurze Darstellung dreier verschiedener Ansätze.

[Chang, Despain 85] beschreibt ein Backtracking-Verfahren, welches die Autoren als Semi-Intelligent bezeichnen. Vor der eigentlichen Übersetzung eines Prolog-Programms wird eine statische Analyse der Datenabhängigkeiten durchgeführt. Dem liegt die Vorstellung zugrunde, daß zwei Literale unabhängig voneinander sind, falls sie keine gemeinsamen ungebundenen Variablen teilen. Pro Klausel wird ein Abhängigkeitsgraph der Literale erstellt und zur Code-Erzeugung benutzt.
Diese statische Analyse benötigt Angaben über Prädikatssymbole, die in einer Anfrage an das Programm enthalten sind, und über die Beschaffenheit der Argumente („entry-declarations"). Der Vorteil dieses Ansatzes ist die geringe Laufzeitbelastung des Verfahrens. Doch leider ist das Intelligente Backtracking klauselbeschränkt, so daß in den meisten Fällen der mit Hilfe der *worst-case*-Analyse erzeugte Code mit dem üblichen, naiv backtrackenden Code identisch ist.

Eine dynamische Analyse der Datenabhängigkeiten schlagen [Kumar, Lin 87] und [Kumar, Lin 88] vor: Nach dem erschöpfenden Versuch aller Alternativen einer Auswahlstelle („deep failure") wird vor dem „deep backtracking" eine Analyse eines „Abhängigkeitsgraphen" durchgeführt und festgestellt, welche Teile der Berechnung am Scheitern beteiligt gewesen sein könnten (in gewisser Weise eine dynamische *worst-case*–Analyse). Das Verfahren kann als vereinfachte Version von [Bruynooghe, Pereira 84] betrachtet werden.

Als letzten Ansatz des Intelligenten Backtrackings für Prolog-Programme soll ein französischer

dienen. In [Codognet et al. 88] wird ein IB-Verfahren vorgestellt, welches bei jedem einzelnen Scheitern einer Unifikation eine Analyse durchführt und die beteiligten Berechnungspunkte aufsammelt. Dazu muß eine komplexe Datenstruktur verwaltet werden, die ihren Ursprung in [Pietrzykowski, Matwin 82] hat. Die Korrektheit und Vollständigkeit wird in [Codognet 89a] bewiesen.
Das Verfahren kann in vielfältiger Form vereinfacht werden, was zwar die Genauigkeit reduziert, aber auch den Aufwand (u.U. erheblich) senkt (s. auch [Codognet 89b] und [Codognet, Sola 91]).
Der von den Codognets vorgetellte Ansatz bildet auch die Grundlage des in Abschnitt 3 vorgestellten Verfahrens, bei dem in geeigneter Weise Vereinfachungen von [Codognet 89a] vorgenommen wurden.

Der Rest der Arbeit gliedert sich wie folgt: Der dritte Abschnitt beschreibt ein IB-Verfahren für Prolog. Die funktional-logischen Sprache Babel wird im vierten Abschnitt vorgestellt, bevor der fünfte Abschnitt das Zusammenspiel von Intelligentem Backtracking und Babel diskutiert.
Nach der genauen Definition eines IB-Verfahrens im sechsten Abschnitt schließt die Arbeit mit der Präsentation von Laufzeitergebnissen.

3 Ein IB-Verfahren für Prolog

In diesem Abschnitt wird ein Auswertungsverfahren für Prolog-Programme mit Intelligentem Backtracking vorgestellt, welches eine Vereinfachung des Ansatzes aus [Codognet 89a] ist und viele Gemeinsamkeiten mit dem in [Codognet et al. 88] beschriebenen Verfahren besitzt. In dieser Arbeit können aus Platzgründen nur die wesentlichen Grundzüge des Verfahrens dargestellt werden. Der interessierte Leser sei für weitere Informationen auf die oben zitierten Arbeiten und [Koch 92] verwiesen. Als Auswertungsstrategie wird wie üblich die Tiefensuche gewählt.

Die wesentlichste Struktur des vorzustellenden IB-Verfahrens ist der Unifikationsgraph. Zusammen mit dem Berechnungsplan wird durch ihn der Zustand der Berechnung beschrieben und der Fortgang der Berechnung definiert. Ein Berechnungsschritt besteht bei erfolgreicher Bearbeitung des aktuellen Berechnungspunktes aus einer (um die Verwaltung spezieller Informationen) erweiterten Unifikation. Kann der Berechnungspunkt nicht erfolgreich bearbeitet werden, so liefert der Unifikationsalgorithmus entsprechende Informationen und Backtracking setzt ein.

3.1 Die Sprache Prolog

Die Orientierung der Arbeit am prozeduralen Verhalten eines Prolog-Programms erfordert die Kenntnis grundlegender Aspekte der Logik-Programmierung. Zum weiteren Verständnis werden einige Begriffe der Logik-Programmierung vorausgesetzt (vgl. [Clocksin, Mellish 87]):

- Jedem Prolog-Programm liegt eine Signatur $\Sigma = \langle \Omega, \Pi \rangle$ (Operations- und Prädikatssymbole) und eine Menge von Variablen Var zu Grunde. Darüber definiert man die Menge der Σ-Terme $T_\Sigma(Var)$ und die Menge der Σ-Formeln $WFF_\Sigma(Var)$. Im folgenden werden insbesondere Literale (atomare Formeln) als Bestandteile von Klauseln betrachtet. $Lit_\Sigma(Var)$ bezeichnet die Menge aller Literale.

- Ein Prolog-Programm ist eine Folge definiter Hornklauseln (Disjunktion eines positiven und beliebig vieler negativer Literale). Eine Programmklausel ohne negative Literale heißt Fakt, sonst Regel.

- Eine Anfrage an ein Prolog-Programm ist eine Konjunktion positiver Literale. Mit ihr wird ein Prolog-Programm aufgerufen. Die Berechnung erfolgt als Widerspruchsbeweis mit Hilfe der negierten Anfrage (Zielklausel) durch Unifikation und Resolution.

- Ist der Widerspruchsbeweis der Zielklausel erfolgreich, so erhält man als Ergebnis der Anfrage G eine Erfüllbarkeitsaussage und eine Substitution σ der in G vorkommenden Variablen (Antwortsubstitution). $G\sigma$ nennt man auch Rechenergebnis.

3.2 Der Berechnungsplan

Die für die Darstellung der Berechnung einer Anfrage an ein Prolog-Programm üblicherweise benutzten SLD-Berechnungsbäume sind für die Zwecke des Intelligenten Backtrackings zu ungenau. Ein einziger Knoten eines SLD-Baumes beschreibt den gesamten Zustand einer Berechnung, wobei der SLD-Baum der gesamten Berechnung entspricht. Viele Informationen sind nur implizit durch das Programm gegeben (noch vorhandene Alternativen, gescheiterte Unifikationsversuche, etc.).
Aus diesem Grund werden für die übersichtliche und dennoch genaue Darstellung die in diesem Abschnitt definierten Berechnungspläne benutzt. Ein Berechnungsplan ist ebenfalls ein Baum und gibt zusammen mit den im nächsten Abschnitt eingeführten Unifikationsgraphen den Zustand der Berechnung zu einem bestimmten Zeitpunkt wieder. Die gesamte Berechnung ist als Folge solcher Pläne und Graphen zu verstehen.

Definition 3.1 (Prolog-Berechnungsplan) Ein Berechnungsplan P (kurz Plan) zu einem Prolog-Programm S und einer Zielklausel G ist ein Ableitungsbaum und stellt eine Folge von Resolutionsschritten dar. Die Knoten des Baumes (Berechnungspunkte) sind 5-Tupel der Form

$$\langle n, l_{up}, l_{down}, Alt, BP \rangle \in (\mathbb{N} \cup \{\perp\}) \times Lit_\Sigma(Var) \times (Lit_\Sigma(Var) \cup \{\perp\}) \times 2^{\mathbb{N}} \times 2^{\mathbb{N}}$$

wobei $2^{\mathbb{N}}$ die Potenzmenge der naürlichen Zahlen ist. Die einzelnen Komponenten haben folgende Bedeutung:

- n ist die Nummer der ausgeführten Unifikation (die Nummer des Berechnungspunktes, auch Label genannt), falls $l_{down} \neq \perp$. Sonst ist ebenfalls $n = \perp$ (d.h. $n \in \mathbb{N}$, falls der Knoten geschlossen ist; s.u.).
- l_{up} ist das zu resolvierende Literal, wenn die Berechnung zu diesem Knoten von P gelangt.
- $l_{down} \neq \perp$ ist der Kopf einer Prolog-Klausel, mit der l_{up} resolviert wurde, falls die Berechnung den Knoten schon erreicht hat ($n \neq \perp$). Erreichte die Berechnung noch nicht den Knoten, so ist $l_{down} = \perp$.
- Alt (Alternativen) ist eine Menge von Programmklauseln, deren Kopfprädikatsymbol mit dem Prädikatsymbol von l_{up} identisch ist (man geht von einer impliziten Durchnummerierung der Programmklauseln aus).
- BP (Backtracking-Punkte) ist eine Menge von Berechnungspunkten (Knotennummern n), die als geeignet für das Zurücksetzen der Berechnung erkannt wurden. Wird die Berechnung von einem Knoten $v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ zurückgesetzt, so genau zu einem Element der Menge BP (und zwar zum Maximum).

Ein Knoten des Berrechnungsplanes heißt offen, wenn die Berechnung diesen Punkt noch nicht erreicht hat. Er hat die Form $\langle \perp, l_{up}, \perp, Alt, BP \rangle$. Ansonsten heißt der Knoten geschlossen. Besitzt ein Plan nur geschlossene Knoten, so ist er komplett.
Die geschlossenen Knoten des Berechnungsplanes werden fortlaufend gemäß der Auswertungsstrategie (d.h. depth-first left-to-right) durchnummeriert.
Der initiale Berechnungsplan P_0 zu einer Anfrage G = ?- $l_1, \ldots, l_r$. ist ein Baum, dessen Knotenmenge aus der Wurzel $\langle 0, goal(X_1', ..., X_k'), goal(X_1, ..., X_k), \emptyset, \emptyset \rangle$ und aus den r Blättern $\langle \perp, l_1, \perp, alt(l_1), \{0\} \rangle$, $\ldots$, $\langle \perp, l_r, \perp, alt(l_r), \{0\} \rangle$ besteht. Dabei ist $\{X_1, ..., X_k, X_1', ..., X_k'\} \subseteq Var$ mit $l_i \in Lit_\Sigma(\{X_1, ..., X_k\})$, $1 \leq i \leq r$, und die Funktion $alt : Lit_\Sigma(Var) \longrightarrow (2^{\mathbb{N}} \setminus \{\emptyset\})$ bildet ein Literal $p(t_1, \ldots, t_s)$ auf die Menge der Programmklauseln ab, die die Prozedur p/s definieren.
$\mathcal{PL}$ bezeichnet die Menge aller Prolog-Berechnungspläne. □

Bemerkung Formal kann man zu jedem Programm S mit Anfrage G = ?- $l_1, \ldots, l_r$. ein normiertes Programm S' angeben, welches zusätzlich zu den m Klauseln aus S die $(m+1)$-te Klausel

$goal(X_1, ..., X_k)$:- $l_1, \ldots, l_r$. mit neuem Prädikatssymbol $goal/k$ besitzt. Die entsprechende Anfrage lautet dann G' = ?- $goal(X'_1, ..., X'_k)$, und der initiale Berechnungsplan besteht nur aus dem offenen Knoten $\langle \perp, goal(X_1, ..., X_k), \perp, \{m+1\}, \emptyset \rangle$. Nach dem ersten Ableitungsschritt (vgl. Def. 3.4) ist der Plan dann identisch mit dem oben definierten initialen Plan. Die Prolog-Programme in dieser Arbeit sind alle normiert.
Offene Knoten können nur Blätter des Berechnungsplanes sein. □

Die folgende Abbildung gibt die graphische Darstellung eines offenen und eines geschlossenen Berechnungspunktes (Knoten eines Berechnungsplanes) an. Vollständige Berechnungspläne sind in den Abbildungen 1 und 2 enthalten.

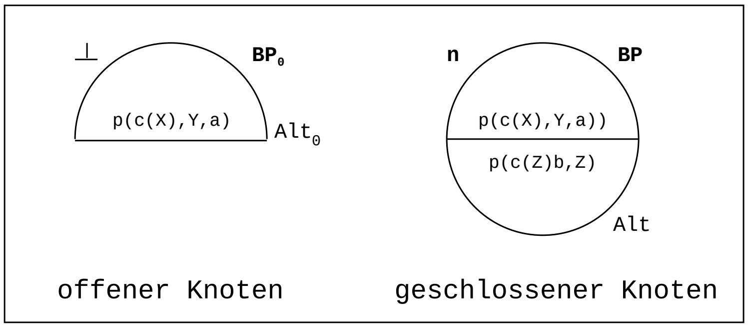


Abbildung 3: Graphische Darstellung von Berechnungspunkten

Eine Berechnung ist beendet, falls alle Knoten des Berechnungsplanes geschlossen sind. Der Berechnungsplan stellt nur das Skelett der Berechnung dar und gibt Auskunft über die vorgenommene Klauselwahl (geschlossene Knoten) und über bestehende Alternativen für diese Wahl. Das eigentliche Schließen eines Knotens, welches der Unifikation des Zielliterals l_{up} und dem Kopfliteral einer Programmklausel unter Beachtung bisheriger Unifikationen entspricht, wird im Unifikationsgraphen definiert und notiert.

3.3 Der Unifikationsgraph

Der Unifikationsgraph enthält neben den Informationen, welche Variablenbindungen vorgenommen wurden, auch die Zeitpunkte der Bindungen. Seine Struktur ist ein gerichteter Graph, dessen Knoten (Klassen genannt) selber gerichtete Graphen sind.

Definition 3.2 (Klasse) Eine Klasse $c = \langle V, E, Lab \rangle$ setzt sich zusammen aus einem gerichteten azyklischen Graphen $\langle V, E \rangle$ mit der Knotenmenge $V \subseteq T_\Sigma(Var)$ und mit der Kantenmenge $E \subseteq V \times V$, und aus einer Label-Menge $Lab \subseteq \mathbb{N}$, die eine Menge von Nummern geschlossener Knoten eines bestimmten Berechnungsplanes ist.
Jede Klasse eines Graphen besitzt einen ausgezeichneten Knoten, zu dem es von jedem anderen Knoten der Klasse aus einen Pfad gibt. Er heißt Repräsentant der Klasse.
Zu jeder Klasse c gibt es die Funktion $deref_c : V \longrightarrow V$, die einen Knoten der Klasse auf den Repräsentanten der Klasse abbildet. □

Bemerkung Ein Term, der keine Variable ist, wird im weiteren Text *Struktur* genannt. Die Knotenmenge einer Klasse besteht aus beliebig vielen Variablen und aus höchstens einer Struktur. Enthält die Knotenmenge einer Klasse solch eine Struktur, dann muß diese ihr Repräsentant sein .
Der Repräsentant einer Klasse kann nie Ursprung einer Kante seiner Klasse sein (azyklischer Graph).
$deref_c(v)$ ist das von v dereferenzierte Objekt.
Man spricht von der Label-Menge eines Knotens und meint damit die Label-Menge der Klasse des Knotens, d.h. der Klasse, deren Knotenmenge den besagten Knoten enthält.
$\bar{t}$ bezeichnet die Klasse, deren Knotenmenge t enthält. □

In Abbildung 1 sind beispielsweise vier Klassen enthalten. Die Klasse mit der umfangsreichsten Knotenmenge besteht aus den sechs Knoten `X',X'',Y',Y'', Z` und `f(X)`, den fünf (unbeschrifteten) Kanten und der Label-Menge `[0,1]`, welche neben dem Repräsentanten der Klasse notiert ist. Die Kanten einer Klasse entsprechen den vorgenommenen Bindungen.

Funktionale Abhängigkeiten zwischen den Knoten zweier Klassen werden im Unifikationsgraphen notiert, dessen Knotenmenge gerade die Klassen und dessen Kanten diese funktionalen Abhängigkeiten bezeichnen.

Definition 3.3 (Prolog-Unifikationsgraph) Ein Unifikationsgraph UG ist ein gerichteter azyklischer Graph $\langle C, F\rangle$, dessen Knotenmenge C aus einer Menge von Klassen besteht. Die Kanten stellen funktionale Abhängigkeiten zwischen zwei Knoten zweier verschiedener Klassen dar (nicht zwischen zwei Klassen). Zur Darstellung gibt es zu jedem Unifikationsgraphen eine Markierungsfunktion $mark_{UG} : F \longrightarrow \mathbb{N} \times T_\Sigma(Var) \times T_\Sigma(Var)$, die eine Kante $f \in F$ auf ein Tripel bestehend aus den Komponenten Argumentposition des Ursprungsterms, Ursprungs- und Zielterm abbildet. Der Ursprungsterm solch einer Abhängigkeit muß eine Struktur (und somit Repräsentant seiner Klasse) sein.
Ein Unifikationsgraph wird durch eine Folge allgemeinster Unifikatoren erstellt und gibt die Komposition dieser wieder. Deshalb gibt es zu jedem Unifikationsgraphen UG eine Funktion $subst_{UG} : T_\Sigma(Var) \longrightarrow T_\Sigma(Var)$, die den Wert (d.h. die aktuelle Bindung) eines jeden Klassenknotens angibt (s. Bemerkung).
Der initiale Unifikationsgraph UG_0 einer Berechnung (für ein normiertes Programm) ist der leere Graph ohne Knoten (und Kanten) $UG_0 = \langle\emptyset, \emptyset\rangle$.
$\mathcal{UG}$ bezeichnet die Menge aller Unifikationsgraphen. □

Bemerkung Die Schreibweise $F \subseteq C \times C$ wurde vermieden, da es parallele Kanten (bzgl. Ursprungs- und Zielklasse) geben kann.
$subst : \mathcal{UG} \longrightarrow Subst$ bildet einen Unifikationsgraphen auf die entsprechende, durch ihn repräsentierte Substitution ab ($Subst = \{\sigma \mid \sigma : T_\Sigma(Var) \rightarrow T_\Sigma(Var)\}$; Schreibweise $subst_{UG}(t) = t'$). Dies ist möglich, da jeder Unifikationsgraph azyklisch ist. Sei $subst(UG) = \sigma$. Dann kann man σ wie folgt induktiv über die Struktur des Unifikationsgraphen konstruieren: Ist der Repräsentant einer Klasse eine Variable oder eine Konstante, so bildet σ jedes Element t der Knotenmenge der Klasse auf diesen Repräsentanten ab. Ist andererseits der Repräsentant einer Klasse c eine Struktur ($deref_c(t) = g(t_1, \ldots, t_k)$ für alle Knoten t der Klasse c), so kann man induktiv über die Struktur $t\sigma = g(t_1\sigma, \ldots, t_k\sigma)$ definieren, falls $UG = \langle C, F\rangle$ und $\forall\ 1 \leq i \leq k\ \exists f \in F\ :\ mark_{UG}(f) = (i, g(t_1, ..., t_k), t_i)$. □

In den Abbildungen 1 und 2 ist jeweils ein Unifikationsgraph enthalten. Die Kanten stellen funktionale Beziehungen zwischen Knoten zweier Klasen dar, wobei sie mit der Position des entsprechenden Argumentes innerhalb der Applikation beschriftet sind (eingekreiste Zahl).

3.4 Beziehung zwischen Berechnungsplan und Unifikationsgraph

Jeder geschlossene Knoten $v = \langle n, l_{up}, l_{down}, Alt, BP\rangle$ eines Berechnungsplanes P entspricht einer Unifikation der (unifizierbaren) Menge $\{l_{up}\sigma, l_{down}\sigma\}$, wobei σ die Folge der bisherigen Unifikationen (geschlossene Knoten Nummer 1 bis $(n-1)$) darstellt. Die Berechnung des allgemeinsten Unifikators und das Notieren desselben erfolgt mit Hilfe des Unifikationsgraphen UG_i zum Plan P_i.

Die Auswertung einer Anfrage an ein Prolog-Programm läßt sich als eine Folge von Paaren aus Berechnungsplan und zugehörigem Unifikationsgraphen darstellen:

$$\langle P_0, UG_0\rangle, \langle P_1, UG_1\rangle, \langle P_2, UG_2\rangle, \ldots$$

Der Folgeplan P_{i+1} eines Planes P_i entsteht entweder durch einen Ableitungsschritt am aktuellen Berechnungsknoten oder durch einen Backtracking-Schritt. Die Berechnung wird am Berechnungsplan mit Hilfe des zugehörigen Unifikationsgraphen definiert.

Definition 3.4 (Ableitungsschritt) Gegeben sei ein Berechnungsplan P_i mit mindestens einem offenen Knoten (d.h. er ist nicht komplett). Sei der n-te Knoten $v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ bzgl. der Prolog-Auswertungsstrategie (depth-first left-to-right) der erste offene Knoten und $Alt \neq \emptyset$. Mit $m = \min Alt$[4] habe eine Variante[5] der Programmklausel mit der Nummer m die Form l_{head} :- $l_1, \ldots, l_r$. ($r \geq 1$; Regel) bzw. l_{head}. (Fakt).
Der erweiterte Unifikationsalgorithmus versucht mit Hilfe der Funktion *extend* (s. Abb. 4) unter Erweiterung des zu P_i gehörenden Unifikationsgraphen UG_i (vgl. Def. 3.5) die Unifikation der Menge $\{l_{up}\sigma, l_{head}\sigma\}$, $\sigma = subst(UG_i)$, zu berechnen.
Dabei ist zu unterscheiden:

- Die Unifikation ist möglich ($extend(l_{up}, l_{down}, n, UG_i) = (0, UG_{i+1}, \emptyset)$) und der berechnete Unifikator ist in $subst(UG_{i+1})$ enthalten.
  Der Folgeplan P_{i+1} ist identisch zu P_i bis auf den Teilbaum mit dem n-ten Knoten v als Wurzel. Dieser Teilbaum besteht bei P_i nur aus dem offenen Knoten v. In P_{i+1} wird dieser Teilbaum aus der (geschlossenen) Wurzel $v' = \langle n, l_{up}, l_{head}, Alt \setminus \{m\}, BP \rangle$ und den (offenen) Blättern $v_j = \langle \perp, l_j, \perp, alt(l_j), \{n\} \rangle$, $1 \leq j \leq r$ gebildet. Für $r = 0$ (Fakt) besteht der Teilbaum nur aus v'.
- Die Unifikation ist nicht möglich ($extend(l_{up}, l_{down}, n, UG_i) = (1, UG', Clash\text{-}Set)$); man spricht von einem *Clash*. *Clash-Set* ist die Menge derjenigen Berechnungspunkte, die den Clash verursacht haben könnten (die sog. Clash-Menge, vgl. Abb. 4).
  Der Folgeplan wird in diesem Falle nicht durch einen Ableitungsschritt sondern durch einen Backtracking-Schritt bestimmt (s. Def. 3.6), wobei zuvor die versuchte Alternative entfernt wird ($v' = \langle \perp, l_{up}, \perp, Alt \setminus \{m\}, BP \rangle$).

□

Bemerkung Die Funktion $alt : Lit_\Sigma(Var) \longrightarrow (2^{\mathbb{N}} \setminus \{\emptyset\})$ bildet ein Literal auf die Menge der „passenden“ Programmklauseln ab (s. Definition 3.1).
Die Menge der Backtracking-Punkte eines Knotens wird mit dem Vater initialisiert, denn eine alternative Klauselauswahl am Vaterknoten (d.h. Kopf einer Regel) könnte die eventuellen Berechnungsprobleme im darunter liegenden Teilbaum (d.h. Regelrumpf) beseitigen. □

Vor der Erläuterung des Backtrackings und einer anschließenden ausführlichen Darstellung einer Berechnung mit Ableitungs- und Backtracking-Schritten ist noch die Definition des erweiterten Unifikationsalgorithmus anzugeben.

Definition 3.5 (Erweiterung eines Unifikationsgraphen) Gegeben sei ein Unifikationsgraph $UG = \langle C, F \rangle$, und der n-te Knoten $v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ eines Berechnungsplanes P soll mit einer Programmklausel geschlossen werden, deren Kopf l_{down} ist. Dann wird der Unifikationsgraph UG „unter dem Schließen des Knotens v“ erweitert durch die nachfolgend (Abb. 4) beschriebene Funktion

$$extend : Lit_\Sigma(Var) \times Lit_\Sigma(Var) \times \mathbb{N} \times \mathcal{UG} \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}.$$

Dabei wird $(l_{up}, l_{down}, n, UG)$ auf ein Tripel abgebildet, dessen erster Eintrag ein Indikator für einen Clash ist ($0 \equiv$ „erfolgreiche Unifikation“: zweite Komponente enthält erweiterten Unifikationsgraphen; $1 \equiv$ Clash: dritte Komponente enthält Clash-Menge). □

[4] Minimum wegen der Prolog-Auswertungsstrategie: Programmklauseln der Reihe nach „von oben nach unten“ ausprobieren.

[5] Variantenbildung einer Programmklausel wird durch Umbenennung der in ihr vorkommenden Variablen in neue, noch nicht vorgekommene Variablen erreicht

$extend \;:\; Lit_\Sigma(Var) \times Lit_\Sigma(Var) \times \mathbb{N} \times \mathcal{UG} \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}$

$extend(l_{up}, l_{down}, n, UG) :=$
  let $l_{up} = p(t_1, \ldots, t_r)$ and $l_{down} = p(s_1, \ldots, s_r)$ and $UG = \langle C, F\rangle$ and $T = \bigcup_{i=1}^{r}(tt(t_i) \cup tt(s_i))$
  and $NewTerms = \{t \in T \mid t \notin Var \vee (t \in Var \wedge \nexists \langle V, E, Lab\rangle \in C \text{ mit } t \in V)\}$
  and $NewClasses = \{\langle\{t\}, \emptyset, \emptyset\rangle \mid t \in NewTerms\}$ and $C' = C \cup NewClasses$
  and $NewFunDep = \{f \mid t \in NewTerms, t = g(t_1, \ldots, t_i, \ldots, t_n),$
    $c_t = \langle V_t, E_t, Lab_t\rangle \in C' \text{ mit } t \in V_t,$
    $c_{t_i} = \langle V_{t_i}, E_{t_i}, Lab_{t_i}\rangle \in C' \text{ mit } t_i \in V_{t_i},$
    $f : c_t \rightarrow c_{t_i} \text{ mit } mark_{UG}(f) = (i, t, t_i)\}$
  and $F' = F \cup NewFunDep$ and $UG' = \langle C', F'\rangle$ and $res_0 = (0, UG', \emptyset)$
  and $\forall j \in \{1, ..., r\} : res_j = unify\_term(t_j, s_j, \emptyset, n, res_{j-1})$
  in res_r

$unify\_term \;:\; T_\Sigma(Var) \times T_\Sigma(Var) \times 2^{\mathbb{N}} \times \mathbb{N} \times (\mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}) \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}$

$unify\_term(t, s, Lab, n, (ind, UG, C\text{-}Set) :=$ % Unifikation von t und s am Knoten n
  if $ind = 1$ then $(ind, UG, C\text{-}Set)$ % Clash
  else let $UG = \langle C, F\rangle$ and $c_t = \langle V_t, E_t, Lab_t\rangle \in C \;\ni\; t \in V_t$ and $rep_t = deref_{c_t}(t)$
    and $c_s = \langle V_s, E_s, Lab_s\rangle \in C \;\ni\; s \in V_s$ and $rep_s = deref_{c_s}(s)$
    in if $rep_t = rep_s$ then $(0, UG, \emptyset)$
      else if $rep_s \in Var$ then $bind(rep_s, rep_t, Lab, n, UG)$
      else if $rep_t \in Var$ then $bind(rep_t, rep_s, Lab, n, UG)$
      else let $rep_t = c_t(t_1, \ldots, t_{r_t})$ and $rep_s = c_s(s_1, \ldots, s_{r_s})$
        in if $(c_t \neq c_s)$ or $(r_t \neq r_s)$
          then $(1, UG, (Lab \cup Lab_t \cup Lab_s) \setminus \{n\})$ % Clash-Menge
          else let $res_0 = (0, UG, \emptyset)$ and $\forall j \in \{1, ..., r_t\}$:
            $res_j = unify\_term(t_j, s_j, Lab \cup Lab_t \cup Lab_s, n, res_{j-1})$
          in res_{r_t}

$bind \;:\; Var \times T_\Sigma(Var) \times 2^{\mathbb{N}} \times \mathbb{N} \times \mathcal{UG} \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}$

$bind(rep_1, rep_2, Lab, n, UG) :=$ % $rep_1 \in Var$ wird an rep_2 gebunden
  let $UG = \langle C, F\rangle$ % aus den Klassen c_1 und c_2 wird die Klasse c erstellt
  and $c_1 = \langle V_1, E_1, Lab_1\rangle \in C \ni rep_1 = deref_{c_1}(v_1)\; \forall v_1 \in V_1$
  and $c_2 = \langle V_2, E_2, Lab_2\rangle \in C \ni rep_2 = deref_{c_2}(v_2)\; \forall v_2 \in V_2$
  and $c = \langle V_1 \cup V_2, E_1 \cup E_2 \cup \{(rep_1 \rightarrow rep_2)\}, Lab \cup Lab_1 \cup Lab_2 \cup \{n\}\rangle$
  and $C' = (C \cup \{c\}) \setminus \{c_1, c_2\}$
  and $F' = \{f \in F \mid f = (c_s \rightarrow c_d),\;\; c_s, c_d \notin \{c_1, c_2\},$
    $mark_{UG}(f) = mark_{UG'}(f)\}$
    $\cup\, \{f' = (c \rightarrow c_d) \mid \exists f = (c_s \rightarrow c_d) \in F,$
    $c_s \in \{c_1, c_2\}, mark_{UG}(f) = mark_{UG'}(f')\}$
    $\cup\, \{f' = (c_s \rightarrow c) \mid \exists f = (c_s \rightarrow c_d) \in F,$
    $c_d \in \{c_1, c_2\}, mark_{UG}(f) = mark_{UG'}(f')\}$
  and $UG' = \langle C', F'\rangle$
  in if UG' zirkulär then $(1, UG, cycle\_set(UG'))$ % occur check
    else $(0, UG', \emptyset)$

wobei
- $tt : T_\Sigma(Var) \longrightarrow 2^{T_\Sigma(Var)}$ einen Term auf die Menge seiner Teilterme abbildet,
- $NewTerms$ die Menge der neu eingeführten Terme ohne bereits früher eingeführte Variablen und $NewFunDep$ die Menge der Kanten entsprechend der neuen funktionalen Abhängigkeiten ist, und
- $cycle\_set : \mathcal{UG} \longrightarrow 2^{\mathbb{N}}$ einen zirkulären Unifikationsgraphen auf die Vereinigung der Label-Mengen der Klassen eines Zyklus abbildet.

Abbildung 4: Die Funktion *extend*

Ein Backtracking-Schritt an einem offenen Knoten v eines Planes P gliedert sich in zwei Schritte. Zunächst wird die Clash-Menge mit Hilfe eines Literals l_{down} berechnet (dies geschieht schon beim Versuch der Unifikation (Def. 3.4) mit der Funktion *extend* (Abb. 4)) und der fünften Komponente von v (BP) hinzugefügt. Im zweiten Schritt wird unterschieden, ob die Menge Alt leer ist oder nicht.
Ist sie nicht leer, so ist die rückwärtsgerichtete Berechnung abgeschlossen („shallow backtracking"). Ansonsten („deep backtracking") wird unter Zugriff auf die Menge BP die Berechnung weiter zurückgesetzt.

Definition 3.6 (Backtracking) Gegeben sei ein nicht kompletter Plan P_i mit dem ersten (bzgl. depth-first left-to-right) offenen Knoten $v = \langle \perp, l_{up}, \perp, Alt_v, BP_v \rangle$. Weiterhin sei eine Clash-Menge $Clash$ gegeben. Die Berechnung des Planes P_{i+1} geschieht dann in zwei Schritten:

- Zuerst wird die Menge der Backtracking-Punkte BP_v um die Menge $Clash$ erweitert ($BP_v := BP_v \cup Clash$). Die Notwendigkeit dieses Aufsammelns der Backtracking-Punkte wird im Abschnitt 3.6 erläutert.
- Danach erfolgt das eigenliche Backtracking. Es wird nach der Menge Alt_v unterschieden:
  - Ist die Menge nicht leer ($Alt_v \neq \emptyset$), so ist der Backtracking-Schritt beendet („shallow backtracking"). Als nächstes fährt die Berechnung mit einem Ableitungsschritt am Knoten v fort.
  - Falls keine alternativen Klauseln mehr bereitstehen ($Alt_v = \emptyset$), so wird die Berechnung zurückgesetzt („deep backtracking"). Dazu wiederum zwei Fälle:
    * $BP_v = \emptyset$: Dann bricht die gesamte Berechnung ab, da keine weitere Lösungen existieren. Entweder besitzt der Plan keinen weiteren Berechnungspunkt, der ein Scheitern eines bestimmten Unifikationsversuches verhindern könnte, oder nach Berechnung einer Lösung gibt es keine Auswahlstelle mit Alternativen mehr (siehe Abschnitt 3.5).
    * $BP_v \neq \emptyset$: Alle Elemente von BP_v sind Nummern von geschlossenen Knoten in P_i. Sei m das Maximum von BP_v (d.h. der Knoten m ist der jüngste Berechnungspunkt aus BP_v). Dann wird die Berechnung sukzessive bis zum Öffnen des Knotens mit der Nummer m (v_m) zurückgesetzt.
      Und zwar werden diejenigen geschlossenen Knoten, die beim Schließen des Knotens v_m als offene Knoten existierten (Knoten mit Nummer aus $limit(m)$), wieder geöffnet. Formal: Bezeichne $father(n)$ die Nummer des Knotens in P_i, der Vater des Knotens mit der Nummer n ist (existiert wegen Normalisierung der Programme). So ist $limit(m) := \{n \in \mathbb{N} \mid n$ Nummer eines geschlossenen Knotens aus P_i, $n \geq m$ und $father(n) < m\}$. P_{i+1} entsteht dann aus P_i, indem alle Teilbäume, deren Wurzel eine Nummer aus $limit(m)$ besitzt, ohne die Wurzel gelöscht werden. Weiterhin werden diese Wurzeln w außer v_m (Nummer aus $limit(m) \setminus \{m\}$) neu „initialisiert" (d.h. $w = \langle \perp, l_w, \perp, alt(l_w), \{father(w)\} \rangle$) und der Knoten v_m wird geöffnet ($v_m = \langle \perp, l_m, \perp, Alt_m, BP'_m \rangle$ mit $BP'_m := BP_m \cup (BP \setminus \{m\})$ (Aufsammeln von Backtracking-Punkten; vgl. Abschnitt 3.6).
      Die Berechnung von P_{i+1} wird dann sukzessive beim „eigentlichen Backtracking" (zweiter Punkt •) weitergeführt, wobei nun v_m aktueller Berechnungspunkt ist.

Der zu P_{i+1} gehörige Unifikationsgraph UG_{i+1} entsteht aus UG_i, indem beim Eliminieren und Öffnen von Berechnungsknoten die an diesen Knoten gemachten Bindungen (Funktion *extend* bzw. *bind*) rückgängig gemacht werden.

□

Bemerkung Man kann zu einem Plan P den zugehörigen Unifikationsgraphen UG auch auf folgendem Weg erhalten: Ausgehend vom initialen (d.h. leeren) Graphen UG_0 werden sukzessive alle Erweiterungen bzgl. geschlossener Knoten von P vorgenommen (wobei es nicht zum Clash kommen kann). Diese Neuberechnung bedeutet aber natürlich einen hohen Berechnungsaufwand, den man bei der Ausführung des Programms vermeiden muß.

Beim Naiven Backtracking ist die Menge BP_v eines Knotens, der nicht die Wurzel eines Planes ist, immer einelementig und enthält den Vorgängerknoten (bzgl. depth-first left-to-right). Bei der Implementation vermeidet man Rücksprünge zu Auswahlstellen ohne Alternativen durch vorzeitiges Entfernen des Auswahlblockes („choice point“) vom Funktionskeller (vgl. [Aït-Kaci 91]). □

Vor der Angabe des IB-Schemas soll an dieser Stelle mit Abbildung 5 die Berechnung einer Anfrage an eine Prolog-Programm illustriert werden. Als Beispiel dient das erste in Abschnitt 2.1 eingeführte Programm (Seite 4), welches der Übersicht wegen noch einmal aufgeführt wird:

Beispiel

1) q(f(X),Z,Z) :- r(X).	2) r(a).	3) p(Y,a).
1') q(a,b,c).	2') r(b).	3') p(Y,b).
4) n(f(b)).		3") p(V,V) :- r(V).
5) goal(X',Y',W') :- q(X',X',Y'), p(g(Y'),W'), n(Y').		

?- goal(X",Y",W").

Nach vier erfolgreichen Ableitungsschritten kommt es im Zustand $\langle P_4, UG_4\rangle$ zu einem Clash. Die Menge BP des letzten Knotens wird um die Clash-Menge {0,1,2} erweitert ($v = \langle \perp, \texttt{n(Y')}, \perp, \emptyset, \{0,1,2\}\rangle$). Wegen des Fehlens weiterer Alternativen wird die Berechnung bis zum Öffnen des Knotens mit der Nummer 2 zurückgesetzt. Dazu werden die letzten beiden Knoten neu initialisiert und der Knoten mit der Nummer 2 geöffnet. Der Backtracking-Schritt endet wegen der bestehenden Alternative `2')`. Die folgenden drei Ableitungsschritte komplettieren den Berechnungsplan.

3.5 Das IB-Schema

Nun sind alle Vorbereitungen getroffen, um ein Schema des Intelligenten Backtrackings bei der Ausführung von Prolog-Programmen anzugeben. Wesentlich ist noch das Verständnis für die Tatsache, daß nach der Berechnung eines kompletten Planes P (d.h. einer Lösung einer Anfrage an ein Logik-Programm) für einen nachfolgenden Plan P_f Intelligentes Backtracking in gemeinsamen Teilbereichen von P und P_f nicht erlaubt ist. Damit ist gemeint, daß der potentielle Suchraum ausführlich getestet werden muß (Naives Backtracking), um die Vollständigkeit des Verfahrens zu erhalten. Denn Backtracking-Punkte in den Mengen BP der Knoten eines Planes werden für das Scheitern von Unifikationen berechnet, aber nicht für explizites Backtracking bei einer gefundenen Lösung. Die Wahl einer alternativen Klausel an einem Berechnungspunkt, den man beim Scheitern einer Unifikation überspringen würde (z.B. weil er „autark“ ist und keinen Unifikationsfehler verschulden kann), könnte eine weitere Lösung zur Folge haben und darf bei einer erfolgreichen Berechnung nicht übersprungen werden. In nicht gemeinsamen Teilen von P und P_f (d.h. neu berechneten Teilen ausgehend von P) ist Intelligentes Backtracking wieder erlaubt. Realisiert wird dieser Wechsel zwischen Intelligentem und Naivem Backtracking dadurch, daß an allen Knoten eines kompletten Planes die Mengen BP mit dem jüngsten (depth-first left-to-right) Vorgänger neu initialisiert werden, dessen *Alt*-Menge nicht leer ist (Simulation des Naiven Backtrackings: Rücksprung zur letzten Auswahlstelle mit Alternativen). Existiert ein solcher Vorgänger nicht, so wird BP mit der leeren Menge initialisiert. Solch ein Wechsel zwischen Intelligentem und Naivem Backtracking ist i.a. nur in der geschilderten extremen Form möglich, damit die Vollständigkeit des Verfahrens bestehen bleibt.

Abbildung 6 zeigt das Schema des Intelligenten Backtrackings bei der Auswertung von Prolog-Programmen. Die Funktion *next_state* stellt in gewiser Weise eine Übergangsfunktion der einzelnen Berechnungszustände dar. Das Schema besteht dann aus einer Folge von Zuständen, die von einem Anfangszustand aus durch Iteration der Übergangsfunktion erreicht werden.

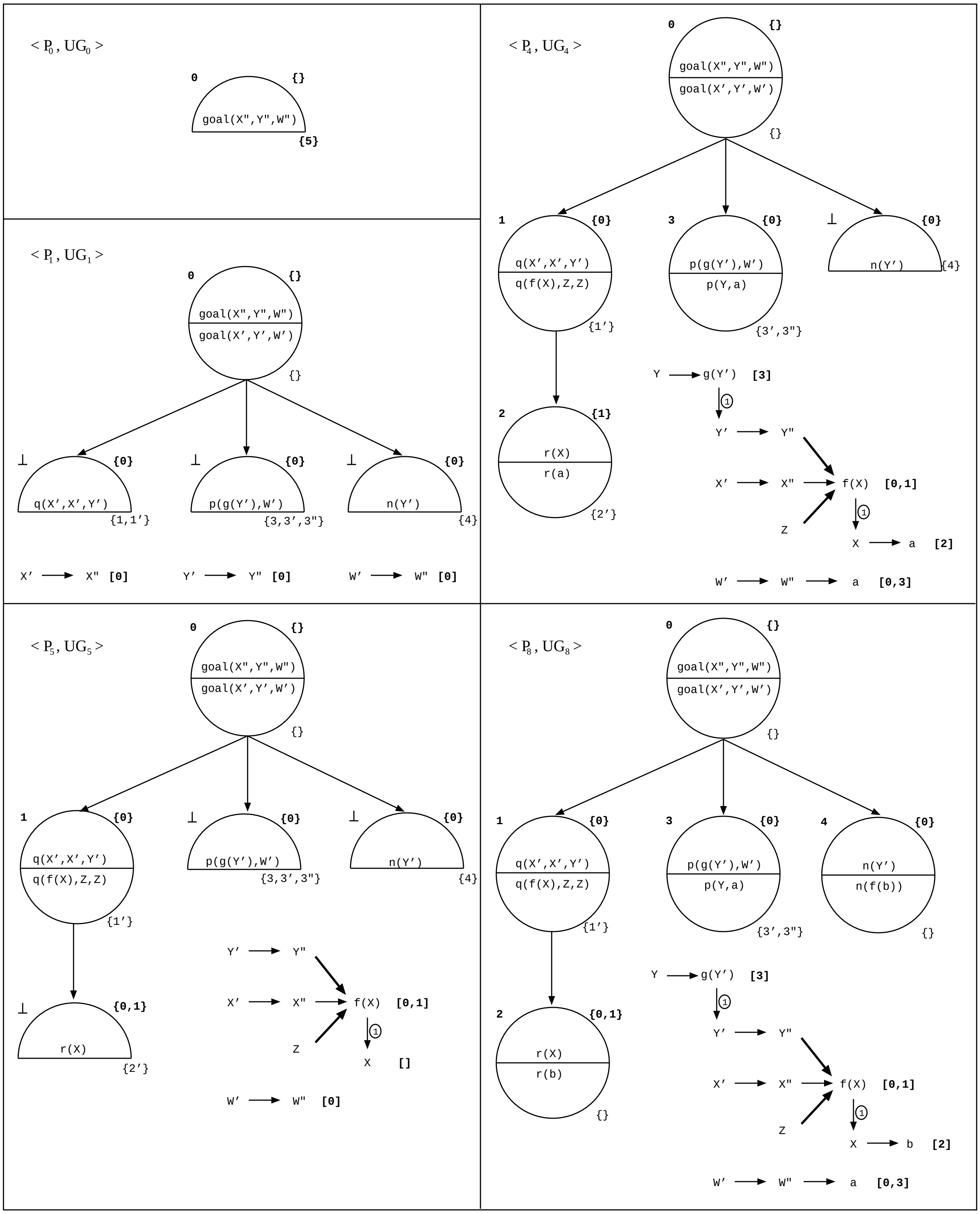


Abbildung 5: Eine Berechnung mit Berechnungsplan und Unifikationsgraph

ib_scheme :=
  let P_0 „initialer Berechnungsplan“
  and UG_0 „initialer Unifikationsgraph“
  and $state_0 = \langle P_0, UG_0 \rangle$
  and $\forall j \in \mathbb{N} : state_{j+1} =$ if $next\_state(state_j) = (\langle P', UG' \rangle, yes)$
    then $\langle P', UG' \rangle$
  in $\langle P_0, UG_0 \rangle, \langle P_1, UG_1 \rangle, \ldots$

$next\_state : \langle \mathcal{PL} \times \mathcal{UG} \rangle \longrightarrow \langle \mathcal{PL} \times \mathcal{UG} \rangle \times \{yes, no\}$

$next\_state(\langle P, UG \rangle)$:=
  if („P nicht komplett“)
  then let $v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ erster (bzgl. depth-first left-to-right) offener Knoten
    and „v soll als n-ter Knoten mit l_{head} (Regel m) geschlossen werden“
    and $extend(l_{up}, l_{head}, n, UG) = (ind, UG', Clash\text{-}Set)$
    in if $ind = 0$ % kein Clash
      then let „P' entsteht aus P durch Schließen von v und
          Einführung eventueller Söhne“
        in $(\langle P', UG' \rangle, yes)$
      else let „P' entsteht aus P durch Änderung von v zu
          $v := \langle \perp, l_{up}, \perp, Alt \setminus \{m\}, BP \cup Clash\text{-}Set \rangle$“
        in $deep\_back(\langle P', UG \rangle)$
  else % P komplett, evtl. Lösung mit Hilfe von UG ausgeben
    let $v = \langle n, l_{up}, l_{down}, Alt, BP \rangle$ geschlossener Knoten mit maximaler Nummer
    and „P', UG' entsteht durch Öffnung von v in P zu $v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$“
    and % Simulation von Naivem Backtracking
      „P'' entsteht aus P', indem die Menge BP eines jeden Knotens mit der
      maximalen Nummer aller jüngeren Knoten (depth-first left-to-rigth) mit
      nichtleeren *Alt*-Mengen initialisiert wird (sonst leere Menge)“
    in $deep\_back(\langle P'', UG' \rangle)$

$deep\_back : \langle \mathcal{PL} \times \mathcal{UG} \rangle \longrightarrow \langle \mathcal{PL} \times \mathcal{UG} \rangle \times \{yes, no\}$

$deep\_back(\langle P, UG \rangle)$:=
  let „$v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ erster (bzgl. depth-first left-to-right) offener Knoten“
  in if $Alt = \emptyset$
    then if $BP = \emptyset$
      then $(\langle P, UG \rangle, no)$ % keine weitere Lösung vorhanden
      else let „$\langle P', UG' \rangle$ entsteht durch Zurücksetzen der Berechnung
          bis zum Öffnen des Knotens v_m mit Nummer $m = \max(BP)$
          und Erweiterung von BP_m des Knotens v_m um $(BP \setminus \{m\})$“
        in $deep\_back(\langle P', UG' \rangle)$
    else $(\langle P, UG \rangle, yes)$ % Backtracking beendet

Abbildung 6: Das IB-Schema

Das vorgestellte Verfahren, welches beim Backtracking evtl. Auswahlstellen mit Alternativen überspringt, ist vollständig bezüglich der üblichen Prolog-Auswertungsstrategie. Alle Ergebnisse, die mit Naivem Backtracking gefunden werden, werden auch mit Intelligentem Backtracking gefunden. Natürlich bleibt weiterhin die Auswertungsstrategie „Tiefensuche“ (depth-first left-to-right) unvollständig. Gemeint ist hier folgendes:

Zu einem Prolog-Programm S und einer Anfrage G seien S_N bzw. S_I die Folgen von Ergebnissen, die mit dem Naiven bzw. Intelligenten Backtracking gefunden werden. Dann gilt:

- Falls S_N eine endliche Folge ist, $S_N = \langle\sigma_1, \ldots \sigma_k\rangle$, $k \in \mathbb{N}$, so ist S_I eine Folge $S_I = \langle\sigma_1, \ldots, \sigma_k, \ldots\rangle$, d.h. eine (möglicherweise unendliche) Folge mit dem Präfix S_N.
- Ist S_N eine unendliche Folge, so ist S_I ebenfalls unendlich und jeweils die Präfixe der Länge $k \in \mathbb{N}$ der beiden Folgen sind identisch.

Der erste Punkt beschreibt die Möglichkeit, daß Intelligentes Backtracking Teile des Suchraumes abschneidet, die bei Naivem Backtracking eine nichtterminierende Berechnung verursachen.
In [Codognet 89a] wird diese Vollständigkeit für ein Verfahren bewiesen, welches genauer ist, d.h. dessen BP-Mengen der Berechnungsknoten immer Teilmengen der entsprechenden Mengen des vorgestellten Verfahrens sind. Das bedeutet, daß das hier beschriebene Verfahren immer zumindest den Suchraum bearbeitet, den auch das als vollständig bewiesene Verfahren absucht. Daraus folgt die Vollständigkeit des hier beschriebenen Verfahrens im obigen Sinne.

3.6 Die fünfte Komponente BP

In diesem Abschnitt wird gezeigt, daß eine Vernachlässigung von früher berechneten Backtracking-Punkten zur Unvollständigkeit im obigen Sinne führt. Wie das IB-Schema und die Definition des Backtrackings festlegen, werden Backtracking-Punkte, die an einem Knoten aufgesammelt wurden (BP-Menge), beim Backtracking von diesem Knoten aus zum neuen aktuellen Knoten mitgenommen und dessen BP-Menge hinzugefügt. Dieses Aufsammeln der berechneten Punkte ist wesentlich für die Vollständigkeit des Verfahrens. Selbst wenn ein Knoten keine alternativen Regeln mehr besitzt ($Alt = \emptyset$) ist die entsprechende Menge BP relevant, so daß beim Intelligenten Backtracking auch diese Knoten (kurz) besucht werden. Das folgende Beispiel erläutert den Sachverhalt.

Beispiel

1) m(c).	2) p(a).	3) q(b,c).
1') m(b).	2') p(b).	3') q(a,b).

4) r(X,X).
5) goal(X',Y',Z') :- m(Z'), p(X'), q(X',Y'), r(Z',Y').

?- goal(X",Y",Z").

Die Berechnung der Anfrage ist auszugsweise in der Abbildung 7 angegeben: Nach drei (vorwärtsgerichteten) Berechnungsschritten haben der Berechnungsplan und zugehörige Unifikationsgraph die dargestellte Struktur. Es folgt der Versuch, den nächsten offenen Knoten unter Anwendung der Regel `3)` zu schließen. Dieses führt zu einem Clash mit der Clash-Menge {0,2}, um die die Menge BP des Knotens erweitert wird. Der darauf folgende Ableitungsschritt ist erfolgreich, so daß sich die Berechnung dann in dem mit $\langle P_5, UG_5\rangle$ angegebenen Zustand befindet.
Der Hauptaspekt des Beispieles liegt im Scheitern des folgenden Ableitungsschrittes mit einer Clash-Menge {0,1,3}: Würde das IB-Verfahren nun den Knoten mit der Nummer 3 wegen der dort nicht vorhandenen alternativen Regeln nicht besuchen und die Berechnung am Knoten mit der Nummer 1 fortführen, so ginge eine Lösung verloren (Regelanwendungen `1,2',3,4`). Anschaulich gesprochen zeigt die Berechnung, daß u.a. am „Scheitern des Knotens“ mit der Nummer 4 der Knoten mit der Nummer 3, und am „Scheiterns des Knotens“ mit der Nummer 3 der Knoten mit der Nummer 2 beteiligt ist. Diese (transitiven) Abhängigkeiten müssen durch das Intellignte Backtracking (so weit wie benötigt) erhalten bleiben. Andererseits enthält die BP-Menge eines durch Intelligentes Backtracking übersprungenen Knotens keine relevanten Informationen.

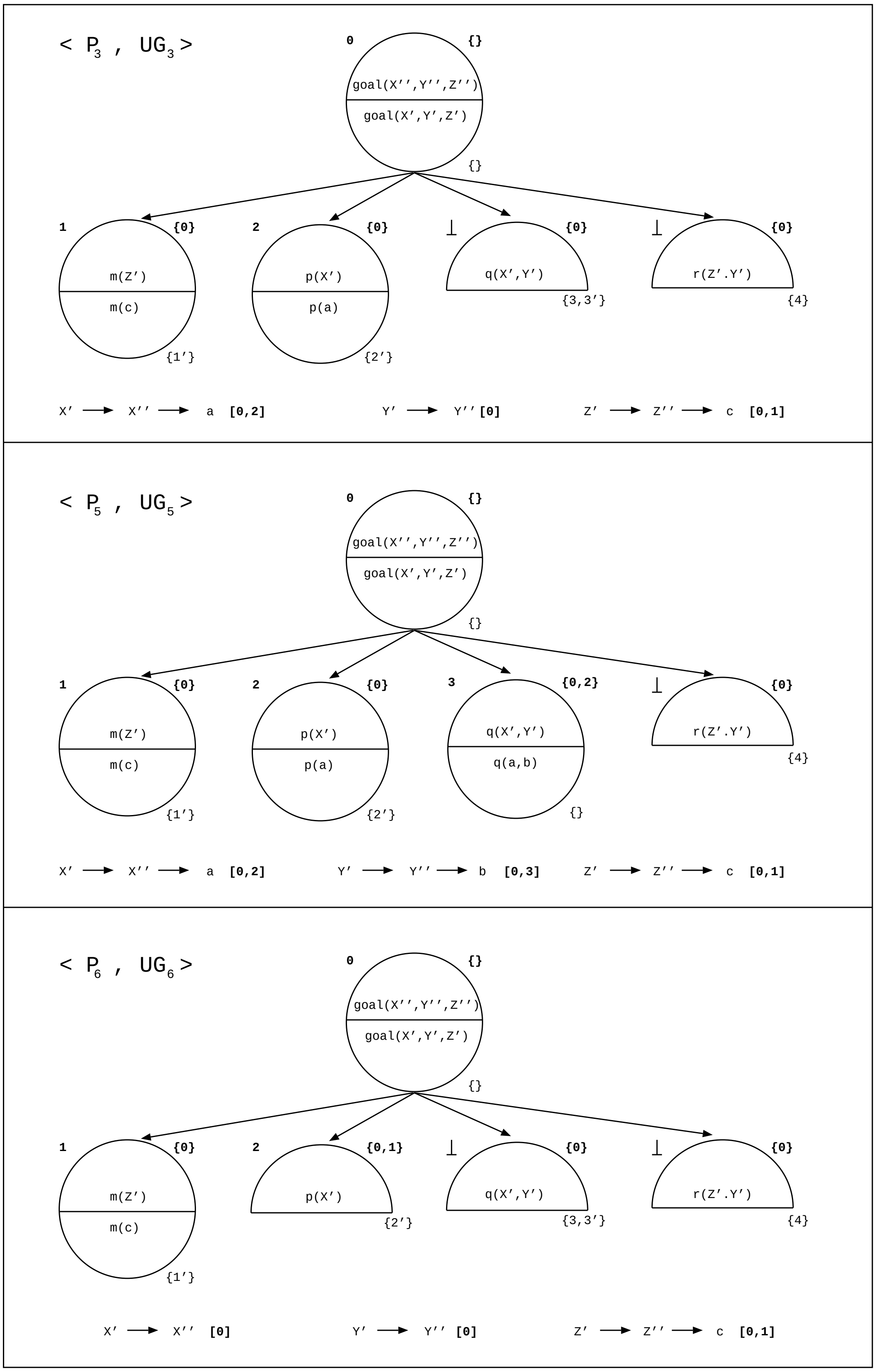


Abbildung 7: Bedeutung der Mengen *BP*

Man kann nun deutlich das unterschiedliche Vorgehen zwischen Naivem und Intelligentem Backtracking erkennen. Beim Naiven Backtracking werden nacheinander alle Alternativen jedes Knotens ausprobiert, wobei für die Auswahl der zu besuchenden Knoten nur deren Mengen *Alt* konsultiert werden. Dagegen interessiert sich das IB-Verfahren (zunächst) nicht für die Tatsache, ob an einem Knoten weitere Alternativen bestehen oder nicht. Einzig und allein eine Menge von Knotennummern, welche die an einem vorherigen Scheitern der Berechnung beteiligten Knoten bezeichnen, steuert das Backtracking.

Die berechneten Mengen von Knotennummern (und damit das Verfahren) sind nur für reine Logik-Programme (reine Hornklausel-Programme) vollständig. Weitergehende Konzepte der Sprache Prolog (metalogische und nichtlogische Prädikate) zerstören die Vollständigkeit des Verfahrens. Dies sind insbesondere der Cut (!) zur expliziten Steuerung der Auswertung, die Arithmetik (Wertzuweisung *is*, Vergleichsoperatoren =:=, <, >=, ...), Typprädikate (*atom, var, number,* ...), die Negation (*not,* \+, ...), Programmanipulatoren (*clause, assert, retract,* ...) und die Ein-/Ausgabe (*tell, write, told,* ...). Der Gebrauch dieser Konzepte erfordert für das Intelligente Backtracking weitergehende Berechnungen (vgl. [Koch 92]).

4 Die funktional-logische Sprache Babel

Dieser Abschnitt gibt einen Überblick über die funktional-logische Sprache Babel. Neben der Syntax der konstruktorbasierten und ungetypten Sprache, die auch das Konzept der Funktionen höherer Ordnung kennt, wird die Semantik eines Babel-Programms angegeben.

4.1 Syntax

Jedem Babel-Programm liegt eine Signatur $\Sigma = (DC_\Sigma, FS_\Sigma)$ mit

- $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC^n$ als Menge der Konstruktorsymbole mit $true, false \in DC^0$,
- $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS^n$ als Menge der Funktionssymbole

und eine Menge Var von Variablen zugrunde. Syntaktische Einheiten der Sprache Babel sind

- Variablen $X, Y, \ldots \in Var$
- Terme $s, t, \ldots \in T_\Sigma(Var)$

$$\begin{array}{lll} t & ::= X & \text{\% Variable} \\ & \mid c & \text{\% Konstante, } c \in DC^0 \\ & \mid c(t_1, \ldots, t_n) & \text{\% Konstruktorapplikation, } c \in DC^n \\ & & \quad t_i \in T_\Sigma(Var), 1 \le i \le n \end{array}$$

- Ausdrücke $M, N, B \ldots \in Exp_\Sigma(Var)$

$$\begin{array}{lll} M & ::= t & \text{\% Term, } t \in T_\Sigma(Var) \\ & \mid c(M_1, \ldots, M_n) & \text{\% Konstruktorapplikation, } c \in DC^n \\ & \mid f\ M_1 \ldots M_n & \text{\% Funktionsapplikation, } f \in FS^m, n \le m \\ & \mid (M\ M_1 \ldots M_n) & \text{\% Applikation} \\ & \mid \text{if } B \text{ then } M & \text{\% Bedingter Ausdruck ohne Alternative} \\ & \mid \text{if } B \text{ then } M_1 \text{ else } M_2 & \text{\% Bedingter Ausdruck mit Alternative} \end{array}$$

Eine Babel-Regel für ein Funktionssymbol $f \in FS^n$ mit optionaler Bedingung B hat die Form

$$\underbrace{f\ t_1 \ldots t_n}_{\text{Lhs}} := \underbrace{\underbrace{\{\text{if } B \text{ then}\}}_{\text{Guard}}\ \underbrace{M}_{\text{Body}}}_{\text{Rhs}} .$$

und besteht aus einer linken Seite *Lhs* und einer rechten Seite *Rhs*, die wiederum in optionalem *Guard* und *Body* unterschieden wird (vgl. [Moreno-Navarro, Rodríguez-Artalejo 89] und [Winkler 91]). Ein Babel-Programm ist eine endliche Folge solcher Regeln und muß folgende Restriktionen erfüllen:

Flache linke Regelseiten $t_i \in T_\Sigma(Var)$.

Linkslinearität Keine Variable tritt mehrmals auf der linken Seite einer Regel auf.

Restriktion freier Variablen Eine Variable, die nur auf der rechten Seite einer Regel erscheint, heißt frei. Freie Variablen dürfen nur im Guard einer Regel auftreten.

Eindeutigkeit Jedes Paar von Regeln für das gleiche Funktionssymbol f

$$\begin{array}{rcl} f\ t_1 \ldots t_n & := & \{\text{if } B \text{ then}\}\ M \\ f\ s_1 \ldots s_n & := & \{\text{if } C \text{ then}\}\ N \end{array}$$

erfüllt (mindestens) eine der drei folgenden Bedingungen:

Keine Überlagerung $f\ t_1 \ldots t_n$ und $f\ s_1 \ldots s_n$ sind nicht unifizierbar.

Fusion der Regelrümpfe Besitzen $f\ t_1 \ldots t_n$ und $f\ s_1 \ldots s_n$ einen allgemeinsten Unifikator σ, so sind $M\sigma$ und $N\sigma$ identisch.

Inkompatibilität der Guards Besitzen $f\ t_1 \ldots t_n$ und $f\ s_1 \ldots s_n$ einen allgemeinsten Unifikator σ und enthalten beide Regeln einen Guard, so ist $B\sigma \wedge C\sigma$ inkompatibel (Inkompatibilität als syntaktisch entscheidbare Unerfüllbarkeit; vgl. [Moreno-Navarro, Rodríguez-Artalejo 89]).

Regeln für die booleschen Operationen Negation, Konjunktion und Disjunktion sind ebenso vordefiniert wie Regeln für die Auswertung bedingter Ausdrücke und für die Gleichheit „=".

Reine Prolog-Programme (reine Hornklausel-Programme) können in einfacher Weise in Babel-Programme überführt werden: Im wesentlichen wird jede Prolog-Klausel als Regel einer booleschen Funktion notiert, wobei der Klauselrumpf den Guard der Babel-Regel (Konjunktion von Applikationen boolescher Funktionen) darstellt. Dabei ist die Linkslinearität von Babel-Regeln zu beachten, welche eventuell die Umbenennung von Variablen und die Hinzunahme von Gleichheitsbedingungen im Guard erfordert. Beispielsweise lautet ein zum Prolog-Programm auf Seite 5 entsprechendes Babel-Programm:

```
1)  app [] L' L" :- if (L'=L") then true.
1') app [E|R] L [E'|RL] :- if ( (E=E'), (app R L RL) ) then true.
2)  goal V W X Y Z :- if ( (app [a|V] W Z), (app Z X [b|Y]) ) then true.

?-  goal V' W' X' Y' Z'.
```

4.2 Semantik

Die operationelle Semantik von Babel wird hier als Innermost-Narrowingsemantik beschrieben (vgl. [Loogen 91]). Die mathematischen Grundlagen und die deklarative Semantik sind in [Moreno-Navarro, Rodríguez-Artalejo 89] gegeben. Eine verzögerte Auswertungsstrategie (*Lazy Narrowing*) verfolgt [Moreno-Navarro et al. 90].

Als Narrowing bezeichnet man einen Auswertungsmechanismus ähnlich der Reduktion in funktionalen Sprachen, bei dem das unidirektionale Pattern Matching durch die bidirektionale Unifikation ersetzt wird. Ein Σ-Ausdruck wird durch Anwendung einer Programmregel mit Narrowing reduziert, indem ein Teilausdruck mit einer linken Regelseite unifiziert und der Teilausdruck durch die rechte Regelseite, auf die die Substitution angewendet wurde, ersetzt wird.
Die Narrowing-Semantik basiert auf einzelnen Narrowing-Regeln

$$\rightarrow_\sigma \;\subseteq\; Exp_\Sigma(Var) \times Exp_\Sigma(Var), \quad \sigma : Var \longrightarrow Exp_\Sigma(Var)$$

- Die impliziten Programmregeln für bedingte Ausdrücke und Regeln mit Guard benötigen folgende Narrowingregeln:

$$\begin{array}{rcl}(\text{if } true \text{ then } M) & \rightarrow_\varepsilon & M, \\ (\text{if } true \text{ then } M_1 \text{ else } M_2) & \rightarrow_\varepsilon & M_1, \\ (\text{if } false \text{ then } M_1 \text{ else } M_2) & \rightarrow_\varepsilon & M_2\end{array}$$

  Dabei bezeichnet ε die leere Substitution ($\varepsilon(X) = X,\ \forall\ X \in Var$).

- Sei $(f\ M_1 \ldots M_n)$ eine Σ-Ausdruck und $f\ t_1 \ldots t_n := R.$ eine variablendisjunkte Variante einer expliziten Programmregel. Gibt es einen allgemeinsten Unifikator $\theta \cup \sigma : Var \longrightarrow Exp_\Sigma(Var)$ mit $t_i\theta = M_i\sigma,\ 1 \leq i \leq n$, so gilt:

$$(f\ M_1 \ldots M_n) \rightarrow_\sigma R\theta$$

Mit Hilfe der Narrowingregeln kann nun induktiv über die Struktur des zu reduzierenden Ausdruckes die Narrowingrelation

$$\Rightarrow_\sigma\ \subseteq\ Exp_\Sigma(Var) \times Exp_\Sigma(Var), \quad \sigma : Var \longrightarrow Exp_\Sigma(Var)$$

definiert werden:

- $\rightarrow_\sigma\ \subseteq\ \Rightarrow_\sigma$

- Mit $M_i \rightarrow_\sigma N_i, i \in \{1, \ldots, n\}$, $f \in FS^n$ folgt

$$(f\ M_1\ \ldots\ M_i\ \ldots\ M_n) \Rightarrow_\sigma (f\ M_1\sigma\ \ldots\ N_i\ \ldots\ M_n\sigma)$$

- Mit $M_i \rightarrow_\sigma N_i, i \in \{1, \ldots, n\}$, $c \in DC^n$ folgt

$$c(M_1\ \ldots\ M_i\ \ldots\ M_n) \Rightarrow_\sigma c(M_1\sigma\ \ldots\ N_i\ \ldots\ M_n\sigma)$$

- $B \rightarrow_\sigma C$ impliziert

$$\begin{array}{rcl}(\text{if } B \text{ then } M) & \Rightarrow_\sigma & (\text{if } C \text{ then } M\sigma) \\ (\text{if } B \text{ then } M_1 \text{ else } M_2) & \Rightarrow_\sigma & (\text{if } C \text{ then } M_1\sigma \text{ else } M_2\sigma)\end{array}$$

Die Berechnung eines Zielausdruckes $M \in Exp_\Sigma(Var)$ bzgl. eines Babel-Programms besteht aus einer Iteration einzelner Schritte gemäß der Narrowingrelation, bis auf den reduzierten Ausdruck keine Regel mehr anwendbar ist:

$$M \overset{*}{\Rightarrow}_\omega N$$

$\overset{*}{\Rightarrow}$ ist die transitive und reflexive Hülle der Narrowingrelation $\Rightarrow_\sigma$, wobei ω die Komposition der einzelnen Substitutionen ist. N wird als Resultat (-ausdruck) und ω als Antwort (-substitution) bezeichnet.
Die abstrakte Babelmaschine SBAM berechnet einen Zielausdruck bzgl. eines Babel-Programms. Ihre Arbeitsweise ist in [Koch 92, Abschnitt 2.3] ausführlich beschrieben (s. auch Abschnitt 7).

5 Intelligentes Backtracking und Babel

Bei der Auswertung eines Babel-Ausdruckes bzgl. eines Babel-Programms kommt es ähnlich wie bei Prolog zu Unifikationen (bei Regelanwendungen) und Backtracking. In diesem Abschnitt werden alle Aspekte dargestellt, die beim Intelligenten Backtracking bei der Auswertung von Babel-Programmen mit der in Abschnitt 4 beschriebenen Innermost-Strategie auftreten. Die Grundlage bildet das in Abschnitt 3 vorgestellte Verfahren des Intelligenten Backtrackings bei der Auswertung von Prolog-Programmen. In der Literatur ist Intelligentes Backtracking nur für die Auswertung logischer Sprachen bekannt, nicht aber für Sprachen, die eine Integration funktionaler und relationaler Konzepte darstellen.

Die funktionalen Konzepte von Babel erfordern eine Erweiterung der Berechnungspläne (im nächsten Abschnitt) und eine von Prolog völlig verschiedene Berechnung. Im Abschnitt 5.2 wird der wesentlichste neue Mechanismus des Intelligenten Backtrackings bzgl. Babel erläutert (Bezug zwischen Berechnung und Ergebnis).
Neben den einem Babel-Programm vorgebenen Regeln für bedingte Ausdrücke (Abschnitt 5.3) erfordert auch das Konzept der Funktionen höherer Ordnung eine spezielle Behandlung.

5.1 Erweiterung des Berechnungsplanes um funktionale Aspekte

Der wesentlichste Unterschied beim Übergang von logischen zu funktional-logischen Sprachen ist das Hinzukommen funktionaler Konzepte. Insbesondere bilden funktionale Ausdrücke syntaktische Einheiten der Sprache Babel. Das Ergebnis eines Zielausdruckes bzgl. eines Babel-Programms ist ein Paar bestehend aus einem Resultat (-ausdruck) und einer Antwort (-substitution) für die im Zielausdruck enthaltenen Variablen.

Die Berechnungspläne für Prolog-Programme werden zu Berechnungsplänen für Babel-Programme erweitert, indem den U-Knoten (Unifikationsknoten) eine zweite Art von Knoten, die A-Knoten (Ausdrucksknoten), hinzugefügt werden. Die graphisch rund dargestellten U-Knoten repräsentieren wie in Prolog-Berechnungsplänen eine Unifikation (hier zweier Ausdrücke), wobei die A-Knoten (graphisch rechteckig dargestellt) funktionale Ausdrücke mit ihren Werten andeuten. Der gesamte Aufbau eines Babel-Berechnungsplanes orientiert sich sehr stark an der Auswertungsstrategie (innermost).
Den weiteren Überlegungen dieses Abschnitts liegt die in Abschnitt 4 (Seite 20) angegebene Signatur zugrunde.

Definition 5.1 (Babel-Berechnungsplan) Ein Berechnungsplan P (kurz Plan) zu einem Babel-Programm S und einem Zielausdruck G ist ein Ableitungsbaum und beschreibt eine Folge von Berechnungsschritten. Die Knotenmenge des Baumes besteht aus zwei Klassen. U-Knoten, die in Aufbau und Bedeutung mit den Knoten der Berechnungspläne für Prolog-Programme identisch sind, haben die Form

$$\langle n, l_{up}, l_{down}, Alt, BP\rangle \in (\mathbb{N} \cup \{\perp\}) \times Exp_\Sigma(Var) \times (Exp_\Sigma(Var) \cup \{\perp\}) \times 2^{\mathbb{N}} \times 2^{\mathbb{N}},$$

wobei die einzelnen Komponenten wie in Definition 3.1 beschrieben die Nummer des U-Knotens, der obere und untere Ausdruck und die Mengen der alternativen Regeln und berechneten Backtracking-Punkte sind. Einziger Unterschied zur Definition 3.1 ist die Tatsache, daß die Up- und Down-„Literale“ hier Σ-Ausdrücke sind (l_{up} ist eine vollständige Funktionsapplikation, $l_{down} \neq \perp$ die linke Seite einer Regel).
Die zweite Klasse von Knoten besteht aus den A-Knoten der Form

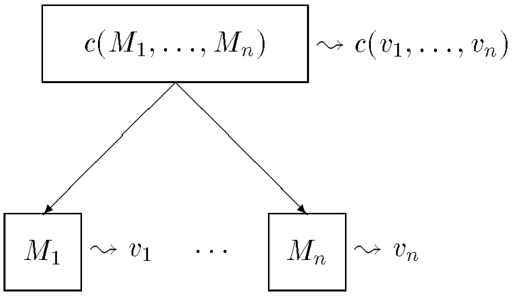
$$\langle M, value_M\rangle \in Exp_\Sigma(Var) \times (Exp_\Sigma(Var) \cup \{\perp\}).$$

$value_M \neq \perp$ ist das vom Programm berechnete Resultat des Ausdruckes M. $value_M = \perp$ deutet einen noch nicht berechneten Wert an.
Die geschlossenen U-Knoten eines Babel-Berechnungsplanes werden wie bei Prolog-Plänen fortlaufend gemäß der Auswertungsstrategie (d.h. depth-first left-to-right) durchnumeriert.

Der initiale Berechnungsplan P_0 zu einem Zielausdruck $G \in Exp_\Sigma(Var)$ besteht nur aus dem A-Knoten $\langle G, \perp \rangle$. □

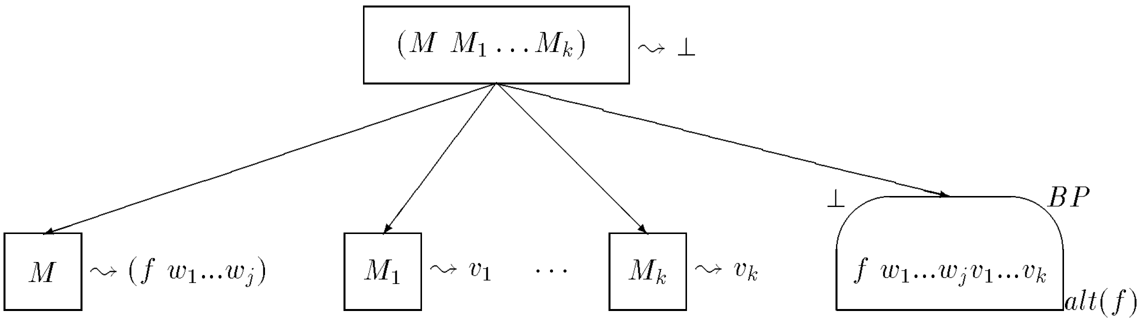
Ein Beispiel eines Babel-Berechnungsplanes gibt Abbildung 8 des nächsten Abschnitts (Seite 26). Hier soll nun kurz anhand eines Ableitungsschrittes (induktiv über die Struktur des Ausdruckes) angedeutet werden, wie die Pläne modifiziert und die Werte von Ausdrücken berechnet werden können ($\langle M, v \rangle$ wird graphisch $\boxed{\mathrm{M}} \leadsto v$ dargestellt):

- Konstruktorapplikation



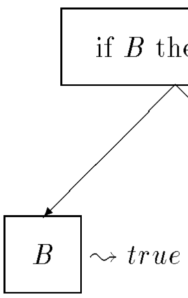
Die Berechnung des Wertes einer Konstruktorapplikation beginnt mit der Auswertung der einzelnen Argumente (innermost). Mit diesen Teilergebnissen wird dann entsprechend der Wert des Gesamtausdruckes konstruiert und im zugehörigen Unifikationsgraphen eingetragen. Dies bewirkt die Einführung einer neuen Klasse $\langle \{c(v_1 \ldots v_n)\}, \emptyset, \emptyset \rangle$ und n Verweise zu den Argumenten $v_i, 1 \leq i \leq n$ (d.h. mit $UG = \langle C, F \rangle$ $\exists f_i \in F$ mit $mark_{UG}(f_i) = \langle i, c(v_1 \ldots v_n), v_i \rangle$).

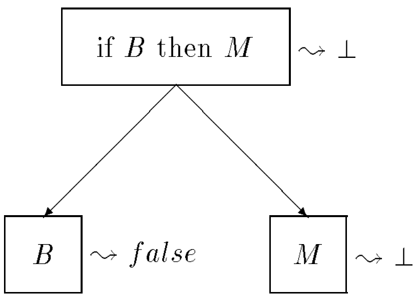
- Vollständige Applikation



Sei der Wert von M eine Funktion ($j = 0$ entspricht üblicher Funktionsapplikation) und $f \in FS^{k+j}$. Dann liegt eine vollständige Funktionsapplikation vor. Nach der Auswertung der $k + 1$ Ausdrücke $M, M_1, \ldots, M_k$ wird der Wert durch eine Applikation der Funktion f bestimmt. $alt(f)$ bezeichnet alle f definierenden Programmregeln, und BP enthält den Vater–U-Knoten (den nächsten U-Knoten zwischen dem hier eingeführten U-Knoten und der Wurzel), falls solch einer existiert. Sonst ist BP leer.

- Bedingter Ausdruck ohne Alternative





Kann der Ausdruck B zu $true \in DC^0$ ausgewertet werden, so ist der Wert des gesamten Ausdruckes gleich dem von M. Wird B zu $false \in DC^0$ ausgewertet, so muß die Berechnung zurückgesetzt werden (Backtracking). Dies ist die einzige Situation, wo nicht aufgrund des Scheiterns einer Unifikation die Berechnung zurückgesetzt wird.
Bedingte Ausdrücke werden speziell im Abschnitt 5.3 behandelt, wo auch die Transformation bedingter Ausdrücke mit Alternative erklärt wird.

Eine mögliche Vereinfachung des Aufbaus von Berechnungsplänen ist die folgende: Wenn der Status eines Ausdruckes M *Normalform* ist (vgl. [Winkler 91]; weitere Stati sind *Narrowing-reduzierbar* und *Failure*), so bedeutet dies, daß kein weiterer Narrowing-Schritt auf diesen Ausdruck anwendbar ist. Dann kann man einem A-Knoten $\langle M, \perp \rangle$ sofort den Wert von M zuweisen ($\langle M, M \rangle$). Unterhalb des Knotens kommt es zu keiner Unifikation, so daß dieser Teil der Berechnung (i.a. Konstruktorapplikationen und partielle Applikationen) für das Intelligente Backtracking nicht von Interesse ist.

5.2 Bezug zwischen funktionaler Berechnung und Ergebnis

Der Wert eines Ausdruckes $(f\ M_1 \ldots M_n)$ wird durch eine Funktionsapplikation $(f\ v_1 \ldots v_n)$ bestimmt, falls die Ausdrücke M_i zu v_i $(1 \leq i \leq n)$ berechnet wurden. Der Berechnungsplan notiert eine Regeanwendung durch das Schließen des U-Knotens und die Einführung eines A-Knotens für die rechte Regelseite als Sohn des geschlossenen U-Knotens.
Nun ist es naheliegend, wie bei Prolog-Berechnungen bzgl. der U-Knoten in Babel-Plänen Intelligentes Backtracking zu realisieren. D.h. die Mengen BP und *Alt* der U-Knoten in Babel-Plänen werden wie in Prolog-Plänen mit dem Vater–U-Knoten bzw. mit der Menge der definierenden Regeln initialisiert. Weiterhin geschieht die Erweiterung des Unifikationsgraphen unter Schließen des U-Knotens im Plan in analoger Weise. Ein Clash vergrößert die Menge BP, und eine erfolgreiche Unifikation ändert die Label-Mengen der Klassen mit den neuen Bindungen.
Doch das folgende Beispiel zeigt den bisher nicht realisierten Bezug zwischen einer Berechnung und ihrem Ergebnis. Abbildung 8 gibt den interessanten Babel-Berechnungszustand an (linke Seite):
Innerhalb der vorangegangenen Auswertung des Zielausdruckes `goal X'` wurde eine Applikation `f X` zu `b` ausgewertet. Dieses Ergebnis `b` ist im Unifikationsgraphen innerhalb einer Klasse aufgeführt. Mit Hilfe des Ergebnisses `b` kommt es zum Ausdruck `g b`, bei dem die Regelanwendung für die Funktion `g` scheitert. Aber wie muß die Label-Menge der `b` enthaltenden Klasse (und damit im Beispiel die Clash-Menge) beschaffen sein?
Ein Vergleich zur Logik-Programmierung, bei der man eine Funktion durch ihren Graphen darstellt (Prolog ist universell), hilft weiter. Dabei werden innere Funktionsapplikationen nach vorne (innermost) „herausgezogen“ (ähnlich wie in [Bosco et al. 89a] und [Bosco et al. 89b], wo Programme in der (first-order) funktional-logische Sprache *Kernel LEAF* (*K-LEAF*) in eine Zwischensprache *Flat-LEAF* übersetzt und dann mit Unifikation und Resolution ausgewertet werden). Die rechte Seite der Abbildung 8 gibt den entsprechenden Prolog-Berechnungszustand an.
Während einer Funktionsberechnung in einem Prolog-Programm werden die (Zwischen-) Ergebnisse an Variablen gebunden, die nur dem Transport dienen. Für das Intelligente Backtracking kann man folgende Beobachtung machen: Die Label-Mengen der Klassen erweitern sich bei jeder Ergebnisübergabe durch die (immer erfogreiche) Unifikation des Ergebnisses mit einer freien „Transportvariablen“ (im Beispiel `Y` am U-Knoten mit der Nummer 1). Wird ein Zwischenergebnis weiter verwendet (Knoten 2), so besitzt seine Label-Menge die Nummern aller an der Berechnung beteiligten Knoten.

Es liegt die Vermutung nahe, daß die Label-Menge eines Ergebnisses um die Nummern derjenigen (geschlossenen) U-Knoten eines Planes erweitert werden muß, die am Entstehen des Ergebnisses (in einem gewissen funktionalen Sinn) beteiligt sind. Es ist aber nicht von vorne herein klar, ob die Nummern aller U-Knoten zwischen der Wurzel eines Planes und dem Ort der Ergebnisberechnung der Label-Menge dieses Ergebnisses hinzugefügt werden müssen.

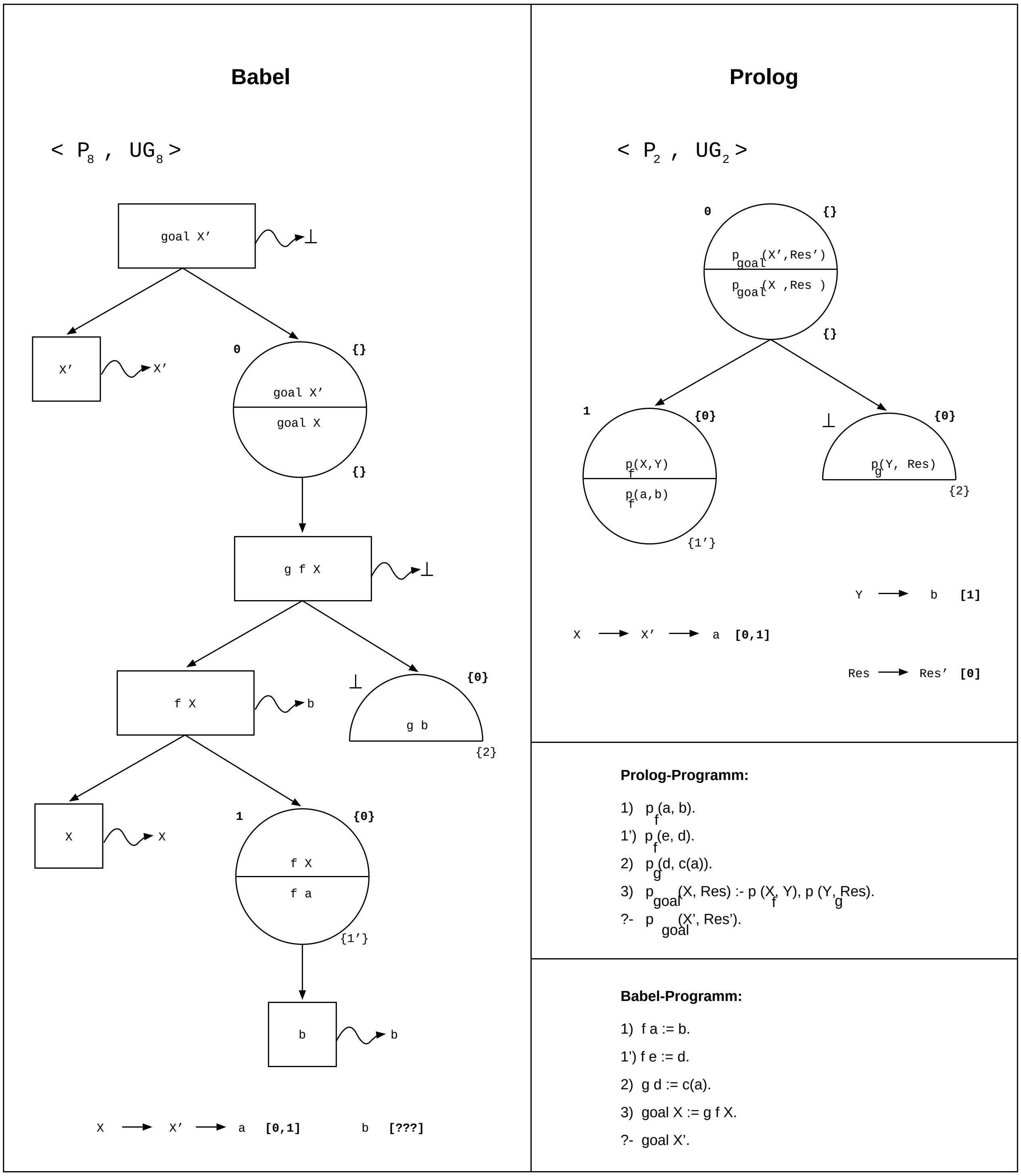
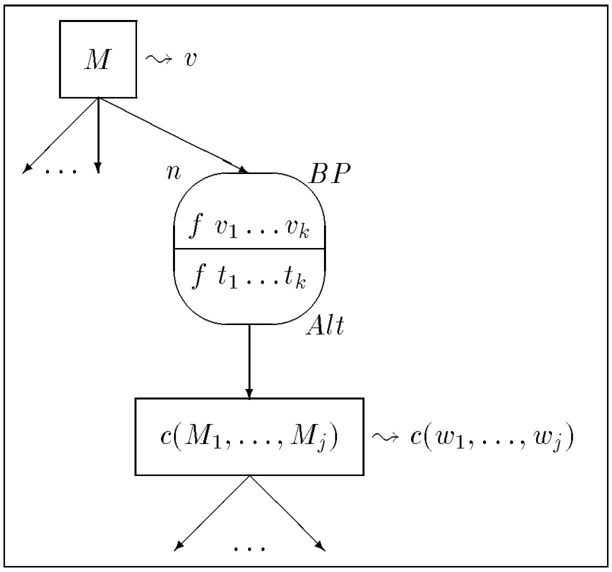
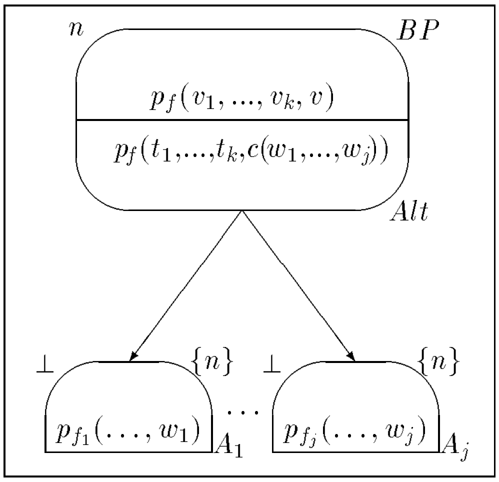


Abbildung 8: Bezug zwischen Berechnung und Ergebnis

Ein systematischer Vergleich (induktiv über die Struktur der rechten Seite der angewandten Regel) zwischen den in Prolog und den in Babel erzeugten Label-Mengen bei der Berechnung zeigt die Unvollständigkeit des oben beschriebenen Verfahrens ebenso wie die für die Vollständigkeit notwendigen weiteren Aktionen (bedingte Ausdrücke und Funktionen höherer Ordnung werden in den separaten Abschnitten 5.3 und 5.4 behandelt). Als Beispiel soll hier eine Konstruktorapplikation als Ergebnis dienen. Eine vollständige Aufführung aller Konstrukte ist in [Koch 92] enthalten. Auf der linken Seite befindet sich ein Ausschnitt eines Babel-Planes, und die rechte Seite zeigt den entsprechenden Ausschnitt eines Prolog-Planes:





Sei $fv_1 \dots v_k$ der betrachtete Funktionsaufruf und $ft_1 \dots t_k := c(M_1, \dots, M_j)$. die versuchte Programmregel. Sind die Argumente jeweils (d.h. aktuelle und formale Argumente paarweise) unter Berücksichtigung der bisherigen Berechnung unifizierbar, so wird der U-Knoten ähnlich der Prolog-Auswertung geschlossen. Der Wert des funktionalen Ausdruckes M ist eine Struktur bestehend aus einem Konstruktor $c \in DC^j, j > 0$ und j Babel-Ausdrücken w_i, $1 \leq i \leq j$, die im Berechnungsplan unterhalb des betrachteten Bereiches berechnet werden.
In Prolog (mit der Darstellung einer Funktion durch ihren Graphen) werden die j Argumente durch entsprechende Literale berechnet. Wie im funktional-logischen Fall geschieht dies unterhalb der betrachteten Stelle im Prolog-Berechnungsplan. Die abschließende „Konstruktion“ des Ergebnisses wird im relationalen Fall vorgezogen und schon beim Schließen des Knotens (mit der Nummer n) initialisiert. Durch diese Unifikation wird die ungebundene Variable v an einen Term gebunden, der durch das Konstruktorsymbol c aufgebaut wird. Auch hier wird die Knotennummer n in die Label-Menge aufgenommen.

Als wesentlichste Erkenntnis kann man dem IB-Verfahren für Prolog im Beispiel entnehmen, daß die Berechnungen der Argumentwerte (w_i) keinen weitergehenden Einfluß auf die Label-Menge des Ergebnisses haben, welches den Knoten mit der Nummer n passiert. Insbesondere gibt es keine transitiven Abhängigkeiten, so daß es für das Intelligente Backtracking bei Babel-Berechnungen wieder genügt, bei der Propaganda des Ergebnisses über den U-Knoten mit der Nummer n (vgl. Abschnitt 6) die Label-Menge der Klasse im Unifikationsgraphen, die das Ergebnis in ihrer Knotenmenge enthält, um gerade diese Nummer n zu erweitern.

In allen in [Koch 92] untersuchten Fällen zeigt sich ein einheitliches Verhältnis zwischen der logischen und der funktional-logischen Auswertung: Die funktionale Wertzuweisung bei der Auswertung eines Babel-Ausdruckes entspricht bei der logischen Auswertung einer weiteren Unifikation, die immer erfolgreich ist und das berechnete Resultat übergibt.
Folgendes Vorgehen bei der funktional-logischen Berechnung führt zu gleichen Label-Mengen in den Unifikationsgraphen wie bei der relationalen Auswertung, und damit zur Vollständigkeit des Verfahrens:

> Wird bei einer Babel-Berechnung ein (dann geschlossener) U-Knoten zum zweiten Mal besucht, so muß bei der Übergabe des Wertes vom Sohn des U-Knotens an den Vater des U-Knotens der Label-Menge der Klasse dieses Wertes die Nummer des U-Knotens hinzugefügt werden. Dadurch erhält man die gleichen Label-Mengen wie im entsprechenden logischen Fall und somit die Volllständigkeit des Verfahrens.
> Die nachfolgende Abbildung 9 zeigt schematisch diese Erweiterung. Die hinter dem Ergebnis notierte Label-Menge wird normalerweise nur im Unifikationsgraphen aufgeführt.

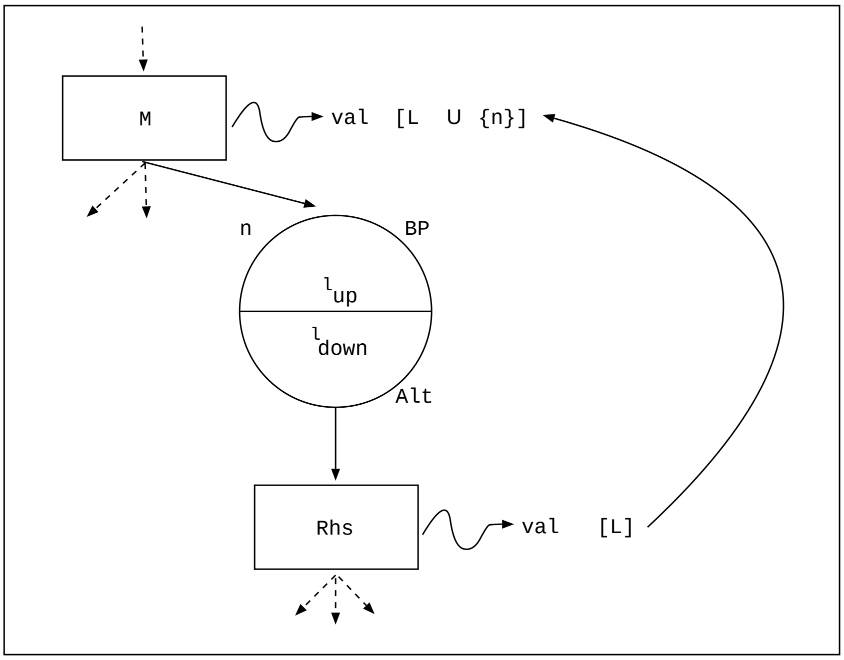


Abbildung 9: Erweiterung von Label-Mengen bei Übergabe des Ergebnisses

Die Berücksichtigung dieses Schemas bewirkt für das Beispiel der Abbildung 8 (Seite 26) die Erweiterung der Label-Menge der Klasse von b um die Nummer 1, und somit die Vollständigkeit des Intelligenten Backtrackings.

Das Verfahren ist korrekt und vollständig für alle benutzerdefinierten Funktionen und wird in Abschnitt 6 formalisiert. Die Vollständigkeit folgt aus der Vollständigkeit des in Abschnitt 3 beschriebenen Verfahrens und der Identität der Label-Mengen zu diesem Verfahren.
Diese Vollständigkeit wird jedoch durch die Anwendung vordefinierter Funktionen, zu denen man auch die bedingten Ausdrücke zählen kann, zerstört. Weiterhin benötigt das Konzept der Funktionen höherer Ordnung eine spezielle Behandlung beim Intelligenten Backtracking.
All dieses wird in den weiteren Abschnitten dieses Abschnitts näherer erläutert.

5.3 Bedingte Ausdrücke und IB

Bedingte Ausdrücke kann man als vordefinierte Funktionen der Sprache Babel ansehen. Für sie gibt es spezielle Narrowing-Regeln (vgl. Abschnitt 4.2), auf denen die Narrowingsemantik basiert. Dieser Abschnitt zeigt die Notwendigkeit einer speziellen Behandlung für bedingte Ausdrücke bzgl. des Intelligenten Backtrackings.

5.3.1 Bedingte Ausdrücke mit Alternative

Ein bedingter Ausdruck mit Alternative hat die Form

$$(\text{if } B \text{ then } M_1 \text{ else } M_2).$$

In Abhängigkeit des Resultates von B wird der erste (M_1) bzw. der zweite (M_2) Ast für die Ermittlung des Resultates des bedingten Ausdruckes verfolgt. Im Berechnungsplan erfolgt die Auswertung von B in einem Teilbaum, der zu den Teilbäumen der Berechnungen der einzelnen Alternativen völlig disjunkt ist. Es erfolgt keine Wertzuweisung oder eine „Übertragung“ des Wertes von B innerhalb des Teilbaumes unter dem Knoten, der den bedingten Ausdruck repräsentiert. Zwei Fragestellungen können den Sachverhalt verdeutlichen:

- Wie kann man innerhalb des IB-Verfahrens eine Neuauswertung (Anwendung verbliebender Alternativen) der Bedingung B erreichen, falls die Berechnung im Teilbaum einer Alternative (M_1 oder M_2) scheitert?
- Wie kann man innerhalb des IB-Verfahrens eine Neuauswertung der Bedingung B erreichen, wenn das Resultat (eines Zweiges) des bedingten Ausdruckes am späteren Scheitern der Berechnung beteiligt ist?

Die Suche nach einem anderen Resultat für die Bedingung ist erforderlich, um evtl. die anderen Alternativen auswerten zu können. Das folgende Beispiel erläutert besonders die zweite Fragestellung (hinter den Resulaten der Ausdrücke sind die Label-Mengen aufgeführt):

Beispiel

1) f X := if (q X) then (h d) else (j b).
2) q a := true.
2') q b := false.
3) h d := c.
4) j b := b.
5) g b := erfolg.

?- g f X'.

Der in Abbildung 10 (teilweise) angegebene Berechnungszustand zeigt die unvollständige Label-Menge des Resultates der Applikation $f\ X'$. Die Clash-Menge der scheiternden Unifikation enthält nicht die Nummer 1, obwohl der U-Knoten mit der Nummer 1 für die Resultatsermittlung (offensichtlich) wesentlich ist. Das bisher beschriebene IB-Verfahren würde zu einer Termination der Berechnung ohne Lösungsfund führen. Das Beispiel zeigt deutlich die Notwendigkeit des Bezuges zwischen dem Resultat der Bedingung und der Auswahl der entsprechenden Alternative.

Ein möglicher Weg ist die explizite Aufnahme der impliziten Funktionen in das Programm. Denn der letzte Abschnitt zeigt ein korrektes und vollständiges Vorgehen für benutzerdefinierte Funktionen. Dieses läßt sich dadurch erreichen, daß man jeden einzelnen bedingten Ausdruck mit Alternative durch die Applikation einer neuen Regel ersetzt:

$$(\ldots(\text{ if } B \text{ then } M_1 \text{ else } M_2)\ldots)$$

wird ersetzt durch

$$(\ldots(f_{neu}\ B\ X_1 \ldots X_k)\ldots),$$

wobei $M_1, M_2 \in Exp_\Sigma(\{X_1, \ldots, X_k\})$ und f_{neu} durch folgende zwei Regeln definiert wird:

$$f_{neu}\ true\ X_1 \ldots X_k \ := M_1.$$
$$f_{neu}\ false\ X_1 \ldots X_k := M_2.$$

Diese Transformation entspricht auch der Realisierung eines bedingten Ausdruckes in Prolog. Obiges Beispielprogramm in Prolog-Notation:

Beispiel

1) p_f(X,Res) :- p_q(X,B), p_{if}(B,Res).
2) p_q(a,true).
2') p_q(b,false).
3) p_h(d,c).
4) p_j(b,b).
5) p_g(b,erfolg).
6) p_{if}(true,V) :- p_h(d,V).
6') p_{if}(false,V) :- p_j(b,V).

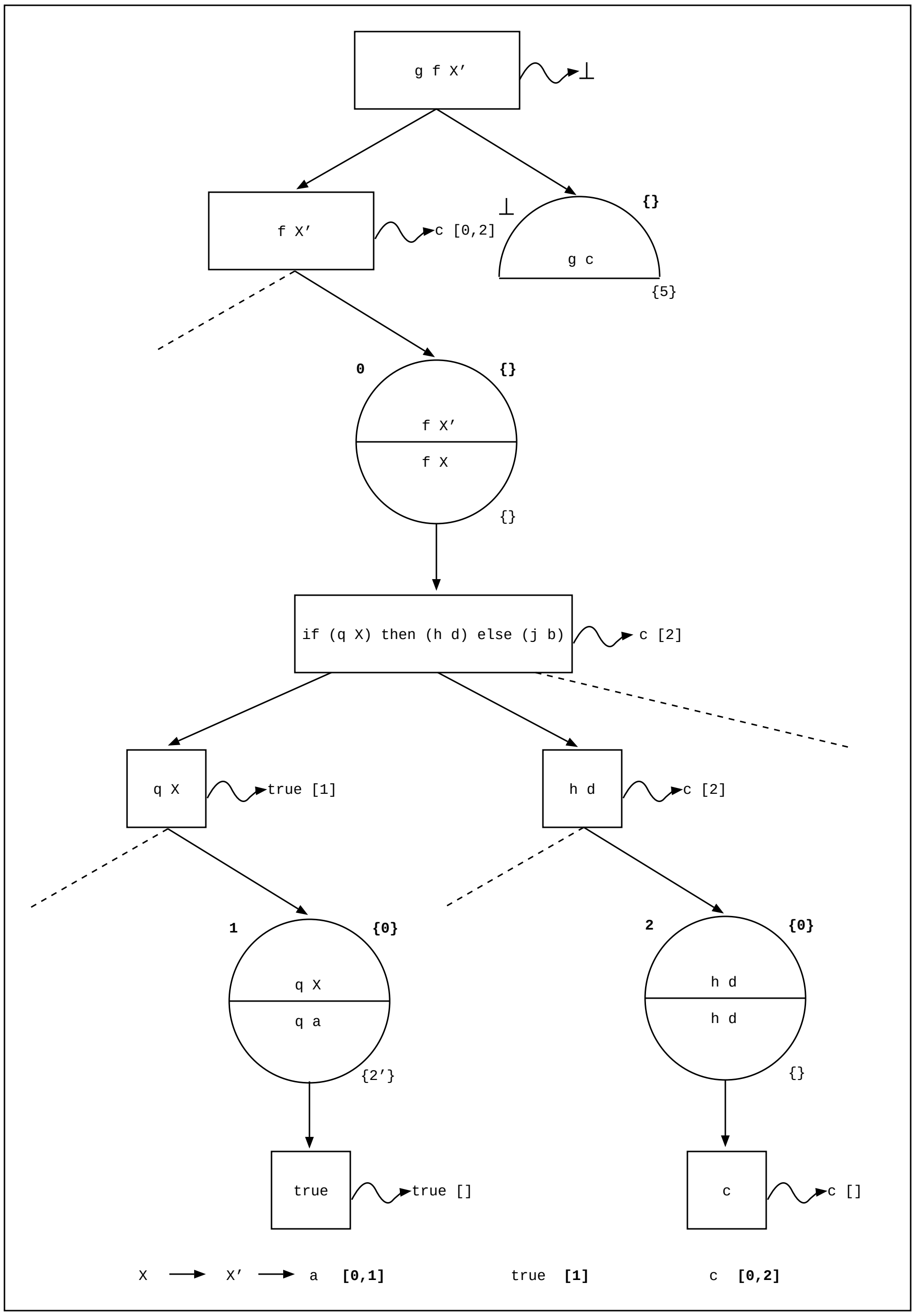


Abbildung 10: Die vordefinierte Funktion *if-then-else*

Die Transformation der bedingten Ausdrücke mit Alternative in Babel und die Formulierung in Prolog zeigen den notwendigen Bezug zwischen der ausgewerteten Bedingung und der Verfolgung einer Alternative: Nach der Berechnung der Bedingung existiert im Unifikationsgraphen eine Klasse, deren Knotenmenge das Resultat der Bedingung enthält. Die Label-Menge dieser Klasse enthält u.a. alle Nummern von U-Knoten, die an der Bildung des Resultates der Bedingung beteiligt sind. Diese Label-Menge muß auch für die Verfolgung einer Alternative („then“- oder „else“- Zweig) verantwortlich gemacht werden.

Im transformierten Babel-Programm wird für einen bedingten Ausdruck mit Alternative des ursprünglichen Programms ein U-Knoten angelegt. Dieser U-Knoten hat als initiale *Alt*-Menge die beiden Regeln der entsprechenden neuen Funktion. Die Auswahl der Alternative erfolgt im Berechnungsplan unterhalb dieses U-Knotens. Bevor nun von diesem U-Knoten aus die Berechnung tiefer zurückgesetzt

wird, müssen beide Regeln für die neue Funktion probiert werden. Dabei kommt es dann zu Clashes, die die Menge BP gerade um die Label-Menge des Resultates der Bedingung erweitern. Da die Menge BP das Intelligente Backtracking steuert, wird auch die Neuauswertung der Bedingung B initiiert, falls dies erforderlich ist.
Durch die Transformation der bedingten Ausdrücke mit Alternative erreicht man eine einheitliche Behandlung dieses Sprachkonstruktes mit den benutzerdefinierten Funktionen.

5.3.2 Bedingte Ausdrücke ohne Alternative

Ein bedingter Ausdruck ohne Alternative bzw. die rechte Seite einer Regel mit Guard hat die Form

$$(\text{if } B \text{ then } M).$$

Es liegt die Versuchung nahe, die bedingten Ausdrücke ohne Alternative in ähnlicher Weise wie die bedingten Ausdrücke mit Alternative zu transformieren:

$$(\ldots(\text{ if } B \text{ then } M)\ldots)$$

wird ersetzt durch

$$(\ldots(f_{neu}\ B\ X_1 \ldots X_k)\ldots),$$

wobei $M \in Exp_\Sigma(\{X_1, \ldots, X_k\})$ und f_{neu} durch folgende Regel definiert wird:

$$f_{neu}\ true\ X_1 \ldots X_k := M.$$

Diese Transformation ist aber nicht immer möglich. Betrachtet man z.B. die Regel

$$f\ X := \text{if } (g\ Y) \text{ then } c(X).$$

Im Guard der Regel kommt eine freie Variable Y vor. Eine Transformation der Regel zu

$$f\ X := f_{neu}\ (g\ Y)\ X.$$
$$f_{neu}\ true\ X := c(X).$$

ist wegen der Restriktion freier Variablen innerhalb einer Programmregel nicht möglich. (vgl. Abschnitt 4, Seite 21).

Auf der anderen Seite zeigt dieser Transformationsversuch ebenso wie ein Vergleich zu Prolog-Programmen, daß solch eine Transformation nur einen Zweck hätte: Die Ursache des Zurücksetzens einer Berechnung würde auf das Scheitern einer Unifikation beschränkt! Diese Eigenschaft wird bei der Auswertung von Babel-Programmen nur durch die Tatsache zerstört, daß bei der Auswertung der Bedingung eines bedingten Ausdruckes ohne Alternative zu $false$ direkt (bisher Naives) Backtracking angestoßen wird. Weitergehende Auswirkungen sind nicht festzustellen.

An folgendem Ausschnitt einer entsprechenden Prolog-Klausel läßt sich das Verhalten, welches man mit der nicht immer erlaubten Transformation bedingter Ausdrücke ohne Alternative in Babel-Programmen erzielen würde, erläutern: Ein entsprechender Ausdruck

$$(\text{if } B \text{ then } M)$$

läßt sich in Prolog als

$$\ldots \text{ :- } \ldots, p_B(\ldots, v_B), p_{true}(v_B), p_M(\ldots, Ergebnis)$$

formulieren. Wird B zu $true$ ausgewertet, so ist $p_{true}(v_B)$ erfolgreich, ohne das der zugehörige Unifikationsgraph verändert wird (d.h. alle Label-Mengen bleiben kostant). Sollte die folgende Berechnung von M scheitern oder ein Ergebnis liefern, welches zu einem Scheitern der Berechnung an anderer Stelle beiträgt, so wird die Label-Menge von v_B in keiner besonderen Weise für das Intelligente Backtracking

bei Prolog benötigt.
Wenn dagegen B zu $false$ ausgewertet wird, kommt es zum Clash und die Menge BP des Berechnungspunktes für $p_{true}(v_B)$ wird gerade um die Label-Menge des Ergebnisses ($false$) erweitert. Wegen des Fehlens weiterer Alternativen kommt es zum Backtracking mit der Menge BP, die dann nur aus der Label-Menge des Ergebnisses der Bedingung und dem Vater–U-Knoten besteht.

Das beschriebene Verhalten kann in einfacher Weise bei der Entwicklung von Berechnungsplänen realisiert werden. Dies ist auch das im weiteren gewählte Vorgehen.

5.4 Funktionen höherer Ordnung und IB

Die Sprache Babel bietet Funktionen höherer Ordnung als Sprachkonzept an. Mögliche Werte von Babel-Ausdrücken sind neben den Termen auch Funktionen. Ist der Wert eines Ausdruckes eine Funktion, so können dieser Funktion weitere Argumente appliziert werden, oder diese Funktion wird selber einer Funktion höherer Ordnung (Funktional) appliziert.
Aus diesem Grund muß die Definition des Unifikationsgraphen um gerade die Funktionen, die als Teilresultate in einer Berechnung auftreten können, als zusätzliche Elemente der Knotenmenge der Klassen erweitert werden. Wie auch schon bei den Konstruktoren werden funktionale Abhängigkeiten zwischen den (partiell applizierten) Argumenten und der Funktion selber durch Kanten des Unifikationsgraphen dargestellt.

Zunächst wird die Knotenmenge einer Klasse um Funktionen höherer Ordnung erweitert (vgl. Definition 3.2). Dabei kommen nicht alle Funktionsapplikationen als Knoten einer Klasse in Frage, sondern nur die partiell applizierten Funktionen in Normalform (vgl. [Winkler 91]), d.h. Ausdrücke, die keine Terme darstellen und nicht weiter durch Narrowing reduzierbar sind. Die Knoten einer Klasse sind Elemente der Menge der Babel-Ausdrücke in Normalform $NF\_Exp_\Sigma(Var)$, die die kleinste Menge ist, die sich folgendermaßen induktiv beschreiben läßt:

- $Var \subseteq NF\_Exp_\Sigma(Var)$
- $f \in FS^n, j < n,\ N_1, \ldots, N_j \in NF\_EXP_\Sigma(Var) \Longrightarrow (f\ N_1 \ldots N_j) \in NF\_EXP_\Sigma(Var)$
- $c \in DC^n, N_1, \ldots, N_n \in NF\_EXP_\Sigma(Var) \Longrightarrow c(N_1, \ldots, N_n) \in NF\_EXP_\Sigma(Var)$

Definition 5.2 (Babel-Klasse) Eine Klasse $c = \langle V, E, Lab\rangle$ setzt sich zusammen aus einem gerichteten azyklischen Graphen $\langle V, E\rangle$ mit der Knotenmenge $V \subseteq NF\_Exp_\Sigma(Var)$ und der Kantenmenge $E \subset V \times V$, und aus einer Label-Menge $Lab \subset \mathbb{N}$, die eine Menge von Nummern geschlossener Knoten eines bestimmten Babel-Berechnungsplanes ist.
Jede Babel-Klasse besitzt einen ausgezeichneten Knoten, zu dem es von jedem anderen Knoten der Klasse aus einen Pfad gibt. Er heißt Repräsentant der Klasse.
Zu jeder Klasse c gibt es die Funktion $deref_c : V \longrightarrow V$, die einen Knoten der Klasse auf den Repräsentanten der Klasse abbildet. □

Weiterhin müssen zwischen den Klassen des Unifikationsgraphen Kanten erlaubt sein, die die Beziehung einer partiellen Applikation zu ihren Argumenten darstellt (vgl. Definition 3.3).

Definition 5.3 (Babel-Unifikationsgraph) Ein Unifikationsgraph UG ist ein gerichteter azyklischer Graph $\langle C, F\rangle$, dessen Knotenmenge C aus einer Menge von Babel-Klassen besteht. Die Kanten stellen funktionale Abhängigkeiten dar und verlaufen vom Repräsentanten einer Klasse (der eine Struktur sein muß) zu einem Knoten einer anderen Klasse. Wie bei Prolog-Unifikationsgraphen kann man zu einem Babel-Unifikationsgraphen eine Markierungsfunktion

$$mark_{UG} : F \longrightarrow \mathbb{N} \times NF\_Exp_\Sigma(Var) \times NF\_Exp_\Sigma(Var)$$

angeben, die jede Kante $f \in F$ auf ein Tripel bestehend aus den Komponenten Argumentposition, Ursprungs- und Zielausdruck abbildet.
Der initiale Unifikationsgraph UG_0 einer Berechnung ist der leere Graph ohne Knoten (und Kanten) $UG_0 = \langle\emptyset, \emptyset\rangle$.
$\mathcal{UG}$ bezeichnet die Menge aller Unifikationsgraphen. □

Bemerkung Ein Unifikationsgraph stellt wie im relationalen Fall die Substitution dar, die sukzessive durch die notierten Bindungen aufgebaut wird, wobei nun eine Variable auch an einen Ausdruck in Normalform gebunden werden kann ($\sigma : Var \longrightarrow NF\_Exp_\Sigma(Var)$). Die Bemerkung zur Definition eines Prolog-Unifikationsgraphen auf Seite 11 gilt entsprechend. □

Funktionen sind im Unifikationsgraphen Repräsentanten ihrer Klassen. Genau wie im Abschnitt 5.2 wird die Label-Menge einer Funktion erweitert, wenn diese als Wert eines Ausdruckes „über einen U-Knoten von unten nach oben“ propagiert wird (vgl. Abschnitt 6). Dazu ein Beispiel:

Beispiel

1) f a := g a.
1') f b := g b.
2) h d(F) := (F a).
3) g b a c := d.

?- ((h d(f Z)) c).

Der Unifikationsgraph der Abbildung 11 zeigt die Erweiterung der Label-Mengen der partiellen Applikationen $g\ a$ und $g\ a\ a$ bei ihrer Propaganda um die Nummern 0 bzw. 1. Dennoch sind die Label-Mengen im Beispiel unvollständig: Die Clash-Menge, die beim Versuch des Schließens des dritten U-Knotens produziert wird, ist leer. Der offensichtliche Bezug der bisher geschlossenen U-Knoten an der vollständigen Funktionsapplikation des offenen U-Knotens wird nicht erkannt und die Lösung durch Anwendung der Regeln `1'),2)` und `3)` nicht gefunden.

Ähnlich wie in früheren Abschnitten dieses Abschnitts wird das bekannte Verhalten von Sprachkonstrukten zur Untersuchung neuer Konzepte benutzt. Hier wird nicht wie im Abschnitt 5.2 ein Vergleich der Auswertungen von Prolog- und Babel-Programmen angestellt, sondern das Konzept der Funktionen höherer Ordnung mit in ihrem Verhalten bekannten Konstrukten der eigenen Sprache (Babel) verglichen.

Das Konzept der Funktionen höherer Ordnung kann man bis zu einem gewissen Grad durch Sprachelemente erster Ordnung simulieren: Partiell applizierte Funktionen werden in Konstruktoren „versteckt“, und die Applikation weiterer Argumente erfolgt durch eine neue Funktion *apply*. Zur Anschauung folgen die erforderlichen zusätzlichen Funktionsdefinitionen für die dreistellige Funktion g des obigen Beispiels:

Beispiel

3) g b a c := d.
3') apply c_{g_0} S := c_{g_1}(S).
3'') apply c_{g_1}(X) Y := c_{g_2}(X,Y).
3''') apply c_{g_2}(U,V) W := g U V W.
3'''') apply c_{h_0} T := h T.

?- apply (h d(f Z)) c.

Diese Transformation kann man folgendermaßen beschreiben:
Für ein Funktionssymbol $f \in FS^n, n \geq 1$ werden n neue Konstruktorsymbole $c_{f_i} \in DC^i$, $0 \leq i < n$ eingeführt. $c_{f_i}(M_1, \ldots, M_i)$ stellt eine $(n-i)$-stellige Funktion dar, entstanden aus der partiellen Applikation der i Argumente $M_1 \ldots M_i$ an die n-stellige Funktion f. Weiterhin wird eine neue Funktion $apply \in FS^2$ benötigt, die den neuen Konstruktorsymbolen weitere Argumente “stückweise applizieren“ kann:

$$\begin{aligned} apply\ c_{f_j}(X_1, \ldots, X_j)\ Y &:= c_{f_{j+1}}(X_1, \ldots, X_j, Y),\ 0 \leq j < n-1 \\ apply\ c_{f_{n-1}}(X_1, \ldots, X_{n-1})\ Y &:= (f\ X_1 \ldots X_{n-1}\ Y). \end{aligned}$$

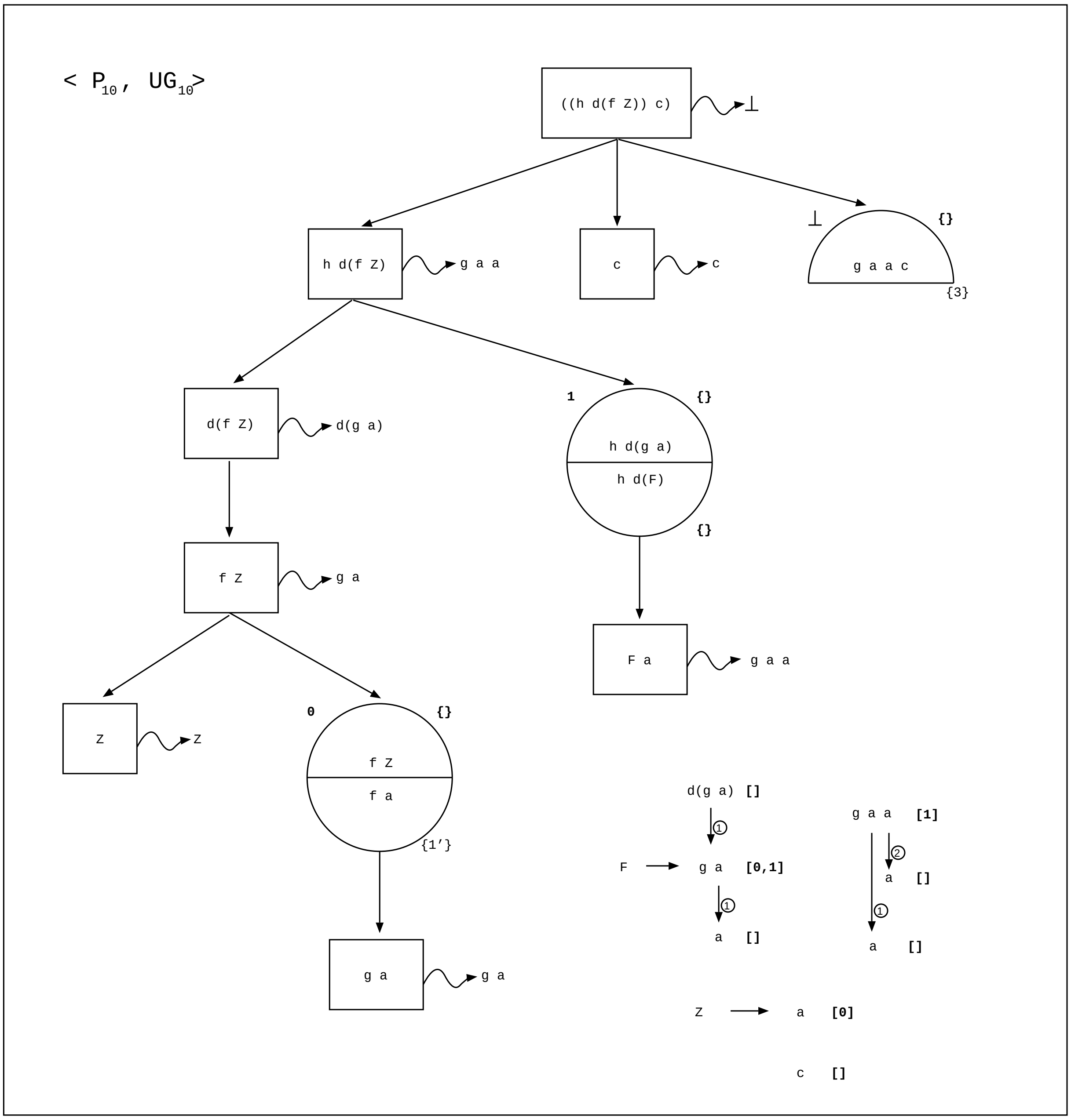


Abbildung 11: Funktionen höherer Ordnung

Eine partielle Applikation $(M\ M_1 \ldots M_r)$ wird durch *apply* $(...(apply\ M\ M_1)...M_r)$ ersetzt. Enthält ein Babel-Programm die möglicherweise partiell applizierten Funktionen $f_1, \ldots, f_k \in FS$ mit den Stelligkeiten $l_1, \ldots, l_k$, dann erfordert das transformierte Programm gegenüber dem ursprünglichen $\sum_{t=1}^{k} l_t$ neue Konstruktorsymbole und die gleiche Anzahl neuer Regeln der Funktion *apply*. Die Berechnung der Anzahl neuer Regeln für die Funktion *apply* zeigt, daß eine Programmtransformation nicht der ideale Weg ist. Die Explosion der Regel-Anzahl bewirkt neben einer Aufblähung des Codes durch die Vielzahl der bestehenden Alternativen für die (bei gegebenem ersten Argument deterministische) Funktion *apply* viele scheiternde Unifikationsversuche. Deshalb wird die Manipulation der Label-Mengen des transformierten Programms auf die Auswertung des nichttransformierten Programms übertragen. Aus Platzgründen kann in dieser Arbeit nur das Verfahren, nicht aber die Herleitung und Alternativen angegeben werden (vgl. [Koch 92]):

Die Label-Menge einer als Resultat hochgereichten Funktion wird um die Nummern der übersprungenen U-Knoten erweitert. Bei einer partiellen Applikation erhält die Klasse der neuen Applikation die Label-Menge der alten Applikation, und bei einer vollständigen Applikation vergrößert sich die Menge BP des zu schließenden U-Knotens um die Label-Menge der partiellen Applikation. Denn durch die Aufnahme der Berechnungspunkte in die Menge BP werden diese Punkte für die unter dem Knoten liegende Berechnung verantwortlich gemacht. Diese Behandlung des Sprachkonzeptes „Funktionen höherer Ordnung" garantiert ausreichend große Label-Mengen im Unifikationsgraphen und im Berechnungsplan, und sichert damit die Vollständigkeit des Vorgehens.

6 Ein IB-Verfahren für Babel

Das in diesem Abschnitt vorgestellte Verfahren des Intelligenten Backtrackings bei der Auswertung von funktional-logischen Babel-Programmen ist auf den ersten Blick dem IB-Verfahren für logische Programme aus Abschnitt 3 sehr ähnlich. Dies ist verständlich, da das Verfahren für logische Programme als Ausgangsbasis für das neuentwickelte Verfahren diente. Trotz vieler Gemeinsamkeiten existieren aber auch grundlegende Unterschiede zwischen den beiden Verfahren. Grundlage der Beschreibung des IB-Kalküls sind die Definitionen 5.1 (Babel-Berechnungsplan) und 5.3 (Babel-Unifikationsgraph). Wie im letzten Abschnitt beschrieben werden bedingte Ausdrücke mit Alternative in einem Babel-Programm transformiert (vgl. Abschnitt 5.3.1).
Die Berechnung eines Babel-Programms wird als Folge von Paaren bestehend aus einem Babel-Berechnungsplan P_i und dem zugehörigen Babel-Unifikationsgraphen UG_i dargestellt:

$$\langle P_0, UG_0\rangle, \langle P_1, UG_1\rangle, \langle P_2, UG_2\rangle, \ldots$$

Der Übergang von einem Berechnungszustand zum nächsten erfolgt mittels eines expandierenden Berechnungsschrittes oder durch ein Zurücksetzen der Berechnung. Diese beiden Möglichkeiten der Berechnungsfortsetzung werden im folgenden näher betrachtet.

6.1 Expandierender Berechnungsschritt

Ein Babel-Berechnungsplan enthält zwei Arten von Knoten: A-Knoten und U-Knoten (vgl. Definition 5.1). An beiden Knotentypen wird etwas berechnet: Die A-Knoten werden expandiert und bekommen bei erfolgreicher Berechnung einen Wert zugewiesen, und an den U-Knoten erfolgt entweder die Unifikation eines Ausdruckes mit einer linken Regelseite oder es wird Intelligentes Backtracking angestoßen.

Der aktuelle Berechnungspunkt (der Knoten, an dem die Berechnung fortgesetzt wird) kann sowohl ein A-Knoten ohne zugewiesenen Wert als auch ein offener U-Knoten sein. Deshalb ist auch die Definition des expandierenden Berechnungsschrittes (Ableitungsschritt) zweigeteilt: Ein Ableitungsschritt an einem U-Knoten ist im wesentlichen identisch mit dem Ableitungsschritt in Abschnitt 3. Ein Ableitungsschritt an einem A-Knoten kann u.U. sehr komplex sein.

Definition 6.1 (Ableitungsschritt) Grundlage dieser Definition ist die Definition 5.1 für Babel-Berechnungspläne und 5.3 für Babel-Unifikationsgraphen.
Gegeben sei ein Berechnungsplan P_i mit zugehörigem Unifikationsgraph UG_i, wobei die Wurzel des Planes noch keinen Wert zugewiesen bekam ($\langle M_{root}, \perp\rangle$). Dann existiert bei der realisierten Strategie als aktueller Berechnungspunkt ein erstes Blatt (Knoten ohne Söhne) bzgl. der Ordnung depth-first left-to-right, das entweder ein offener U-Knoten $\langle \perp, l_{up}, \perp, Alt, BP\rangle$ oder ein A-Knoten ohne zugewiesenen Wert $\langle M, \perp\rangle$ ist. Die Definition des zum Folgeplan P_{i+1} mit zugehörigem Unifikationsgraph UG_{i+1} führenden Ableitungsschrittes ist je nach Knotentyp des aktuellen Berechnungspunktes unterschiedlich:

- Falls der aktuelle Berechnungspunkt ein A-Knoten der Form $\langle M, \perp\rangle$ ist, unterscheidet die Definition weiter nach der Struktur des Ausdruckes M. Befindet sich M in Normalform, so wird diesem und evtl. weiteren A-Knoten des Berechnungsplanes ein Wert zugewiesen. Anderenfalls wird M durch die Einführung von Söhnen expandiert.

- Sei $M \notin NF\_Exp_\Sigma(Var)$. Dann erhält man den Berechnungsplan P_{i+1} durch die Einführung gewisser Söhne des aktuellen Berechnungspunktes, wobei Art und Anzahl der Söhne wiederum vom Aufbau von M abhängen. Der Unifikationsgraph bleibt bei diesem Ableitungsschritt konstant ($UG_{i+1} = UG_i$). Es folgt die Unterscheidung nach der Struktur von M:
  * $c(M_1, \ldots, M_r), r \geq 1$: Als neue Söhne von M werden die A-Knoten $\langle M_1, \perp\rangle, \ldots, \langle M_r, \perp\rangle$ eingeführt.
  * $(f\ M_1 \ldots M_r), f \in FS^r$: Neue Söhne sind die A-Knoten $\langle M_1, \perp\rangle, \ldots, \langle M_r, \perp\rangle$ und der U-Knoten $\langle \perp, (f\ ?_1 \ldots ?_r), \perp, alt(f), BP\rangle$, wobei BP nur den Vorgänger–U-Knoten enthält, falls ein solcher existiert (sonst ist BP leer).
  * $(f\ M_1 \ldots M_s),\ f \in FS^r, r > s$: Neue Söhne werden die A-Knoten $\langle M_1, \perp\rangle, \ldots, \langle M_s, \perp\rangle$.
  * $(M\ M_1 \ldots M_r)$ vollständige Applikation : Als neue Söhne werden die A-Knoten $\langle M, \perp\rangle$,$\langle M_1, \perp\rangle, \ldots$, $\langle M_r, \perp\rangle$ und der U-Knoten $\langle \perp, (?\ ?_1 \ldots ?_r), \perp, ?, BP\rangle$ eingeführt (für BP gilt das gleiche wie oben).
  * $(M\ M_1 \ldots M_s)$ partielle Applikation: Neue Söhne sind $\langle M, \perp\rangle$,$\langle M_1, \perp\rangle, \ldots$, $\langle M_r, \perp\rangle$
  * (if B then M) : Neue Söhne werden die beiden A-Knoten $\langle B, \perp\rangle$ und $\langle M, \perp\rangle$.
- Falls $M \in NF\_Exp_\Sigma(Var)$ (beispielsweise $M \in T_\Sigma(Var)$), so ist der Wert des Ausdruckes mit dem Ausdruck identisch. Deshalb enthält P_{i+1} den Knoten $\langle M, M\rangle$. Dieser Wert M und alle seine Teilterme werden im Unifikationsgraphen UG_{i+1} zusammen mit den entsprechenden funktionalen Beziehungen aufgenommen.
  Neben der Wertzuweisung am aktuellen Berechnungspunkt kann es weiter möglich sein, gewissen anderen A-Knoten, die Vorfahren des betrachteten Knotens sind, ihren Wert zuzuweisen. Diese Möglichkeit der Propaganda der Wertzuweisung in Richtung der Wurzel hängt von der Art und Struktur des Vaterknotens des aktuellen Berechnungspunktes ab:

  Ist der Vater im Berechnungsplan ein U-Knoten, so wird das Ergebnis über den (geschlossenen) U-Knoten propagiert, wobei im einzelnen folgende Schritte erforderlich sind:

  1. Im Unifikationsgraph wird die Label-Menge des propagierten Ergebnisses um die Nummer des übersprungenen U-Knotens erweitert.
  2. Der Vater des U-Knotens bekommt ebenfalls den hochgereichten Wert zugewiesen und wird aktueller Berechnungspunkt.

  Die Propaganda wird sukzessive weitergeführt, falls die Wurzel des Berechnungsplanes noch keinen Wert zugewiesen bekam

  Als zweites kann der Vater-Knoten des aktuellen Berechnungspunktes einen Ausdruck darstellen. Dann ist für die Propaganda von Werten die Unterscheidung nach der Stellung des aktuellen Knotens bzgl. seiner Brüder wesentlich:

  * Fallls der aktuelle Berechnungspunkt nicht der jüngste (letzte) Sohn des Vaterknotens (bzgl. depth-first left-to-rigth) und nicht die Bedingung eines bedingten Ausdruckes ohne Alternative ist, so ist die Propaganda beendet, da für die Wertzuweisung am Vaterknoten weitere Berechnungen notwendig sind.
  * Stellt der aktuelle Berechnungspunkt die Bedingung eines bedingten Ausdruckes ohne Alternative dar, so ist der zugewiesene Wert wesentlich (Sonderbehandlung wegen einer vordefinierten Funktion).
    Ist der Wert der Bedingung gleich $true$, so ist die Propaganda beendet. Beim Wert $false$, muß an dieser Stelle Backtracking erfolgen. Als Clash-Menge wird die Label-Menge der Klasse von $false$ benutzt. P_{i+1} und UG_{i+1} werden dann durch Intelligentes Backtracking bestimmt.

* Als letzter Fall ist der aktuelle Berechnungspunkt jüngster Sohn des Vaterknotens . Dann sind alle Berechnungen für die Wertzuweisung des Vaters gemacht, wobei wieder nach der Struktur des Vaters unterschieden wird:
  - Repräsentiert der Vater eine Konstruktorapplikation $\langle c(M_1,\ldots,M_r),\perp\rangle$, $r \geq 1$, und besitzt er die Söhne $\langle M_1,v_1\rangle,\ldots,\langle M_r,v_r\rangle$, so bekommt er den Wert $c(v_1,\ldots,v_r)$ zugewiesen ($\langle c(M_1,\ldots,M_r),c(v_1,\ldots,v_r)\rangle$). Weiterhin wird der neue Wert im Unifikationsgraphen zusammen mit den entsprechenden funktionalen Beziehungen eingetragen (durch eine neue Klasse $\langle\{c(v_1,\ldots,v_r)\},\emptyset,\emptyset\rangle$ und r Kanten zu den jeweiligen Argumentklassen).
  - Falls der Vater eine partielle Funktionsapplikation $\langle(f\ M_1,\ldots,M_r),\perp\rangle$ darstellt, bekommt er als Wert $(f\ v_1\ldots v_r)$ zugewiesen, sofern die Söhne $\langle M_i,v_i\rangle, 1 \leq i \leq r$, sind. Wie im ersten Fall wird das Ergebnis mit allen Beziehungen im Unifikationsgraphen eingetragen.
  - Stellt der Vaterknoten eine partielle Applikation $\langle(M\ M_1\ldots M_r),\perp\rangle$ dar, $M \notin FS$, und sind die Teilausdrücke zu $v, v_1,\ldots,v_r$ ausgewertet worden, so bekommt der Vaterknoten als Wert $(v\ v_1\ldots v_r)$ zugewiesen. Neben der wie oben erforderlichen Eintragung in den Unifikationsgraphen muß die Label-Menge des neuen Ergebnisses noch um die Label-Menge von v erweitert werden (Sonderbehandlung bei Funktionen als Ausdruckswerte; vgl. Abschnitt 5.4).
  - Als letztes kann der Vater einen bedingten Ausdruck ohne Alternative $\langle(\text{if } B \text{ then } M),\perp\rangle$ beschreiben. Die Söhne haben dann die Form $\langle B, true\rangle$ und $\langle M,v\rangle$. Dann wird im Berechnungsplan der Vaterknoten zu $\langle(\text{if } B \text{ then } M),v\rangle$ geändert, wobei der Unifikationsgraph konstant bleibt.

  Für alle Fälle, bei denen der Vater des aktuellen Berechnungspunktes ein A-Knoten mit zugewiesenem Wert ist, wird dieser Vaterknoten aktueller Berechnungspunkt und die Propaganda seines Wertes fortgesetzt, falls er nicht die Wurzel des Planes ist.

- Als zweite Möglichkeit bei der generellen Unterscheidung für die Definition des Ableitungsschrittes kann der aktuelle Berechnungspunkt ein U-Knoten sein. In diesem Fall repräsentiert der Vater-Knoten eine vollständige Applikation der Form $(M\ M_1\ldots M_r)$ oder $(f\ M_1\ldots M_r)$.
  Hat der aktuelle Berechnungspunkt die Form $\langle\perp,(?\ ?_1\ldots ?_r),\perp,?,BP\rangle$ (mit einem Vater der ersten Form), so besitzt der betrachtete U-Knoten $(r+1)$ Brüder der Form $\langle M,(f\ w_1\ldots w_k)\rangle$ und $\langle M_i,v_i\rangle, 1 \leq i \leq k$. In einem ersten Schritt ändert sich die zweite Komponente des U-Knotens zu $(f\ w_1\ldots w_k\ v_1\ldots v_r)$ und die vierte Komponente zu $alt(f)$ (Menge aller f definierenden Regeln). Weiterhin muß die Menge BP des Knotens um die Label-Menge von $(f\ w_1\ldots w_k)$ (Resultat des ältesten Bruders) im Unifikationsgraphen erweitert werden (Sonderbehandlung für Funktionale; vgl. Abschnitt 5.4).
  Desweiteren kann der Berechnungspunkt die Form $\langle\perp,(f\ ?_1\ldots ?_r),\perp,Alt,BP\rangle$ (Vater zweiter Art) besitzen. Dann existieren die Brüder $\langle M_i,v_i\rangle, 1 \leq i \leq r$, so daß zunächst die zweite Komponente des U-Knotens zu $(f\ v_1\ldots v_r)$ geändert wird.
  In beiden Fällen kann man bei der weiteren Definition des Ableitungsschrittes von einem Berechnungsknoten der Art $\langle\perp,(f\ u_1\ldots u_s),\perp,Alt,BP\rangle$ ausgehen. Die weiteren Aktionen des Ableitungsschrittes sind mit den in Definition 3.4 aufgeführten sehr ähnlich:

  > Sei m das Minimum der Menge $Alt \neq \emptyset$, und die m-te Babel-Regel habe die Form $f\ t_1\ldots t_s := Rhs$. Dann berechnet ähnlich wie im relationalen Fall die Funktion $extend_B$ (s. Abb. 12) einen allgemeinsten Unifikator der Menge $\{(u_1,\ldots,u_s)\sigma,(t_1,\ldots,t_s)\sigma\}$, $\sigma = subst_{UG_i}$. Dieser Unifikator wird als Erweiterung der bisherigen Substitution σ in den erweiterten Unifikationsgraphen UG_{i+1} eingetragen.
  > Ist die Unifikation obiger Menge nicht möglich, so bestimmt die Funktion $extend_B$ die entsprechende Clash-Menge.

Der Ableitungsschritt an einem U-Knoten unterscheidet die beiden möglichen Ergebnisse des Funktionsaufrufes $extend_B((f\ u_1 \dots u_s), (f\ t_1 \dots t_s), n, UG_i)$:

- Ist die Menge unifizierbar, so ist das Ergebnis des Aufrufes der Funktion $extend_B$ gleich $\langle 0, UG_{i+1}, \emptyset\rangle$. Der zum Folgeplan passende Unifikationsgraph UG_{i+1} enthält alle notwendigen Informationen.
  Der Folgeplan P_{i+1} entsteht aus P_i durch das Schließen des U-Knotens zu $\langle n, (f\ u_1 \dots u_s), (f\ t_1 \dots t_s), Alt \setminus \{m\}, BP\rangle$ und dem Einführen des Sohnes $\langle Rhs, \perp\rangle$ (Rhs ist die rechte Seite der angewandten Regel).
- Ist die Menge nicht unifizierbar, so enthält das Ergebnis des Aufrufes der Funktion $extend_B$ in der dritten Komponente die berechntete Clash-Menge ($\langle 1, UG', Cl\text{-}Set\rangle$). Der Folgeplan wird durch einen rückwärtsgerichteten Berechnungsschritt mit dieser Clash-Menge bestimmt.

□

In der obigen Definition ist ein Ableitungsschritt bzgl. benutzerdefinierter Funktionen definiert. Elementare Sprachelemente wie Funktionen höherer Ordnung und bedingte Ausdrücke, die ebenfalls in der Definition berücksichtigt werden, begründen eine Reihe von Sondermaßnahmen.

Die Funktion $extend_B$ bestimmt, wie in der vorhergehenden Definition angedeutet, die Erweiterung des Unifikationsgraphen. Sie ist in Abbildung 12 ausführlich dargestellt und wird in folgender Definition erläutert.

Definition 6.2 (Erweiterung eines Babel-Unifikationsgraphen) Gegeben sei ein Babel-Unifikationsgraph $UG_i = \langle C, F\rangle$. Ausgehend von einem offenen U-Knoten $v = \langle \perp, (f\ M_1 \dots M_r), \perp, Alt, BP\rangle$ des zugehörigen Berechnungsplanes P_i wird versucht, mit Hilfe einer linken Regelseite $(f\ t_1 \dots t_r)$ diesen Knoten zu schließen und den Folgegraphen UG_{i+1} zu bestimmen. Sei v der n-te U-Knoten (d.h. in P_i sind n-1 U-Knoten geschlossen). Falls die Menge $\{(M_1, \dots, M_r)\sigma, (t_1, \dots, t_r)\sigma\}$, $\sigma = subst_{UG_i}$ unifizierbar ist, so ist $extend_B(f\ M_1 \dots M_r, f\ t_1 \dots t_r, n, UG) = (0, UG_{i+1}, \emptyset)$. Anderenfalls gibt die erste Ergebniskomponente mit ihrem Wert 1 das Scheitern der Unifikation an, und die dritte Komponente enthält die Clash-Menge.

□

6.2 Intelligentes Backtracking

Scheitert beim Versuch einer Regelanwendung die Unifikation des Redexes mit der linken Regelseite, so berechnet die Funktion $extend_B$ die entsprechende Clash-Menge. Das Vorgehen ist dem Backtracking bei der Auswertung von Prolog-Programmen ähnlich, aber es gibt auch wesentliche Unterschiede. Z.B. müssen beim Zurücksetzen der Berechnung über den Berechnungspunkt hinaus (*deep backtracking*) Wertzuweisungen an A-Knoten rückgängig gemacht werden.
Neben dem Scheitern einer Unifikation kann auch wegen der (erfolgreichen) Auswertung einer Bedingung eines bedingten Ausdruckes ohne Alternative zu $false$ Backtracking veranlaßt werden. Dieser besondere Fall muß in das Intelligente Backtracking-Verfahren integriert werden.

$extend_B \;:\; NF\_Exp_\Sigma(Var) \times NF\_Exp_\Sigma(Var) \times \mathbb{N} \times \mathcal{UG} \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}$

$extend_B(l_{up}, l_{down}, n, UG) :=$
let $l_{up} = (f\ M_1 \ldots M_r)$ and $l_{down} = (f\ t_1 \ldots t_r)$ and $UG = \langle C, F\rangle$ and $A = \bigcup_{i=1}^{r}(ta(M_i) \cup ta(t_i))$
and $NewExps = \{a \in A \mid a \notin Var \vee (a \in Var \wedge \nexists \langle V, E, Lab\rangle \in C$ mit $a \in V)\}$
and $NewClasses = \{\langle\{a\}, \emptyset, \emptyset\rangle \mid a \in NewExps\}$ and $C' = C \cup NewClasses$
and $NewFunDep = \{f \mid a \in NewTerms, a = c(a_1, \ldots, a_i, \ldots, a_n),$
$c_a = \langle V_a, E_a, Lab_a\rangle \in C'$ mit $a \in V_a$,
$c_{a_i} = \langle V_{a_i}, E_{a_i}, Lab_{a_i}\rangle \in C'$ mit $a_i \in V_{a_i}$,
$f : c_a \rightarrow c_{a_i}$ mit $mark_{UG}(f) = (i, a, a_i)\}$
$\cup \{f \mid a \in NewTerms, a = (g\ a_1 \ldots a_i \ldots a_n),$
$c_a = \langle V_a, E_a, Lab_a\rangle \in C'$ mit $a \in V_a$,
$c_{a_i} = \langle V_{a_i}, E_{a_i}, Lab_{a_i}\rangle \in C'$ mit $a_i \in V_{a_i}$,
$f : c_a \rightarrow c_{a_i}$ mit $mark_{UG}(f) = (i, a, a_i)\}$
and $F' = F \cup NewFunDep$ and $UG' = \langle C', F'\rangle$ and $res_0 = (0, UG', \emptyset)$
and $\forall j \in \{1, ..., r\} : res_j = unify\_exp(M_j, t_j, \emptyset, n, res_{j-1})$
in res_r

$unify\_exp \;:\; NF\_Exp_\Sigma(Var) \times T_\Sigma(Var) \times 2^{\mathbb{N}} \times \mathbb{N} \times (\mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}) \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}$

$unify\_exp(M, t, Lab, n, (ind, UG, C\text{-}Set) :=$ % Unifikation von M und t am Knoten n
if $ind = 1$ then $(ind, UG, C\text{-}Set)$ % Clash
else let $UG = \langle C, F\rangle$ and $c_M = \langle V_M, E_M, Lab_M\rangle \in C \ni M \in V_M$
and $c_t = \langle V_t, E_t, Lab_t\rangle \in C \ni t \in V_t$ and $rep_M = deref_{c_M}(M)$ and $rep_t = deref_{c_t}(t)$
in if $rep_M = rep_t$ then $(0, UG, \emptyset)$
else if $rep_t \in Var$ then $bind_B(rep_t, rep_M, Lab, n, UG)$
else if $rep_M \in Var$ then $bind_B(rep_M, rep_t, Lab, n, UG)$
else let $rep_M = c_M(M_1, \ldots, M_{r_M})$ and $rep_t = c_t(t_1, \ldots, t_{r_t})$
in if $(c_M \neq c_t)$ or $(r_M \neq r_t)$ then $(1, UG, (Lab \cup Lab_M \cup Lab_t) \setminus \{n\})$ % Clash-Menge
else let $res_0 = (0, UG, \emptyset)$ and $\forall j \in \{1, ..., r_t\}$:
$res_j = unify\_exp(M_j, t_j, Lab \cup Lab_M \cup Lab_t, n, res_{j-1})$
in res_{r_t}

$bind_B \;:\; Var \times NF\_Exp_\Sigma(Var) \times 2^{\mathbb{N}} \times \mathbb{N} \times \mathcal{UG} \longrightarrow \mathbb{N} \times \mathcal{UG} \times 2^{\mathbb{N}}$

$bind_B(rep_1, rep_2, Lab, n, UG) :=$ % $rep_1 \in Var$ wird an rep_2 gebunden
let $UG = \langle C, F\rangle$ % aus den Klassen c_1 und c_2 wird die Klasse c erstellt
and $c_1 = \langle V_1, E_1, Lab_1\rangle \in C \ni rep_1 = deref_{c_1}(v_1)\ \forall v_1 \in V_1$
and $c_2 = \langle V_2, E_2, Lab_2\rangle \in C \ni rep_2 = deref_{c_2}(v_2)\ \forall v_2 \in V_2$
and $c = \langle V_1 \cup V_2, E_1 \cup E_2 \cup \{(rep_1 \rightarrow rep_2)\}, Lab \cup Lab_1 \cup Lab_2 \cup \{n\}\rangle$ and $C' = (C \cup \{c\}) \setminus \{c_1, c_2\}$
and $F' = \{f \in F \mid f = (c_s \rightarrow c_d),\ \ c_s, c_d \notin \{c_1, c_2\},\ \ mark_{UG}(f) = mark_{UG'}(f)\}$
$\cup \{f' = (c \rightarrow c_d) \mid \exists f = (c_s \rightarrow c_d) \in F,\ \ c_s \in \{c_1, c_2\}, mark_{UG}(f) = mark_{UG'}(f')\}$
$\cup \{f' = (c_s \rightarrow c) \mid \exists f = (c_s \rightarrow c_d) \in F,\ \ c_d \in \{c_1, c_2\}, mark_{UG}(f) = mark_{UG'}(f')\}$
and $UG' = \langle C', F'\rangle$
in if UG' zirkulär then $(1, UG, cycle\_set(UG'))$ else $(0, UG', \emptyset)$

wobei
- $ta : NF\_Exp_\Sigma(Var) \longrightarrow 2^{NF\_Exp_\Sigma(Var)}$ einen Babel-Ausdruck in Normalform auf die Menge seiner Teilausdrücke abbildet,
- $NewExps$ die Menge der neu eingeführten Ausdrücke ohne bereits früher eingeführte Variablen, $NewFunDep$ die Menge der Kanten entsprechend der neuen funktionalen Abhängigkeiten ist, und
- $cycle\_set : \mathcal{UG} \longrightarrow 2^{\mathbb{N}}$ einen zirkulären Unifikationsgraphen auf die Vereinigung der Label-Mengen der Klassen eines Zyklus abbildet.

Abbildung 12: Die Funktion $extend_B$

Definition 6.3 (Babel-Backtracking) Zunächst wird Backtracking aufgrund des Scheiterns einer Unifikation definiert.
Scheitert die Unifikation zwischen einem Redex und einer linken Regelseite, so ist der aktuelle Berechnungspunkt im Berechnungsplan P_i ein offener U-Knoten $v = \langle \perp, l_{up}, \perp, Alt_v, BP_v \rangle$. Weiterhin wurde beim Versuch des Schließens des Knotens v eine Clash-Menge *Clash-Set* berechnet. Der Folgeplan P_{i+1} wird dann durch einen rückwärtsgerichteten Berechnungsschritt (Backtracking) bestimmt, welcher aus zwei Teilschritten besteht:

- Wie im logischen Fall wird die Menge *Clash-Set* der Menge BP_v als weitere Backtracking-Punkte hinzugefügt ($BP_v := BP_v \cup \mathit{Clash\text{-}Set}$).
- Das weitere Vorgehen wird nach der Menge Alt_v des Knotens v unterschieden.
  - Bei vorhandenen Alternativen für eine Regelanwendung ($Alt_v \neq \emptyset$) ist das Zurücksetzen beendet („shallow backtracking"). Die Berechnung fährt mit einem Ableitungsschritt fort.
  - Sind keine Alternativen vorhanden ($Alt_v = \emptyset$), so muß die Berechnung tiefer zurückgesetzt werden („deep backtracking"). Zu diesem Zweck wird die Menge der berechneten Backtracking-Punkte BP konsultiert und je nach Zustand weiterverfahren:
    - Bei leerer Menge BP bricht die gesamte Berechnung ab, da keine weiteren Lösungen existieren. Entweder besitzt der Plan keinen weiteren U-Knoten, der ein Scheitern eines bestimmten Unifikationsversuches verhindern könnte, oder nach Berechnung einer Lösung gibt es keine Auswahlstelle mit Alternative mehr.
    - Enthält die Menge der Backtracking-Punkte mindestens ein Element ($BP_v \neq \emptyset$), so sei m das Maximum dieser Menge (m ist der jüngste Berechnungspunkt aus BP_v). Die Berechnung wird dann sukzessive bis zum Öffnen des U-Knotens mit der Nummer m (v_m) zurückgesetzt.

      Zum einen werden alle Knoten, die beim Schließen von v_m als offene Knoten im entsprechenden Plan existierten, wieder geöffnet. Dazu sei $limit(m) := \{n \in \mathbb{N} \mid n$ Nummer eines geschlossenen U-Knotens aus P_i, $n \geq m$ und $father(n) < m$, falls existiert$\}$, wobei $father(n)$ die Nummer des Vorgänger–U-Knotens des Knotens mit der Nummer n bezeichnet. Die Struktur des Planes P_i wird durch einen Backtracking-Schritt verändert, indem alle Teilbäume, deren Wurzel eine Nummer aus $limit(m)$ besitzt, ohne die Wurzel gelöscht werden. Weiterhin werden diese Wurzeln w außer v_m (Nummer aus $limit(m) \setminus \{m\}$) neu „initialisiert" (d.h. $w = \langle \perp, l_w, \perp, alt(l_w), BP \rangle$; BP enthält nur Vorgänger–U-Knoten, falls dieser existiert) und der Knoten v_m wird geöffnet ($v_m = \langle \perp, l_m, \perp, Alt_m, BP'_m \rangle$ mit $BP'_m := BP_m \cup (BP \setminus \{m\})$.

      Zum anderen muß die Wertzuweisung an gewisse A-Knoten rückgängig gemacht werden, da durch die Propaganda der Ergebnisse A-Knoten einen nicht mehr gültigen Wert besitzen können. Dies erreicht man, indem jeder A-Knoten auf einem Pfad zwischen der Wurzel (inklusive) und einem U-Knoten aus $limit(m)$ als Wert $\perp$ zugewiesen bekommt.

      Die Berechnung von P_{i+1} wird sukzessive beim „eigentlichen Backtracking" (zweiter Punkt •) weitergeführt, wobei dann v_m aktueller Berechnungspunkt ist.

Der zweite Teil der Definition des Backtrackings befaßt sich mit bedingten Ausdrücken ohne Alternative. Neben dem Scheitern einer Unifikation veranlaßt nur noch eine zu *false* ausgewertete Bedingung eines bedingten Ausdruckes ohne Alternative das Zurücksetzen der Berechnung. In diesem Fall befindet sich die Berechnung gerade bei der Propaganda der Resultate von A-Knoten, nicht aber an einem offenen U-Knoten.
Ein in Abschnitt 5.3 durch Programmtransformation als korrekt gezeigtes Vorgehen ist es, nach der Berechnung der Bedingung zu *false* „deep backtracking" (zweiter Punkt •) durchzuführen, wobei als Menge der Backtracking-Punkte die Label-Menge des Resultates *false*, erweitert um den Vorgänger–U-Knoten für die linke Regelseite (falls dieser existiert), zu nehmen ist. □

6.3 Das IB-Schema für Babel-Programme

Das nun vorgestellte Schema des Intelligenten Backtrackings basiert auf den vorher in diesem Abschnitt definierten Berechnungsschritten. Die im Abschnitt 5 gezeigten Ausnahmen (explizite Unifikation, Funktionale) werden in diesem überblickartigen Schema nicht gesondert behandelt. In diesem Sinne ist es vereinfachend.
Die für das Intelligente Backtracking im logischen Fall gemachten Aussagen (Abschnitt 3.5) gelten analog. Abbildung 13 zeigt das Schema.

7 Laufzeitergebnisse

Dieser Abschnitt enthält Informationen über die Realisierung und Implementierung der abstrakten Maschinen SBAM und IBSBAM. Danach folgt ein abschließender Vergleich des Laufzeitverhaltens beider Maschinen.

Beide abstrakten Maschinen wurden in der Programmiersprache C unter dem Betriebssystem Unix auf SUN Workstations der Modellreihe 4 (SUN 4/65, 64 MB Ram) realisiert. Die Laufzeitmessungen wurden ebenfalls auf SUN Workstations der Modellreihe 4 gemacht.
Als Ausgangsbasis diente eine Realisierung der SBAM, welche dann zur IBSBAM modifiziert wurde. Der (in der SBAM nicht vorhandenen) Verwaltung von Label-Mengen kommt in der IB-SBAM eine besondere Bedeutung zu, da alle wichtigen Maschinenbefehle Label-Mengen modifizieren (vgl. [Koch 92]). Die Realisierung der Label-Mengen wurde in einem neuen Datentypmodul (Bezeichnung nach [Nagl 90]) zusammengefaßt. Durch die gewählte Struktur der Software kann unabhängig vom Rest des Systems die Verwaltung und Manipulationen der Label-Mengen geändert werden. So enthielt die erste Version der IB-SBAM eine sehr einfache Mengen-Verwaltung, welche schrittweise durch verbesserte Versionen ersetzt wurde.

Als Abschluß der Arbeit soll ein Vergleich des Auswertungsverhaltens von acht Babel-Programmen zwischen der IB-SBAM und der SBAM dienen (für eine Erläuterung der Babel-Programme sei wieder auf [Koch 92] verwiesen).
Das primäre Ziel des entwickelten IB-Verfahrens ist die Einschränkung des Suchraumes bei der Berechnung eines Zielausdruckes. Ein geeignetes Maß zur Bestimmung des Umfangs des Suchraumes ist die Anzahl der versuchten Regelanwendungen, d.h. die Anzahl der Versuche, einen U-Knoten im Babel-Berechnungsplan zu schließen. In der linken Hälfte der Tabelle werden gerade diese Anzahlen gegenübergestellt. Dabei ist die Einsparung des Suchraumes als $1 - \frac{IB-SBAM}{SBAM}$ berechnet.

Die rechte Hälfte der Tabelle vergleicht die Auswertungszeiten und berechnet die Beschleunigung ebenfalls als $1 - \frac{IB-SBAM}{SBAM}$ (d.h. negative Beschleunigung bedeutet Verlangsamung).

	Anzahl Berechnungspunkte			Zeit/sec		
	SBAM	IB-SBAM	Einsp.	SBAM	IB-SBAM	Beschl.
„nand“	759.698	29.513	96 %	57,960	6,617	89 %
„cubes“	2.482.913	130.960	95 %	63,403	5,497	91 %
„db_query“	1.194	417	65 %	0,040	0,020	50 %
„move“	218	70	68 %	0,030	0,013	57 %
„map_fav“	286	286	0 %	0,010	0,013	-30 %
„map_unf“	6.208	788	87 %	0,217	0,057	73 %
„queen_fav“	6.541	6.541	0 %	0,697	2,077	-198 %
„queen_unf“	10.429	5.739	44 %	1,107	1,880	-70 %

Tabelle 1: Messungen der SBAM und IB-SBAM im Vergleich

$ib\_scheme_B :=$
let P_0 „initialer Berechnungsplan“ and UG_0 „initialer Unifikationsgraph“
and $state_0 = \langle P_0, UG_0 \rangle$
and $\forall j \in \mathbb{N} : state_{j+1} =$ if $next\_state_B(state_j) = (\langle P', UG' \rangle, yes)$ then $\langle P', UG' \rangle$
in $\langle P_0, UG_0 \rangle, \langle P_1, UG_1 \rangle, \ldots$

$next\_state_B \ : \ \langle \mathcal{PL} \times \mathcal{UG} \rangle \longrightarrow \langle \mathcal{PL} \times \mathcal{UG} \rangle \times \{yes, no\}$

$next\_state_B(\langle P, UG \rangle) :=$
if („Wurzel von P der Form $\langle M, \perp \rangle$“)
then let „v erstes Blatt (bzgl. depth-first left-to-right), welches entweder
ein offener U-Knoten oder ein A-Knoten ohne zugewiesenen Wert ist“
in if $v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ % U-Knoten
then let „v soll als n-ter Knoten mit l_{head} (Regel m) geschlossen werden“
and $extend(l_{up}, l_{head}, n, UG) = (ind, UG', Clash\text{-}Set)$
in if $ind = 0$ % kein Clash
then let „P' entsteht aus P durch Schließen von v und
Einführung des Sohnes für rechte Regelseite (A-Knoten)“
in $(\langle P', UG' \rangle, yes)$
else let „P' entsteht aus P durch Änderung von v zu
$v = \langle \perp, l_{up}, \perp, Alt \setminus \{m\}, BP \cup Clash\text{-}Set \rangle$“
in $deep\_back_B(\langle P', UG \rangle)$
else let $v = \langle M, \perp \rangle$ % A-Knoten
in if $M \in NF\_Exp_\Sigma(Var)$
then let „P' entsteht aus P durch Wertzuweisung am Knoten v
und Propaganda des Wertes Richtung Wurzel“
and „UG' geht aus UG hervor, erweitert um in P' zugewiesene
Werte und erweiterte Label-Mengen durch Propaganda
über U-Knoten hinweg.“
in $(\langle P', UG' \rangle, yes)$
else let „P' entsteht aus P durch Einführung von Söhnen
(A-Knoten) entsprechend der Struktur von M“
in $(\langle P', UG \rangle, yes)$
else % Wurzel hat Wert zugewiesen bekommen; evtl. Lösung ausgeben
let „$v = \langle n, l_{up}, l_{down}, Alt, BP \rangle$ (geschl.) U-Knoten mit maximaler Nummer“
and „P' entsteht aus P durch Öffnung von v zu $v := \langle \perp, l_{up}, \perp, Alt, BP \rangle$“
and % Simulation von Naivem Backtracking
„P'' entsteht aus P', indem die Menge BP eines jeden U-Knotens mit der
maximalen Nummer aller jüngeren U-Knoten (depth-first left-to-rigth) mit
nichtleeren Alt-Mengen initialisiert wird, falls solch eine existiert
(sonst mit der leeren Menge)“
in $deep\_back_B(\langle P'', UG' \rangle)$

$deep\_back_B \ : \ \langle \mathcal{PL} \times \mathcal{UG} \rangle \longrightarrow \langle \mathcal{PL} \times \mathcal{UG} \rangle \times \{yes, no\}$

$deep\_back_B(\langle P, UG \rangle) :=$
let „$v = \langle \perp, l_{up}, \perp, Alt, BP \rangle$ erster (bzgl. depth-first left-to-right) offener U-Knoten“
in if $Alt \neq \emptyset$ then $(\langle P, UG \rangle, yes)$ % Backtracking beendet
else if $BP = \emptyset$ then $(\langle P, UG \rangle, no)$ % keine weitere Lösung vorhanden
else let „Berechnung bis zum Knoten v_m mit Nummer $m = \max(BP)$
zurücksetzten (Teilbäume des Planes löschen, Propaganda rückgängig
machen), und BP_m des Knotens v_m um $(BP \setminus \{m\})$ erweitern;
ergibt $\langle P', UG' \rangle$“
in $deep\_back_B(\langle P', UG' \rangle)$

Abbildung 13: Das IB-Schema für Babel

Das primäre Ziel dieser Arbeit wurde erreicht: Das entwickelte IB-Verfahren für die Auswertung funktional-logischer Babel-Programme schränkt den Suchraum der Zielberechnung deutlich ein (bis zu 96 %). Die untersuchten Programme entstammen alle der Literatur und wurden nur (von der Prolog-Syntax) in die Babel-Syntax überführt. Es handelt sich also nicht um speziell für den Vergleich angefertigte Programme.
Natürlich bestand auch als sekundäres Ziel die Verkürzung der Auswertungszeit (durch Einschränkung des Suchraumes). Dieses Ziel konnte mit dem vorgestellten IB-Verfahren (bisher) nur teilweise erreicht werden. Für die Änderung der Auswertungszeit ist das Verhältnis zwischen eingespartem Suchraum und zusätzlichem Aufwand des IB-Verfahrens wesentlich, wobei die Verwaltung der Label-Mengen den größten Teil des zusätzlichen Aufwands verursacht. Z.B. verursacht Intelligentes Backtracking bei der Auswertung des Programms „map_fav“ ohne Einschränkung des Suchraumes eine Verlängerung der Auswertungszeit um 30 %, wohingegen bei „queen_unf“ trotz einer Einsparung des Suchraumes um 44 % die Auswertungszeit um 70 % zunahm. Der Grund dieser auf den ersten Blick verwunderlichen Tatsache liegt in der Größe der verwalteten Label-Mengen, die im zweiten Fall viel umfangreicher sind. Aus dem gleichen Grund kommt es bei der Auswertung des Programms „queen_fav“ zu einer solch großen Erhöhung der Auswertungszeit um 198 %.

In dieser Arbeit lag das Hauptinteresse in der Demonstration der Machbarkeit des Ansatzes und in der Verfolgung des primären Ziels. Das vorgestellte Verfahren bietet hinsichtlich der Größe der verwalteten Mengen weitere Möglichkeiten der Verbesserung, ohne an Genauigkeit zu verlieren. Weiterhin können raffinierte Implementierungstechniken (z.B. Bit-Vektoren) den zusätzlichen Aufwand verringern. Zusammenfassend kann man eine deutliche Einschränkung des Suchraumes bei der Auswertung von Babel-Programmen durch das entwickelte IB-Verfahren feststellen, welche (in den untersuchten Fällen) meistens auch eine Verkürzung der Auswertungszeit bewirkt. Bei deterministischen Berechnungen oder bei der Verwaltung großer Mengen kann es zu einer Verlangsamung der Auswertung kommen. Eine weitere Verfolgung des Ansatzes „Intelligentes Backtracking“ scheint interesasant, z.B. die Integration dieses Verfahrens mit weiteren Optimierungen.

Literatur

[Aït-Kaci 91] Hassan Aït-Kaci : „Warren's Abstract Machine: A Tutoriel Reconstruction“, MIT Press 1991.

[Bosco et al. 89a] P. G. Bosco, C. Cecchi, E. Giovannetti, C. Moiso, C. Palamidessi : „Using Resolution for a Sound and Efficient Integration of Logic and Functional Programming“, in : J. de Bakker (ed.), „Languages for parallel architectures: Design, Semantics, Implementation Models“, Wiley 1989.

[Bosco et al. 89b] P. G. Bosco, C. Cecchi, C. Moiso : „An extension of WAM for K-LEAF : A WAM-based compilation of conditional narrowing“, Proceedings of the International Conference on Logic Programming (ICLP) 1989, MIT Press.

[Bruynooghe, Pereira 84] Maurice Bruynooghe, Luís M. Pereira : „Deduction Revision by Intelligent Backtracking“, in: J. A. Campbell (ed.), „Implementations of Prolog“, Ellis Horwood Limited 1984.

[Chang, Despain 85] Jung-Herng Chang, Alvin M. Despain : „Semi-Intelligent Backtracking of Prolog based on Static Data Dependency Analysis“, Proceedings of the Symposium on Logic Programming (SLP) 1985, IEEE Comp. Soc. Press.

[Clocksin, Mellish 87] W. F. Clocksin, C. S. Mellish : „Programming in Prolog“, Springer-Verlag 1987.

[Codognet et al. 88] Christian Codognet, Philippe Codognet, Gilberto Filè : „Yet another Intelligent Backtracking Method“, Proceedings of the International Conference on Logic Programming (ICLP) / Symposium on Logic Programming (SLP) 1988, MIT Press.

[Codognet 89a] Christian Codognet : „Backtracking Intelligent en Programmation Logique: Un cadre général“, Dissertation (franz.), Universität Paris, 1989.

[Codognet 89b] Philippe Codognet : „Backtracking Intelligent en Programmation Logique: De la théorie à l'implémentation et à l'application au parallélisme“, Dissertation (franz.), Universität Bordeaux, 1989.

[Codognet, Sola 91] Philippe Codognet, Thierry Sola : „Extending the WAM for Intelligent Backtracking“, Proccedings of the International Conference on Logic Programming (ICLP) 1991, MIT Press.

[Cox 84] P. T. Cox : „Finding Backtrack Points for Intelligent Backtracking“, in: J. A. Campbell (ed.), „Implementations of Prolog“, Ellis Horwood Limited 1984.

[Cox 87] P. T. Cox : „On determing the causes on nonunifiability“, Journal of Logic Programming, 1987, Number 4.

[Koch 92] Patrik Koch : „Intelligentes Backtracking bei der Auswertung funktional-logischer Programme“, Diplomarbeit, RWTH Aachen, 1992.

[Kumar, Lin 87] Vipin Kumar, Yow-Jian Lin : „An Intelligent Backtracking Scheme for Prolog“, Proceedings of the Symposium on Logic Programming (SLP) 1987, IEEE Comp. Soc. Press.

[Kumar, Lin 88] Vipin Kumar, Yow-Jian Lin : „A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog“, Journal of Logic Programming, 1988, Number 5.

[Loogen 91] Rita Loogen : „From Reduction Machines to Narrowing Machines“, Colloquium on Combining Paradigms for Software Development – CCPSD, TAPSOFT 1991, LNCS 494, Springer-Verlag 1991.

[Malhotra 90] Vishv M. Malhotra : „An Algorithm for Optimal Back-Striding in Prolog“, Proceedings of the International Conference on Logic Programming (ICLP) 1990, MIT Press.

[Matwin, Pietrzykowski 82] Stanislaw Matwin, Tomasz Pietrzykowski : „Exponential Improvements of Efficient Backtracking: Data Structures and Implementation“, Proceedings of the Conference on Automated Deduction 1982, LNCS 138, Springer-Verlag.

[Matwin, Pietrzykowski 85] Stanislaw Matwin, Tomasz Pietrzykowski : „Intelligent Backtracking in Plan-Based Deduction“, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 1985, Number 6.

[Moreno-Navarro et al. 90] Juan J. Moreno-Navarro, Herbert Kuchen, Rita Loogen, Mario Rodríguez-Artalejo : „Lazy Narrowing in a Graph Machine“, Proceedings of the Conference on Algebraic and Logic Programming 1990, LNCS 463, Springer-Verlag.

[Moreno-Navarro, Rodríguez-Artalejo 89] Juan J. Moreno-Navarro, Mario Rodríguez-Artalejo : „Logic Programming with Functions and Predicates: The Language Babel“, Technical Report DIA/89/3, Universidad Complutense, Madrid, 1989.

[Nagl 90] Manfred Nagl : „Softwaretechnik: Methodisches Programmieren im Großen“, Springer-Verlag 1990.

[Pereira, Porto 80] Luís M. Pereira, António Porto : „Selective Backtracking for Logic Programms“, Proceedings of the Conference on Automated Deduction 1980, LNCS 87, Springer-Verlag.

[Pietrzykowski, Matwin 82] Tomasz Pietrzykowski, Stanislaw Matwin : „Exponential Improvements of Efficient Backtracking: A Strategy for Plan-Based Deduction“, Proceedings of the Conference on Automated Deduction 1982, LNCS 138, Springer-Verlag.

[Schöning 89] U. Schöning : „Logik für Informatiker“, BI Wissenschaftsverlag 1989.

[Winkler 91] Stephan Winkler : „Entwicklung eines Programmsystems für die funktionale logische Sprache Babel durch Integration einer Programmierumgebung, eines Compilers und einer stackbasierten abstrakten Maschine“, Diplomarbeit, RWTH Aachen, 1991.

[Wolfram 86] David A. Wolfram : „Intractabel Unifiability Problems and Backtracking“, Proceedings of the International Conference on Logic Programming (ICLP) 1986, LNCS 225, Springer-Verlag.