Formale Beschreibung von DATR

Diplomarbeit
an der Technischen Fakultät der Universität Bielefeld
Dezember 1997

vorgelegt von

Thorsten Gerdsmeier

Betreuer:

Prof. Dr. Peter B. Ladkin	Prof. Dr. Dafydd Gibbon
Universität Bielefeld	Universität Bielefeld
Technische Fakultät	Fakultät für Linguistik
Gruppe RVS	und Literaturwissenschaft
Universitätsstraße 25	Universitätsstraße 25
33615 Bielefeld	33615 Bielefeld

Thorsten Gerdsmeier, Borriesstraße 16, 33719 Bielefeld, 29.Dezember 1997

Inhaltsverzeichnis

1 Einführung **4**

2 Semantische Netze und DATR - die Pragmatik **9**
2.1 Semantische Netze 9
2.2 Vererbung in DATR 12

3 Die Syntax von DATR **16**
3.1 Konkrete und abstrakte Syntax 16
3.2 Die konkrete Syntax von DATR 17
3.3 Die abstrakte Syntax von DATR 18
3.4 Beziehung zwischen abstrakter und konkreter Syntax in DATR 19
3.5 Variablen in DATR 21
3.6 Mechanismen zur Variablenbehandlung 22

4 Die Semantik von DATR **24**
4.1 Motivation zur Entwicklung einer Semantik 24
4.2 Interpreter-Semantik und Denotationale Semantik 25
4.2.1 Die Interpreter Semantik 25
4.2.2 Die Denotationale Semantik 25
4.3 Ergänzung von Kellers Semantik 34
4.4 Natural Semantics 37
4.4.1 Begriffe der Natural Semantics 37
4.4.2 Die Natural Semantics von DATR 37
4.4.3 Die Regeln 38
4.4.4 Funktionen 49
4.5 Beispiel für eine Ableitung 50
4.6 Der Theorembeweiser Isabelle zur Ausführung von DATR 54

4.6.1 Nutzung von Isabelle zur Ausführung von DATR . . . 54
4.6.2 Das HTML User Interface zu DATR für Isabelle 56
4.6.3 Einschränkungen 57

Kapitel 1

Einführung

Die vorliegende Arbeit behandelt die formale Beschreibung von DATR und besteht aus drei Teilen : der erste Teil beschäftigt sich mit der Pragmatik von DATR, der zweite Teil beschreibt die Syntax von DATR und der dritte Teil setzt sich mit der Semantik von DATR auseinander.
Im ersten Teil werden Beispiele zur Pragmatik von DATR vorgestellt. Unter der Pragmatik wird der Nutzen und der Gebrauch der Sprache verstanden. Das impliziert mögliche Anwendungsgebiete der Sprache. DATR wurde konzipiert, um lexikalisches Wissen darzustellen. Die im ersten Teil gewählten Beispiele von DATR Programmen sind jedoch nicht aus dem Gebiet der Linguistik gewählt worden. Damit soll die Vielfältigkeit von DATR illustriert werden.
Nachdem im ersten Teil ein kurzer Eindruck von der Sprache DATR entstanden ist, wird im zweiten Teil die konkrete und eine abstrakte Syntax von DATR vorgestellt. Es bestehen schon einige Arbeiten zur Beschreibung der konkreten Syntax von DATR [16] [8] . Die Beschreibungen umfassen nicht immer den gesamten Umfang von DATR, wie er in den DATR-Papers von Gazdar und Evans beschrieben wird [6]. Diese DATR Papers dienen als Grundlage für die vorliegende Arbeit. Die explizite Unterscheidung und die Darstellung des Zusammenhangs zwischen abstrakter und konkreter Syntax ist ein wichtiger Schritt zur Definition einer Semantik.
Der dritte Teil beschäftigt sich mit der Beschreibung der Semantik von DATR. Dieses Kapitel ist das Kernstück dieser Arbeit. Bisher gibt es noch nicht viele Arbeiten zu diesem Thema. Bekannt ist die Semantikbeschreibung für DATR von Evans und Gazdar [9]. Hier wird versucht, die Semantik von DATR durch einen Automaten darzustellen. Es ist jedoch nicht möglich diesen Ansatz auf den gesamten Umfang der Sprache auszudehnen. Eine weitere Beschreibung von DATR wird in Form von Inferenzregeln gemacht.
Bill Keller hat eine Denotationale Semantik für DATR entworfen [21] .

Die Semantik von DATR wurde auch durch Veröffentlichung eines PROLOG Quellcodes dargestellt [17]. Es ist festzustellen, daß die maschinenunabhängigen Semantiken nicht konsequent formal dargestellt wurden.
In dieser Arbeit wurde besonderer Wert darauf gelegt, die Formulierung einer Semantik an den Zielen zu messen, die mit ihr verbunden werden.
Die Benutzer einer Programmiersprache ohne eine formale Semantik haben eine intuitive Idee, wie sich ihre Programme verhalten. Die Akzeptanz einer Semantik für eine Programmiersprache hängt davon ab, ob die Benutzer dieser Sprache davon überzeugt werden können, daß die Semantik das erwartete Verhalten ihrer Programme beschreibt. Die Plausibilität einer Semantik kann am besten überprüft werden, wenn ein Interpreter für die Semantik zur Verfügung steht.
Es ist beobachtbar, daß sich verschiedene Semantikformalismen unterschiedlich gut für die Beschreibung einer Programmiersprache eignen. Ein Semantikformalismus kann zum Beispiel zur Beschreibung einer funktionalen Sprache eine gute Wahl sein, sich jedoch für die Beschreibung einer imperativen Sprache nicht gut eignen. Dieses hängt auch davon ab, wie stark sich die Struktur des Semantikformalismus und der zu beschreibenden Sprache ähneln. Wir betrachten eine Semantik als gut geeignet für die Beschreibung einer Programmiersprache, wenn sich diese Programmiersprache in dem Formalismus kompakt darstellen läßt. Die Semantik muß sich weiterhin entweder gut als ein Handbuch für die Programmiersprache eignen, oder aber die Möglichkeit bieten, aus ihr einen effizienten, beweisbar richtigen Compiler ableiten zu können. Wenn es einen Interpreter für die Semantik gibt, so ist dieses ein weiterer Vorteil.
In dieser Arbeit wurde zunächst die Denotationale Semantik betrachtet. Da Keller [21] eine Denotationale Semantik für DATR formuliert hat, wurde diese auf die Eignung für DATR geprüft. Zur Zeit ist kein Verfahren bekannt, um aus einer Denotationalen Semantik einen beweisbar richtigen, effizienten Compiler zu entwickeln. Deshalb sollte diese Semantik in erster Linie als ein Handbuch für DATR betrachtet werden. Es stellte sich heraus, daß einige Punkte in Kellers Semantik nicht explizit formuliert wurden. Deshalb wurde in einer weiteren Denotationalen Semantik versucht, diese Punkte explizit darzustellen.
Mit der Übertragung einer Denotationalen Semantik in eine funktionale Sprache kann im Prinzip ein Interpreter entwickelt werden. Die Denotationale Semantik von Keller erwies sich als zu abstrakt, um sie direkt in eine funktionale Sprache zu übertragen.
Nach der Untersuchung der Denotationalen Semantik wurde DATR in Natural Semantics betrachtet. Die Natural Semantics eignet sich für ein Handbuch gut. Die verwendete Logik ist einfach und schnell zu verstehen. DA-

TR läßt sich kompakt in diesem Formalismus darstellen. Einzelne Ableitungsschritte von DATR können direkt durch einzelne Regeln der Natural Semantics dargestellt werden. Als subjektiver Vorteil gilt die Komfortabilität der Beschreibung. Es ist möglich, die Beschreibung mit einigen wenigen Regeln zu beginnen, die Schritt für Schritt um weitere Regeln ergänzt werden können. Die schrittweise Vergrößerung der Regelmenge läßt ein einfaches Testen der Regeln zu. Es bleibt zu Untersuchen, ob sich bekannte Software-Entwicklungsmethoden wie zum Beispiel Rapid-Prototyping auf die Entwicklung einer Semantik in Natural Semantics übertragen lassen.
Es kann ein effizienter, beweisbar richtiger Compiler aus Natural Semantics entwickelt werden. Das Abstraktionsniveau einer solchen Semantik ist hoch genug, um eine Vielzahl von Implementationen zu ermöglichen. Die Übertragung der Natural Semantics in einen Formalismus wie zum Beispiel TLA+ ist problemlos möglich.
Damit kann die Natural Semantics als eine Spezifikation dienen, aus der mit Hilfe von hierachischen Methoden, wie TLA+, eine Implementation abgeleitet werden kann. Eine Ableitung ist nicht Bestandteil dieser Diplomarbeit.
Der wichtige Vorteil der Entwicklung eines beweisbar richtigen Compilers aus einer Natural Semantics liegt in der Möglichkeit, die Semantik in einen Theorembeweiser übertragen zu können. Damit wird ein großer Schritt in Prüfung der Plausibilität der Semantik gemacht. Die zusätzliche Nutzung der maschinenunabhängigen Regeln als ein Programm für einen Interpreter, dem Theorembeweiser, erhöhen das Vertrauen in die Semantik und letzendlich in einen Compiler, der diese Regeln als Grundlage nutzt. Die Ausführung der Regeln in einem Theorembeweiser ist also als zusätzlicher, wichtiger Kontrollmechanismus zu betrachten.
Zusammenfassend ergibt sich ein möglicher Weg zur Entwicklung eines effizienten beweisbar richtigen Compilers :

- Beschreibung einer Programmiersprache durch Regeln in Natural Semantics.

- Übertragung der Regeln in einen Theorembeweiser. Damit wird ein Interpreter erhalten, mit dem Programme der beschriebenen Programmiersprache ausgeführt werden können.

- Übertragung der Regeln der Natural Semantics nach TLA+. Damit wird wird eine Spezifikation in TLA+ erhalten. Diese dient als Ausgangspunkt für die Entwicklung eines Compilers.

- Schrittweise Verfeinerung der TLA+ Spezifikation durch weitere TLA+ Module bis eine konkrete Implementation erhalten wird.

- Direkte Übertragung der Implementation, die durch TLA+ Module beschrieben wird, in eine Programmiersprache

In dieser Arbeit hat die Ausführung einiger DATR Testprogramme mit Hilfe des Theorembeweisers keinen Widerspruch zu dem Verhalten von anderen DATR Implementationen gezeigt. Damit können die in dieser Arbeit aufgestellten Regeln von DATR in Natural Semantics als sehr plausibel angesehen werden.
Es wurde auch *Action Semantics* als Formalismus zur Beschreibung von DATR untersucht. *Action Semantics* ist von Peter Mosses und David A. Watt entwickelt worden, um Programmiersprachen und andere formale Sprachen, wie z.B Abfragesprachen für Datenbanken, zu spezifizieren. Die Semantik einer Sprache wird in diesem Formalismus in Form von *actions* beschrieben.
Jede syntaktische Phrase der beschriebenen formalen Sprache erhält durch die Ausführung einer *action* eine Bedeutung. Es existieren Tools, die aus einer Beschreibung in Action Semantics einen Compiler für eine Sprache generieren. Es ist auch möglich, die Syntax und Plausibilität von Regeln in Action Semantics durch einen Interpreter zu überprüfen.
Es wurde eine Beschreibung von DATR in Action Semantics angefertigt. Diese Beschreibung ist nicht Bestandteil der Diplomarbeit, weil es nicht möglich war, ihre Plausibilität durch eine Implementation nachzuweisen.
Die Richtigkeit einer Action Semantics Description kann mit den ASD Tools [5] überprüft werden. Mit diesen Tools kann eine Action Semantics Description, die in LaTeXgeschrieben wurde, in eine ASCII Repräsentation umgewandelt werden. Weiterhin wird die Beschreibung auf Typkorrektheit und syntaktische Korrektheit überprüft. Eine solche ASCII Repräsentation kann als Eingabe in den *actioneer generator* dienen [33].
Die ASD Tools benutzen das ASF+SDF System. Dieses System basiert auf dem Centaur System. Es war nicht möglich das Centaur System zu installieren,weil dafür eine Lizenz benötigt wird. Deshalb war eine automatische Syntax- und Typüberprüfung der Action Semantics Beschreibung von DATR nicht möglich. Da diese Tools als automatische Aufbereitung der DATR Semantik für die Eingabe in den *actioneer generator* fehlten, war es nicht möglich, ein DATR Programm auf dem Interpreter auszuführen.
Die Vorgehensweise zur Ausführung eines Programms mit dem Action Notation Interpreter ist in der Abbildung 1.1 skizziert : Mit Hilfe von ml-lex und ml-yacc wird ein Source Code in einen abstrakten Syntaxbaum übersetzt. Der *actioneer generator* hat als Eingabe eine Semantik und generiert ein Programm *actioneer* für die Sprache, dessen Semantik die Eingabe war. Der *actioneer* hat wiederum als Eingabe einen abstrakten Syntaxbaum eines Programms und generiert actions, die von dem Action Notation Interpreter

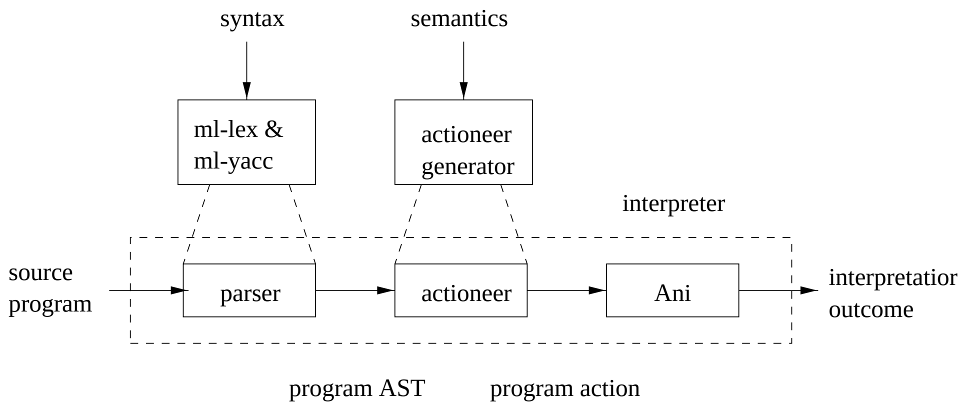

Abbildung 1.1: Der Action Semantics Interpreter zur Ausführung von DATR

ANI ausgeführt werden können. Es gibt auch Tools, die aus einer *Action Semantics-Beschreibung* Compiler generieren. Erfahrungen mit Compilergeneratoren für Action Semantics haben gezeigt, daß die erhaltenen Compiler Code erzeugen, der um Zehnerpotenzen langsamer als Code von handgeschriebenen Compilern ist.

Eine *Action Semantics* von DATR umfaßt etwa 30 - 40 Seiten. DATR läßt sich in diesem Formalismus also nicht sehr kompakt darstellen.

Ein Weg,um aus einer Beschreibung in Action Semantics einen beweisbar richtigen Compiler zu entwickeln, ist die Übertragung der *actions* der Action Semantics nach TLA+. Aus der so erhaltenen Zustandsmaschine kann eine Implementation entwickelt werden. Eine Übertragung der Zustandsmaschine in den Theorembeweiser Isabelle kann hier gewichtige Indizien für die Richtigkeit der Regeln liefern.

Wenn die Action Semantics in einen Theorembeweiser übertragen werden soll, so wird es praktisch sein, wenn die Action Semantics als Natural Semantics in TLA formuliert wurde.

Der Umfang der Action Semantics und die Vielzahl der Details, die in dieser Semantik festgelegt werden, wie zum Beispiel ein sehr konkretes Speichermodell, lassen vermuten, daß die aus einer Action Semantics entwickelten Implementationen immer langsamer sind, als Implementationen, die direkt aus einer Natural Semantics entwickelt wurden.

Kapitel 2

Semantische Netze und DATR - die Pragmatik

2.1 Semantische Netze

Die Programmiersprache DATR wurde 1989 von Roger Evans und Gerald Gazdar entwickelt. Ein DATR Programm definiert ein Semantisches Netz zur Repräsentation von Wissen. Die Sprache wurde speziell für den Zweck entwickelt um lexikalisches Wissens darzustellen, insbesondere sollte die Darstellung regulärer und irregulärer Lexeme effizient möglich sein. Teile von Arabisch und vielen europäischen Sprachen wurden in DATR dargestellt. Es hat sich später herausgestellt, daß DATR Turing komplett ist und auch ein praktischer Formalismus für vielfältige andere Anwendungen ist. Zu den neueren Anwendungen gehören D. Gibbons Programmierung des *bubble sort* Algorithmus in DATR [15] und die Darstellung von P. Ladkins *WB-Graph Modell* in DATR [12] [11].
Die Entwicklung von Formalismen zur lexikalischen Wissensrepräsentation wurde von Semantischen Netzen aus der Künstlichen Intelligenz beeinflußt. Bei der Entwicklung von DATR wurde die Forderung aufgestellt, daß die Sprache eine explizite Theorie der Vererbung haben soll. Diese Forderung ist im Zusammenhang mit den Problemen bei einigen Semantischen Netzen in der Künstlichen Intelligenz zu verstehen. Semantische Netze sind in der KI seit etwa 1968 bekannt. Ein Semantisches Netz besteht aus Knoten. In Knoten werden Objekte, Ereignisse oder Sachverhalte gespeichert. Beziehungen zwischen Knoten werden durch Kanten beschrieben. Ein Beispiel eines Semantischen Netzes ist FRL, die *Frame Representation Language*. In FRL heißen die Knoten des Netzes *Frames*. Der Begriff Frame wurde 1975 von Marvin Minsky geprägt. Ein FRAME besteht aus einer Menge von *slots*. In

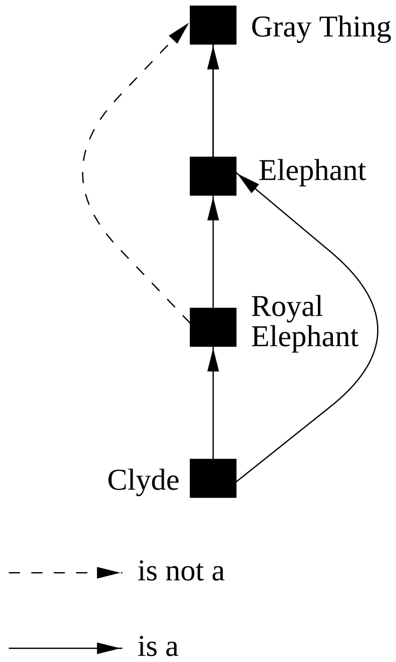

Abbildung 2.1: Ein Semanisches Netz

einem slot können Eigenschaften eines Objektes gespeichert werden. Jeder Frame besitzt einen besonderen slot, den AKO slot. AKO steht für *a kind of*. In einem AKO slot können beliebig viele Links zu anderen Frames stehen. Ein Frame erbt die Eigenschaften aller Frames, zu denen er einen AKO Link hat. Damit kann ein Frame von mehreren anderen Frames Eigenschaften erben. Mehrfachererbung in Semantischen Netzen kann problematisch sein, wie folgendes Beispiel von Touretzky [43] zeigt: In dem Netz Abbildung 2.1 werden folgende Aussagen gemacht:

- elephants are gray
- royal elephants are elephants
- royal elephants are not gray
- Clyde is a royal elephant
- Clyde is an elephant

Eine Anfrage an das Netz, ob Clyde grau ist, ergibt keine befriedigende Antwort. Clyde ist ein royal elephant und royal elephants sind weiß , andererseits ist Clyde auch ein Elefant und Elefanten sind grau. In FRL bekommt man als Ergebnis einer solchen Anfrage: Clyde is grey und Clyde is not grey, weil beide Aussagen aus dem Netz geschlossen werden können. Bei den frühen KI Sprachen hing das Ergebnis häufig von dem Algorithmus ab, der die Inferenzen ableitet. Dieser kann bei verschiedenen Implementationen verschieden sein. Häufig wird die *shortest path inference* benutzt. Bei dieser Methode wird als Ergebnis der Wert geliefert, der durch das Folgen der wenigsten Links erhalten werden kann. In diesem Beispiel wäre jedoch auch das problematisch, da beide Ergebnisse durch das Folgen der gleichen Anzahl von Links erhalten werden können.

Um vorhersagbare Ergebnisse zu erzielen, wurde bei der Entwicklung von DATR die Forderung nach einer expliziten Theorie der Vererbung gestellt. Die Semantischen Netze, die in DATR definiert werden können, sollen orthogonal sein. Das bedeutet Mehrfachererbung soll möglich sein, die ererbten Attribute sollen jedoch immer disjunkt sein. Die Problematik der Vererbung von Eigenschaften in Semantischen Netzen zur Wissensrepräsentation liegt in dem Wunsch allgemeine Eigenschaften vererben zu können, andererseits sollen jedoch Objekte auch individuelle Eigenschaften haben, die sie von anderen Objekten unterscheiden. Bei der Entwicklung von DATR wurde weiterhin die Forderung gestellt, daß der Formalismus eine klare Semantik hat. Eine klare Semantik fehlt vielen Semantischen Netzen der frühen KI. Knoten und Kanten ist keine genaue Bedeutung zugeordnet. Gleichartiges Wissen kann sowohl in Kanten wie auch in Knoten gespeichert werden. Die Arbeiten von Woods [47] ‚What's in a Link ‘und Brachman [1] ‚What's in a Concept ‘trugen zur Klärung des Problems bei.

In DATR gibt es nur einen Kantentyp, die ‚Is-A ‘- Kante. Damit kann zu einer Klasse von Objekten eine Unterklasse definiert werden. Die Intention bei der Entwicklung von DATR war nicht eine Sprache zu schaffen, die so umfangreich und vielseitig ist, wie die bekannten KI Sprachen. Es sollte vielmehr eine Sprache für den speziellen Einsatz in der lexikalischen Wissensrepräsentation geschaffen werden. Aber auch in diesem Bereich ist die Definition von individuellen Eigenschaften und allgemeinen Eigenschaften wichtig.

2.2 Vererbung in DATR

Zunächst wird Vererbung in DATR an einem einfachen Beispiel diskutiert. Das Beispiel ist eine verkürzte Fassung des DATR Programms zum Cali-Accident WB-Graph Modell [12] [11] .

```
E1:
 <event>  == 'aircraft impacts mountain'
 <why>    == S11: <state>  S11: <why>  E12: <event>.
S11:
 <state>  == 'GWPS monoeuvre failed'
 <why>    == E111: <event>  E111: <why>
             S112: <state>  E113: <event>.
E12:
 <event>  == 'AC in mountainous terrain'.
E111:
 <event>  == 'GWPS manoeuvre initiated'
 <why>    == E1111: <event>.
E1111:
 <event>  == 'GWPS warning sounds'.
S112:
 <state>  == 'AC did not exhibit optimal climb performance'.
E113:
 <event>  == 'AC very close to mountains'.
S121:
 <state>  == 'AC on wrong course/position (2D-planar)'.
S112:
 <state>  == 'AC did not exhibit optimal climb performance'.
```

Ein DATR Programm definiert ein Semantisches Netz. Semantische Netze können durch Graphen dargestellt werden. In DATR ist der Graph durch eine Menge von Gleichungen definiert. Auf der linken Seite einer Gleichung steht immer ein Knoten und ein Pfad. Beispiele für Pfade in dem obigen DATR Programm sind `<event>` und `<state>`. Auf der rechten Seite einer Gleichung können Atome und Verweise stehen. Die Verweise von `S11:<why>` sind zum Beispiel `E111:<event>`, `E111: <why>`, `S112: <state>` und `E113: <event>`. In diesem Programm sind alle Atome auf der rechten Seite einer Gleichung in Hochkommata eingeschlossene Zeichenketten. `'aircraft impacts mountain'` ist ein Beispiel für ein Atom.
Mehrere Gleichungen können unter einem Knoten zusammengefaßt werden. Im folgenden sind zwei Gleichungen dem Knoten `S11` zugeordnet:

```
S11:
 <state>  == 'GWPS monoeuvre failed'
 <why>    == E111: <event>  E111: <why>
             S112: <state>  E113: <event>.
```

Die Bedeutung ist gleich der von

```
S11:<state>  == 'GWPS monoeuvre failed'
S11:<why>    == E111: <event>  E111: <why>
                S112: <state>  E113: <event>.
```

An ein DATR Programm können Anfragen gestellt werden. Die Auswertung der Anfrage kann man durch eine Grafik veranschaulichen. In der Abbildung 2.2 wird die Auswertung der Anfrage `E1:<why>` dargestellt. Das Ergebnis einer Anfrage ist immer eine Sequenz von Atomen. In der Baumdarstellung sind die Blätter des Baumes das Ergebnis der Anfrage. Das Ergebnis einer Anfrage kann nicht aus Verweisen bestehen, da dann die Anfrage nicht vollständig ausgewertet wäre.
In dem Beispiel erbt `E1:<why>` also Werte von `S11:<state>,S11:<why>` und `E12:<event>`.
Das Beispiel des Cali Unfalls zeigt den einfachsten Fall von Vererbung in DATR. Als nächstes wird ein Beispiel für Vererbung mit defaults in DATR gegeben. Als Beispiel dient noch einmal Clyde, the royal elephant. In DATR wird zu einer Anfrage immer der längste Präfix des Anfragepfades ausgewertet, der in dem DATR Programm definiert ist. `<>` bedeutet den leeren Pfad. Der leere Pfad ist ein Präfix jeder Anfrage.
Die Anfrage `Clyde:<color>` wird folgendermaßen ausgewertet. Der leere Pfad ist der längste Präfix, der zu der Anfrage paßt. Die weitere Auswertung der Anfrage muß berechnet werden. Dazu wird der Pfad auf der linken Seite der passenden Gleichung (der nach Definition ein Präfix des Anfragepfades sein muß) vom Anfragepfad abgeschnitten. Der verbleibende Rest heißt $default$ Pfad. Der $default$ Pfad des Anfragepfades wird an alle Pfade der Verweise auf der rechten Seite angehängt. Dieses ergibt bei der Anfrage `Clyde:<color>` das Auswerten des Verweises `RoyalElephant:<color>`. Der Anfragepfad bleibt in diesem Beispiel unverändert, weil ein leerer Pfad als Präfix von der Anfrage abgeschnitten wird und anschließend der sich ergebende Rest an einen leeren Pfad (an den von `RoyalElephant:<>`) angehängt wird. Die weitere Auswertung der Anfrage ergibt dann das Ergebnis `white`.
Ein anderes Beispiel ist die Anfrage `Clyde:<legs>` . `Clyde:<legs>` wird zu dem Verweis `RoyalElephant:<legs>` ausgewertet, dieser wiederum zu `Elephant:<legs>` und der letzendlich zu `four`.
Die gegebenen Beispiele beschreiben nicht den gesamten Umfang von DATR.

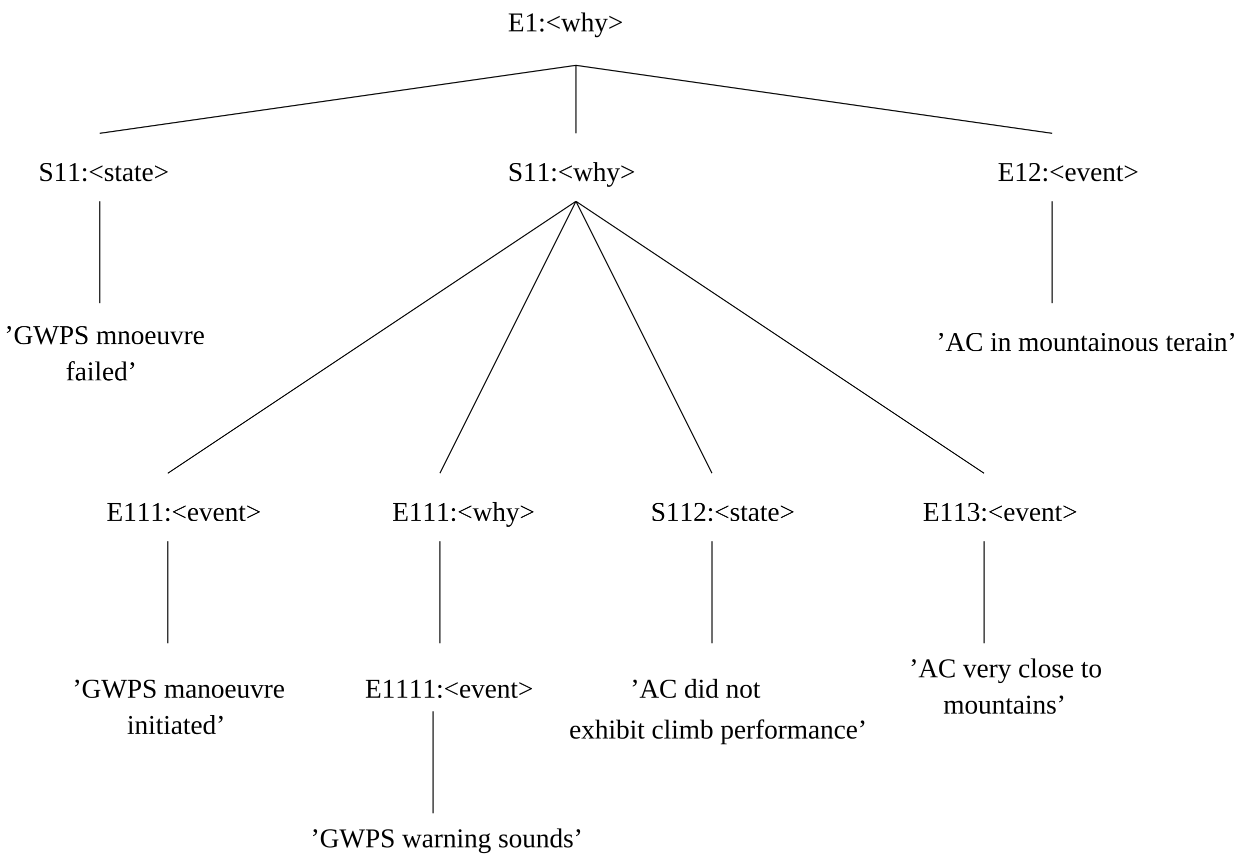

Abbildung 2.2: Ein DATR Programm als Graph dargestellt

```
Elephant:
<color> == grey
<legs>  == four.

RoyalElephant:
<>      == Elephant:<>
<color> == white.

Clyde:
<> == RoyalElephant:<>.

Tom:
<>     == Elephant:<>
<legs> == three.
```

Abbildung 2.3: Ein DATR Programm mit Benutzung von defaults

Sie sollen vielmehr einen Eindruck von DATR geben. Eine genaue Beschreibung von DATR wird mit der formalen Semantik in den folgenden Kapiteln gegeben.

Kapitel 3

Die Syntax von DATR

3.1 Konkrete und abstrakte Syntax

Im folgenden Abschnitt wird die Beziehung zwischen abstrakter Syntax und konkreter Syntax hergestellt. Die konkrete Syntax beschreibt die syntaktisch zulässigen DATR Programme, die ein Parser für DATR verarbeiten muß . Die Bedeutung eines DATR Programms, also die Semantik, ergibt sich jedoch aus der Struktur des Programms, die durch den konkreten Programmtext beschrieben wird. Die Aufgabe der abstrakten Syntax ist die Beschreibung der Struktur, nicht unbedingt der konkreten Darstellung eines Programms. Der Abstraktionsgrad der abstrakten Syntax ist nicht festgelegt. Es sollten in der abstrakten Syntax jedoch die Phrasen der konkreten Syntax leicht wiedererkennbar sein.
In DATR gibt es viele abkürzende Schreibweisen für Programmkonstrukte. Diese ändern jedoch nicht die Struktur des Programms. Die Abkürzungen können jederzeit durch die entsprechenden ausführlichen Schreibweisen ersetzt werden. Um in den folgenden Kapiteln die Bedeutung eines DATR Programms zu beschreiben, genügt es die syntaktischen Phrasen der abstrakten Syntax zu beschreiben, da jedes Programm in konkreter Syntax einem Programm in abstrakter Syntax entspricht. Aus praktischen Gründen wird eine Semantik in der Regel auf der abstrakten Syntax definiert.
Variablen spielen in DATR eine besondere Rolle. Sie sind wichtig, weil sich durch diese die Ausdruckskraft von DATR stark erhöht, wie zum Beispiel der *bubble sort* von Dafydd Gibbon zeigt [15]. Variablen können als Makros implementiert werden. Damit sind Variablen eigentlich kein Teil der abstrakten Syntax, weil ein Makro nicht die Struktur eines Programms ändert. Eine Implementation von Variablen als Makros ist jedoch ungünstig und schränkt die Anwendbarkeit von Variablen ein.

3.2 Die konkrete Syntax von DATR

$$
\begin{array}{lcl}
<theory> & ::= & (<comment> \mid <exvardecl>) \\
 & & (<comment> \mid <vardeclseq>) \\
 & & <definitions> \\
<definitions> & ::= & <sentence><definitions> \mid \\
 & & <comment><definitions> \mid <epsilon> \\
<sentence> & ::= & <node>:<eqseq> . \\
<eqseq> & ::= & <equation> \mid <equation><eqseq> \\
<equation> & ::= & <<atomseq>>=<atomval> \mid \\
 & & <<atomseq>>==<descval> \\
<atomval> & ::= & <atomseq> \mid (<atomseq>) \\
<atomseq> & ::= & <epsilon> \mid <atom><atomseq> \\
<descval> & ::= & <descseq> \mid (<descseq>) \\
<desqseq> & ::= & <epsilon> \mid <desc><descseq> \\
<desc> & ::= & <atom> \mid \text{“} <spec> \text{“} \mid <spec> \\
<spec> & ::= & <node>:<<descseq>> \mid \\
 & & <node> \mid <<descseq>> \\
<node> & ::= & <upper\_case><char\_string> \\
<constant> & ::= & <not\_upper\_case><char\_string> \\
<constantseq> & ::= & <constantseq><constant> \mid <epsilon> \\
<atom> & ::= & <not\_upper\_case><char\_string> \mid \\
 & & '<any\_string>' \mid <varname> \\
<char\_string> & ::= & <epsilon> \mid <char><char\_string> \\
<any\_string> & ::= & <epsilon> \mid <any><any\_string> \\
<res\_char> & ::= & [: \; < \; > \; \text{“} \; \# \; = \; == \; \$ \; ' \; . \; \%] \\
<char> & ::= & [I2CHAR(33)...I2CHAR(127)] - [<res\_char>] \\
<any> & ::= & [I2CHAR(0)...I2CHAR(127)] \\
<upper\_case> & ::= & [A - Z] \\
<not\_upper\_case> & ::= & [<char>] - [<upper\_case>] \\
<exvardecl> & ::= & \# \; qnode :<node> .\# \; qpath :<<constantseq>> . \mid \\
 & & <epsilon> \\
<vardeclseq> & ::= & <vardecl><vardeclseq> \mid <epsilon> \\
<vardecl> & ::= & \# \; vars \;\; <varname>: \;\; <decl> . \\
<decl> & ::= & <atomseq> \mid <varname> + <varname> \mid \\
 & & <varname> - <varname> \mid <varname><varname> \\
<varname> & ::= & \$ <charstring> \\
<comment> & ::= & \% <any\_string'><cr> \mid \\
 & & <comment> \% <any\_string'><cr> \\
<any\_string'> & ::= & <any'><any\_string'> \mid <epsilon> \\
<any'> & ::= & [(I2CHAR(0)...I2CHAR(127)) - I2CHAR(13)] \\
<cr> & ::= & I2CHAR(13)
\end{array}
$$

3.3 Die abstrakte Syntax von DATR

$$
\begin{array}{lll}
<\mathit{theory}> & ::= & <\mathit{exvardecl}><\mathit{vardeclseq}><\mathit{sentence\_decl}> \\
<\mathit{sentence\_decl}> & ::= & <\mathit{sentence}><\mathit{sentence\_decl}>|\mathit{epsilon} \\
<\mathit{sentence}> & ::= & <\mathit{node}>:<<\mathit{atomseq}>>==<\mathit{descseq}>. \\
<\mathit{descseq}> & ::= & <\mathit{epsilon}> \mid <\mathit{desc}><\mathit{descseq}> \\
<\mathit{desc}> & ::= & <\mathit{atom}> \mid <\mathit{node}>:<<\mathit{descseq}>> \mid \\
 & & ``<\mathit{node}>:<<\mathit{descseq}>>``| \\
 & & ``<<\mathit{descseq}>>``|``<\mathit{node}>`` \\
<\mathit{node}> & ::= & <\mathit{upper\_case}><\mathit{char\_string}> \\
<\mathit{atom}> & ::= & <\mathit{constant}> \mid <\mathit{varname}> \\
<\mathit{atomseq}> & ::= & <\mathit{atom}><\mathit{atomseq}> \mid <\mathit{epsilon}> \\
<\mathit{constant}> & ::= & <\mathit{not\_upper\_case}><\mathit{char\_string}> \\
<\mathit{constantseq}> & ::= & <\mathit{constantseq}><\mathit{constant}> \mid <\mathit{epsilon}> \\
<\mathit{char\_string}> & ::= & <\mathit{epsilon}> \mid <\mathit{char}><\mathit{char\_string}> \\
<\mathit{res\_char}> & ::= & [:\ <\ >\ ``\ (\ )\ \#\ =\ ==\ \$\ .] \\
<\mathit{char}> & ::= & [I2CHAR(33)...I2CHAR(127)] \\
 & & -[<\mathit{reschar}>] \\
<\mathit{upper\_case}> & ::= & [A-Z] \\
<\mathit{not\_upper\_case}> & ::= & [<\mathit{char}>]-[<\mathit{upper\_case}>] \\
<\mathit{exvardecl}> & ::= & \#\ \mathit{qnode}:<\mathit{node}>.\#\ \mathit{qpath}:<<\mathit{constantseq}>>.| \\
 & & <\mathit{epsilon}> \\
<\mathit{vardeclseq}> & ::= & <\mathit{vardecl}><\mathit{vardeclseq}> \mid <\mathit{epsilon}> \\
<\mathit{vardecl}> & ::= & \#\ \mathit{vars}\ \ <\mathit{varname}>:\ \ <\mathit{decl}>. \\
<\mathit{decl}> & ::= & <\mathit{not\_upper\_case}><\mathit{char\_string}><\mathit{decl}> \mid \\
 & & <\mathit{epsilon}> \\
<\mathit{varname}> & ::= & \$<\mathit{charstring}>
\end{array}
$$

3.4 Beziehung zwischen abstrakter und konkreter Syntax in DATR

In diesem Abschnitt wird beschrieben, welche Entsprechungen einzelne Elemente der konkreten Syntax in der abstrakten Syntax haben.
$< comment >$ bezeichnet in der konkreten Syntax einen Kommentar. Ein Kommentar ist für die Semantik eines Programms irrelevant. Deshalb gibt es dafür keine Entsprechung in der abstrakten Syntax.
$(< atomseq >)$ und $< atomseq >$ haben die gleiche Bedeutung. Für die Programmstruktur sind die Klammern ohne Bedeutung. In der abstrakten Syntax ist deshalb nur $< atomseq >$ definiert.
$(< descseq >)$ und $< descseq >$ haben die gleiche Bedeutung. Deshalb ist nur $< descseq >$ in der abstrakten Syntax definiert.
Es gibt eine Deskriptorsequenz *descseq*, die nur aus Atomen besteht. In der konkreten Syntax gibt es ein Nichtterminal $< atomval >$ und ein Nichtterminal $< descval >$. Es genügt in der abstrakten Syntax $< descval >$ eine Bedeutung zu geben, weil damit $< atomval >$ implizit eine Bedeutung gegeben wird. Also wird in der abstrakten Syntax $<< atomseq >>=< atomval >$ nicht aufgeführt und nur $<< atomseq >>==< descseq >$ eine Bedeutung gegeben.
In der abstrakten Syntax gibt es kein Atom, das aus einer beliebigen Folge von Charaktern, eingeschlossen in Hochkommmata, besteht. (Die syntaktische Variable $< any\_string >$.) Dieses Atom wird semantisch nicht anders behandelt als ein Atom $< constant >$. Die Definition $< decl >::=< varname > + < varname >$ gibt es nur in der konkreten Syntax. Die Bedeutung von $< varname > + < varname >$ ist eine Sequenz von Atomen, die aus der Verkettung der zwei Sequenzen von Atomen besteht, die jeweils durch die beiden Variablennamen definiert sind. Es genügt also für die Definition $< decl >::=< atomseq >$ eine Bedeutung festzulegen. Dann ergibt sich die Bedeutung von $< decl >::=< varname > + < varname >$ automatisch durch Ersetzung von *varname* durch die dazugehörige Sequenz von Atomen. Im folgenden sind drei Variablendeklarationen zu sehen. Die Variable `$i` kann die Atome `a e i o u` annehmen. Entsprechendes gilt für `$j`.

```
#vars $i : a e i o u
#vars $j : a b c d
#vars $k : $i + $j
```

Die Werte, die `$k` annehmen kann, sind die Atome `a b c d e i o u`.
Die Bedeutung von $< varname > - < varname >$ ist eine Sequenz von

Atomen. Diese Sequenz besteht aus allen Atomen, die in der Sequenz vorkommen, die durch den ersten Variablennamen definiert sind, abzüglich aller Atome die in der Sequenz vorkommen, die durch den zweiten Variablennamen definiert sind. Da es sich auch hier um eine abkürzende Schreibweise handelt, wird die Definition $< \mathit{decl} >::=< \mathit{varname} > - < \mathit{varname} >$ in der abstrakten Syntax nicht aufgeführt. Beispiel:

```
#vars $i : a e i o u
#vars $j : a b c d
#vars $k : $i - $j
```

Die Werte, die `$k` annehmen kann, sind die Atome `e i o u`.
Die Bedeutung von $< \mathit{node} >:< \mathit{eqseq} >$. ist die Bedeutung einer Menge von Sätzen. Diese Sätze haben die Form $< \mathit{node} >$: *equation*. Ein Beispiel zeigt den Zusammenhang.

```
N: <ab> = cd
   <ef> = gh.
```

hat die Bedeutung:

```
N: <ab> = cd.
N: <ef> = gh.
```

Der Knoten $< \mathit{node} >$ vor einer $< \mathit{eqseq} >$ wird vor jeder $< \mathit{equation} >$ von $< \mathit{eqseq} >$ gesetzt.
In der konkreten Syntax sind sieben verschiedene Deskriptoren definiert. Davon sind zwei Deskriptoren nur abkürzende Schreibweisen. Bei $< \mathit{desc} >::=< \mathit{node} >$ fehlt auf der rechten Seite der Definition ein Pfad. Der Knoten auf der rechten Seite wird um den Pfad p der linken Seite ergänzt.
In der Gleichung $N :< \mathit{desc}_1 > ::= << \mathit{desc}_2 >>$ fehlen auf der rechten Seite zwei Knoten. Für die fehlenden Knoten wird jeweils der Knoten der linken Seite der Gleichung eingesetzt.

```
N:<abc> = <abc <def>>. <=> N:<abc>  = N:<abc N:<def>>.
N1:<abc> = N2.         <=> N1:<abc> = N2:<abc>.
```

3.5 Variablen in DATR

In DATR können Variablen definiert werden. Das erste Zeichen eines Variablennamens ist immer ein $. Eine Variablendeklaration bestimmt den Wertebereich einer Variablen. Die Deklaration `# vars $vowels: a e i o u` bedeutet, daß die Variable `$vowels` die Atome `a e i o u` als Werte annehmen kann. Es können auch Variablen in einem DATR Programm benutzt werden, die nicht deklariert wurden. Wenn eine Variable nicht deklariert wurde, so bedeutet dieses, daß die Variable alle Atome als Werte annehmen kann, die in dem DATR Programm vorkommen. Das Auslassen einer Variablendeklaration ist also eine abkürzende Schreibweise für eine Variablendeklaration.
Mit der Deklaration oder der Nicht-Deklaration einer Variablen wird also der Wertebereich einer Variablen festgelegt.
Variablen können in Pfaden benutzt werden. Wenn in einer Satzdeklaration auf der linken Seite einer Gleichung in einem Pfad eine Variable vorkommt, so wird damit eine Menge von Gleichungen beschrieben. Die Menge von Gleichungen ergibt sich, wenn man die Werte aus dem Wertebereich für die Variable in den Pfad einsetzt. Wenn die Variable auch auf der rechten Seite der Gleichung vorkommt, so muß sie durch den gleichen Wert ersetzt werden, durch den sie auch auf der linken Seite ersetzt wurde. Als Beispiel für die Benutzung von Variablen sei ein DATR Programm gegeben, das die Atome \/ und /\ durch `or` und `and` in einer Formel ersetzt, die über dem Alphabet `(,), True, False,` \/ und /\ definiert ist.

```
#vars $abc : ( ) True False
LOGIK:<>     ==
      <$abc> == $abc LOGIK:<>
      <\/>   == or LOGIK:<>
      </\>   == and LOGIK:<>.
```

Der vorhergehende Satz der DATR Theorie hat die gleiche Bedeutung, wie der folgende Satz:

```
LOGIK:<>        ==
      < ( >     == ( LOGIK:<>
      < ) >     == ) LOGIK:<>
      <True>    == True LOGIK:<>
      <False>   == False LOGIK:<>
      < \/ >    == or LOGIK:<>
      < /\ >    == and LOGIK:<>.
```

Beide Sätze benutzen zur Definition *definition by default* , wie es schon in dem Beispiel mit den Elefanten Clyde und Tom benutzt wurde.
Bei der Verwendung von mehreren Variablen in einem Pfad kann die Menge der so definierten Gleichungen sehr groß sein. Falls vier verschiedene Variablen, die jeweils 10 verschiedene Werte annehmen können, in einem Pfad benutzt werden, so können dadurch 10000 Gleichungen definiert sein, wenn alle Wertebereiche der Variablen disjunkt sind. Es ist daher sinnvoll, schon in einer Semantik von DATR einen direkten Implementationsvorschlag für die Behandlung von Sätzen mit Variablen zu machen, um eine effiziente Implementation von Variablen zu garantieren. Damit wird die Einsatzmöglichkeit von DATR vergrößert.

3.6 Mechanismen zur Variablenbehandlung

Es gibt verschiedene Auffassungen, welche Bedeutung Variablen in DATR haben sollten, wenn sie nicht als Makros aufgefaßt werden. Zwei verschiedene Mechanismen können gerechtfertigt werden. Beide unterscheiden sich, wenn auf der rechten Seite einer Gleichung eine Sequenz von Deskriptoren steht, die länger als eins ist, und wenn bei der Auswertung dieser Deskriptoren Gleichungen mit Variablen verwendet werden.
Zur Erläuterung dieses Sachverhaltes diene folgendes DATR Programm:

```
#vars $i : a b.
N1:<$i>   == N2:<a> N2:<b c>.
N2:<>     == c.
N2:<$i>   == a.
N2:<$i c> == b.
```

Die Anfrage `N1:<a>` kann nun nach einem Mechanismus zu dem Ergebnis `a c` und nach einem anderen Mechanismus zu dem Ergebnis `a b` ausgewertet werden. Diese Mechanismen ergeben sich wie folgt :
Das erste Ergebnis `a c` ergibt sich, weil durch die Anfrage `N1:<a>` die Variable `$i` an `a` gebunden wird und die Auswertung von `N2:<a>` und `N2:<b c>` weiter ausgewertet werden in einer Umgebung, in der `$i` an `a` gebunden ist.
Das zweite Ergebnis ergibt sich, in dem zuerst durch die Anfrage `N1:<a>` die Variable `$i` an `a` gebunden wird, anschließend bei den Deskriptoren `N2:<a>` und `N2:<b c>` diese Bindung ,vergessen ‘wird, so daß diese Deskriptoren, unabhängig ausgewertet, auf die linken Seiten `N2:<$i>` und `N2:<$i c>` passen.
In dieser Arbeit wurde der erste Mechanismus gewählt. Aus folgenden Grund:

Wenn `N2:<a>` zu `N2:<$i>` und `N2:<b c>` zu `N2:<$i c>` paßt, dann sollte man das Programm auch so schreiben können:

```
#vars $i : a b.
N1:<$i>   == N2:<$i> N2:<$i c>.
N2:<>     == c.
N2:<$i>   == a.
N2:<$i c> == b.
```

In diesem Programm kann jedoch `N1:<$i> == N2:<$i> N2:<$i c>` nur als `N1:<a> == N2:<a> N2:<a c>` oder `N1:<b> == N2:<b> N2:<b c>` interpretiert werden. Wenn auch die Sätze `N1:<a> == N2:<a> N2:<b c>` und `N1:<a> == N2:<b> N2:<a c>` gelten würden, dann wäre das DATR Programm keine partielle Funktion mehr. Nach Gazdar [6] ist das jedoch von DATR gefordert.

Kapitel 4

Die Semantik von DATR

4.1 Motivation zur Entwicklung einer Semantik

Mit der Definition einer Semantik für eine Programmiersprache werden vier wichtige Punkte verfolgt:

- Eine Semantik definiert einen Standard für eine Implementation. Damit wird garantiert, daß sich verschiedene Implementationen einer Sprache gleich verhalten.

- Ein Programmierer, der den Formalismus der Semantik kennt, kann die Semantikbeschreibung der Sprache als ein Manual benutzen, um Fragen bezüglich der Bedeutung von Programmkonstrukten zu klären.

- Mit der Definition einer Semantik können Eigenschaften der Sprache analysiert werden. Damit kann die Pragmatik der Sprache untersucht werden, bevor die Sprache implementiert wird und Änderungen an der Semantik der Sprache vorgenommen werden. Außerdem kann mit der Festlegung einer Semantik ein Vorschlag für elegante Implementationen gemacht werden.

- Eine formale Semantik kann als Eingabe für einen Compilergenerator dienen. Der Nutzen eines Compilergenerators liegt auf der Hand: Bei richtiger Spezifikation und Implementation des Compilergenerators entfallen zukünftige fehlerträchtige Implementationen einer Sprache durch einen Programmierer.

Es sind verschiedene Formalismen zur Definition von Semantik üblich. Im folgenden werden drei bekannte Formalismen untersucht: *Interpreter Semantik,* *Denotationale Semantik und Natural Semantics.* Die Interpreter Semantik wird hier nur kurz beschrieben, da sie zur Beschreibung von DATR nicht geeignet ist .

4.2 Interpreter-Semantik und Denotationale Semantik

4.2.1 Die Interpreter Semantik

Die Interpreter Semantik benutzt einen Interpreter um eine Semantik zu definieren. Die Bedeutung eines Programms ist die Folge der Zustände, die der Interpreter durchläuft. Diese Methode ist die älteste Semantikdefinition. Sie hat jedoch zwei wesentliche Nachteile: Der Definition liegt ein konkretes Maschinenmodell zugrunde und die Bedeutung einzelner Programmkonstrukte kann nur durch die Sequenz von Zuständen beschrieben werden, die der Interpreter bei der Ausführung des Programmkonstrukts durchläuft. In der Praxis möchte man jedoch aus Geschwindigkeitsgründen keinen Interpreter, sondern einen Compiler zur Ausführung von Programmen haben.

4.2.2 Die Denotationale Semantik

Die Denotationale Semantik ist nach Watt [45] durch drei Eigenschaften charakterisiert:

- Die Bedeutung jeder Phrase **p** einer Sprache wird als ein Wert **d** einer Domäne spezifiziert. **d** wird die *Denotation* der Phrase **p** bezeichnet.
- Zu jeder Klasse von Phrasen **P** wird eine Klasse **D** der Denotationen spezifiziert und eine *Semantische Funktion f*, die jede Phrase in **P** auf ihre Denotation in **D** abbildet. ($f : P \rightarrow D$).
- Die Semantische Funktion **f** wird durch eine Menge von *Semantischen Gleichungen* definiert. Jede Gleichung gilt für eine Phrase in **P**. Die Denotation einer Phrase wird nur durch die Bedeutung der Subphrasen erklärt.

Die Denotationale Semantik hat eine vollständig mathematische Grundlage. Damit können die im vorhergehenden Abschnitt beschriebenen Anforderungen eins und drei prinzipiell erfüllt werden : die Beschreibung kann einen Standard für Implementationen bilden und das Verhalten eines Programms

kann vorhergesagt werden, ohne daß dieses auf einem Computer direkt ausgeführt wird. Dieses ist durch einfaches Anwenden der definierten Gleichungen auf ein Programm in vorgegebener Syntax möglich. Die Gleichungen sollten als ein Algorithmus für die Interpretation der Programmiersprache dienen.
Nach Watt [45] erweist es sich als praktisch, die Gleichungen in eine funktionale Sprache wie ML oder Miranda zu übertragen. Das sollte möglich sein, da funktionale Sprachen, ebenso wie die Denotationale Semantik, auf *Churchs* λ-*Kalkül* basieren. Damit hat man einen Interpreter für die beschriebene Sprache entwickelt und kann die Gleichungen zur Interpretation einer Phrase maschinell anwenden. Dieser Interpreter kann natürlich nicht mit dem Speicherplatzbedarf und der Laufzeiteffizienz eines compilierten Programms konkurrieren. Der Interpreter dient einzig als Prototyp um Kenntnisse von Eigenschaften einer Sprache und von günstigen Implementationen zu erlangen. Zu diesem Zweck darf das Abstraktionsnivieau der Beschreibung in Denotationaler Semantik nicht zu hoch sein. Die Wahl des Abstraktionsniveaus ist durch die Denotationale Semantik an sich in keiner Weise festgelegt und allein die Entscheidung des Designers der Sprache. Um das Verhalten eines Programms zu untersuchen, muß neben einer Semantik in einem nicht zu hohen Abstraktionsniveau zumindest eine abstrakte Syntax vorliegen. Diese wird in der Regel durch eine kontextfreie Grammatik beschrieben.
Als Metasprache bei der Definition der Denotationalen Semantik wird der $\lambda-$*Kalkül* benutzt.

Die abstrakte Syntax des $\lambda-$*Kalküls* :

$$\begin{array}{lll} <\lambda - expression> & ::= & <variable> \mid <constant> \\ & & <application> \mid <abstraction> \\ <application> & ::= & (<\lambda - expression>) <\lambda - expression> \\ <abstraction> & ::= & \lambda <variable> . <\lambda - expression> \end{array}$$

Konventionen und Semantik des $\lambda-Kalk\ddot{u}ls$:

- $(\lambda x_1.(\lambda x_2.\cdots.(\lambda x_n.t)\ldots)) = (\lambda x_1 x_2 \ldots x_n.t)$
- Applikation ist linksassoziativ: $t_1 t_2 \ldots t_n = ((t_1 t_2) \ldots t_n)$

Die Menge der freien Variablen eines λ-Ausdrucks E wird mit $\phi(e)$ bezeichnet und ist induktiv definiert:

- $\phi(c) = \{\}$, falls c eine Konstante ist
- $\phi(x) = \{x\}$, falls x eine Variable ist
- $\phi(\lambda x.P) = \phi(P) - \{x\}$
- $\phi(P(Q)) = \phi(P) \cup \phi(Q)$

Eine Variable x in einem λ - Ausdruck E ist gebunden, wenn sie in einem Subterm der Form $\lambda x.P$ vorkommt; sonst heißt sie frei.
Die Substitution von Q für alle freien Vorkommen der Variable x in P, $[Q/x]P$ ist induktiv definiert:

- $[Q/x]x = Q$
- $[Q/x]y = y$
- $[Q/x]\lambda x.E = \lambda x.E$
- $[Q/x]\lambda y.E = \lambda y.[Q/x]E$,falls $x \neq y$ und $x \notin \varphi(\mathrm{E}) y \notin \varphi(\mathrm{Q})$
- $[Q/x]\lambda y.E = \lambda z.[Q/x]\{z/y\}E$
  für z nicht frei und nicht gebunden in $(E)Q$ und $x \neq z \neq y$. Falls $x \neq y$ dann $x \in \varphi(\mathrm{E})$ und $y \in \varphi(\mathrm{Q})$
- $[Q/x](E_1)E_2 = ([Q/x]E_1)[Q/x]E_2$

Es gelten folgende Regeln im λ Kalkül:

- α Konversion : $\lambda x.E \leftrightarrow (\lambda y.E[y/x])$
- β Reduktion: $(\lambda x.E)M \leftrightarrow E[M/x]$
- δ Reduktion: Auswertung von Basisfunktionen
- η Reduktion: $(\lambda x.Ex) \leftrightarrow E$

Schwierigkeiten bei der Denotationalen Semantik entstehen, wenn die Korrektheit einer konkreten Implementation gezeigt werden soll. Es muß gezeigt werden, daß die Implementation alle Eigenschaften der Beschreibung in Denotationaler Semantik hat. Bisher wurden zwei Methoden verwendet: Aus der kombinatorischen Logik ist bekannt, daß sich jeder λ-Ausdruck in einen gleichbedeutenden Kombinatorausdruck, in dem nur die Kombinatoren S,K und I vorkommen, übersetzen läßt. (Curry und Feys [3]). Turner [44] hat die Idee verfolgt, diese Kombinatoren in Hardware zu implementieren. Damit können Programme, die in Denotationaler Semantik beschrieben sind, durch Übersetzung des $\lambda-$ Ausdrucks in einen Kombinatorausdruck direkt ausgeführt werden. Dieses ist jedoch nicht auf konventionellen Maschinen möglich.
Eine ausführliche Darstellung ist in Peyton Jones [41] zu finden. Dort wird beschrieben, wie ein Programm in Form eines $\lambda-$*Ausdrucks* durch einen $\lambda-$*lifter* mit dem *SK compilation algorithm* in *Supercombinator Form* übersetzt wird. Das Programm in *Supercombinator-Form* wird anschließend in $G-Code$ übersetzt und eine Virtuelle Maschine entwickelt, die den *G-Code* ausführen kann. Die Korrektheit eines *G-machine compilers* wurde von Lester [28] gezeigt.

Eine andere Methode um einen beweisbar richtigen Compiler für eine Denotationale Semantik zu erhalten ist die Konstruktion von λ-*Kalkül* Interpretierern . Mosses [30] hat eine Maschine implementiert, die einen λ-Ausdruck durch schrittweise Reduktion zu einen Wert reduziert. Es wurden von anderen ähnliche Maschinen konstruiert. Sie sind jedoch alle für reale Anwendungen viel zu langsam.
Hierachische Beweismethoden mit Refinementmapping, wie zum Beispiel TLA+ (Temporal Logic of Actions), sind wünschenswert um zu zeigen, daß eine Implementation eine Beschreibung in einem Semantikformalismus erfüllt. Logische Verfahren , wie zum Beispiel TLA+, können jedoch nicht für die Denotationale Semantik verwendet werden, weil diese Semantik auf dem λ-*Kalkül* basiert und sich dieser nicht durch allgemeine Regeln in Logik übersetzen läßt.
Da etwa zwei Jahrzehnte Forschung auf dem Gebiet der Denotationalen Semantik nicht zu einem Verfahren geführt haben, das die Entwicklung von effizienten,beweisbar richtigen Compilern ermöglicht, ist die Denotationale Semantik zur Zeit kein wichtiger Formalismus mehr zur Formulierung einer Semantik.
Keller [21] hat eine Denotationale Semantik für DATR entworfen. Diese Semantik ist sehr kompakt und stellt DATR auf einem sehr hohen Abstraktionsniveau dar. Die Semantik von DATR ist ohne den externen globalen Kontext,

Die SKI-Regeln:

S-reduction	$Sfgx \to fx(gx)$
K-reduction	$Kcx \to c$
I-reduction	$Ix \to x$
I-transformation	$\lambda x.x \Rightarrow I$
K-transformation	$\lambda x.c \Rightarrow Kc(c \neq x)$
S-transformation	$\lambda x.e_1 e_2 \Rightarrow S(\lambda x.e_1)(\lambda x.e_2)$

Algorithmus zur Auswertung des Ausdrucks e

WHILE e enthält λ-Abstraction **do**

1. Wähle das *innermost* $\lambda - Abstraction$ von e
2. Wenn der *body* eine *application* ist, dann wende die S-transformation an
3. Sonst ist der *body* eine Variable oder Konstante. Dann wende die K oder I Transformation an.

Abbildung 4.1: Implementierung und Compilierung von λ-Ausdrücken

$$
\begin{array}{lcl}
[\![\mathbf{a}]\!]_{\mathbf{c}} & = & F(a) \\
[\![\mathbf{N} : \langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle]\!]_{\mathbf{c}} & = & \begin{cases} \text{if } v_i = [\![\mathbf{d_i}]\!]_{\mathbf{c}} \text{ is defined for each } i (1 \leq i \leq n) \text{ ,} \\ \text{then } \kappa(c)(F(N), v_1 \cdots v_n) \\ \text{undefined otherwise} \end{cases} \\
[\![\text{"}\mathbf{N} : \langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle\text{"}]\!]_{\mathbf{c}} & = & \begin{cases} \text{if } v_i = [\![\mathbf{d_i}]\!]_{\mathbf{c}} \text{ is defined for each } i (1 \leq i \leq n), \\ \text{then } \kappa(c')(c') \text{ where } c' = (F(N), v_1 \cdots v_n) \\ \text{undefined otherwise} \end{cases} \\
[\![\text{"}\langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle\text{"}]\!]_{\mathbf{c}} & = & \begin{cases} \text{if } v_i = [\![\mathbf{d_i}]\!]_{\mathbf{c}} \text{ is defined for each } i (1 \leq i \leq n), \\ \text{then } \kappa(c')(c') \text{ where } c = (u, v) \text{ and } c' = (u, v_1 \cdots v_n) \\ \text{undefined otherwise} \end{cases} \\
[\![\text{"}\mathbf{N}\text{"}]\!]_{\mathbf{c}} & = & \left\{ \kappa(c')(c') \text{ where } c = (u, v) \text{ and } c' = (F(N), v) \right.
\end{array}
$$

Abbildung 4.2: Die Semantik von Deskriptoren in Keller [21]

bestehend aus externen globalen Knoten und externen globalen Pfad, sowie ohne Variablen in DATR beschrieben.
Keller [21] beschreibt die Denotationale Semantik von DATR wie folgt:

Eine DATR Interpretation ist ein Tripel $I = (U, \kappa, F)$, bei der gilt:

- *U ist eine Menge*
- *κ ist eine Funktion, die jedem Element der Menge $(U \times U^*)$ eine partielle Funktion von $(U \times U^*)$ nach U^* zuordnet*
- *F ist eine Auswertungsfunktion, die jedem Knoten N und jedem Atom a ein Element von U zuordnet, so daß verschiedenen Atomen verschiedene Elemente zugeordnet werden.*

Keller [21] :
$C = (U \times U^*)$ wird der globale Kontext genannt. U^* ist die Domäne der Werte/Pfade. Mit der Interpretation I wird eine Funktion $[\![]\!]$: $\mathbf{DESC} \rightarrow (\mathbf{C} \rightarrow \mathbf{U}^*)$ assoziert. κ ist eine Abbildung von globalen Kontexten auf lokale Kontexte zu Werten.

Er benutzt keine kontextfreie Grammatik um die Form von DATR Sätzen zu beschreiben.
Schmidt [42] S. 63 : *A denotational definition of a language consists of three parts: the abstract syntax definition of the language, the semantic algebra, and the valuation function.*

Wir finden diese drei Teile sind bei Kellers Semantik nicht übersichtlich dargestellt.
Im folgenden soll die Semantik von folgenden einfachen Programm bestimmt werden:

```
N1 : < a b > == N2 : <b>.
N2 : < b >   == c.
```

Die Anfrage soll lauten `N1 : < a b >`. Durch Anwendung der ersten beiden Gleichungen muß $\kappa(F(N1)$, *(F(a) , F(b)))* $(F(N1)$,*(F(a), F(b)))* ausgewertet werden. Die Semantik einer Anfrage an ein DATR Programm scheint bei Keller nicht deutlich definiert zu sein. Der globale Kontext einer Anfrage ist der Knoten und der Pfad der Anfrage. Diesen globalen Kontext haben wir bei der Anfrage eingesetzt. Eine formale Definiton von F wäre brauchbar. Die Funktion κ scheint in Kellers Semantik [21] nicht explizit definiert worden zu sein. Es wird nur beschrieben, daß ihr Ergebnis eine Sequenz von Elementen der Menge U ist. Die Angabe eines Wert- und Bildbereich ist keine ausreichende Definition. Im obigen Programm sollte das zu dem Wert $(F(N1)$, *(F(a), F(b)))* das Element $F(c)$ des Bildbereichs sein.
Wir schließen : die Funktion ist nicht formal genug beschrieben worden.
Da die Funktion κ nicht explizit definiert wurde, ist nicht klar, ob die Satzumgebung in ihr enthalten ist. Die Bedeutung eines Deskriptors hängt von dem Programm ab, in dem er vorkommt. Deshalb schlagen wir vor, die Deskriptoren um eine Funktion zu ergänzen, die die Satzdeklarationen beschreibt.
Es ist besser $[\![\mathbf{N} : \langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle]\!]_{\mathbf{c},\mathbf{\Gamma}}$ zu schreiben, um zu verdeutlichen, daß die Bedeutung eines Deskriptor von einem Programm abhängt.
Wir finden es einfacher, die Bedeutung des Deskriptors $(F(N1)$, *(F(a) , F(b)))* schrittweise über die Bedeutung von $F(N2) :< F(b) >$ zu definieren.
(Bei Kellers Semantik ist es eine Abbildung auf eine Sequenz von Atomen, die direkt durch κ definiert ist. Eine schrittweise Ableitung würde als Argument der Funktion κ eine weitere Funktion verlangen, die die Satzdeklarationen beschreibt.)
Zitat Keller [21] :
a finite set of DATR sentences Γ *is called a DATR theory. ...*
$\Gamma/N = \{s \in \Gamma | s = N : P == \phi\}$ ...
$[\![\mathbf{\Gamma/N}]\!]_{\mathbf{c}} = \bigcup_{\mathbf{N:P==\phi\in\Gamma}} \{(\mathbf{F(P)}, [\![\phi]\!]_{\mathbf{c}})\}$...
for $\phi = d_1 \cdots d_n$ *we write* $[\![\phi]\!]_{\mathbf{c}}$ *to denote* $v_1 \cdots v_n \in U^*$,*if* $v_i = [\![\mathbf{d_i}]\!]_{\mathbf{c}}$ *is defined (and* $[\![\phi]\!]_{\mathbf{c}}$ *is undefined otherwise) ...*
Hier wird eine Menge mit Hilfe von $[\![\mathbf{d_i}]\!]$ definiert. Später wird $[\![\mathbf{d_i}]\!]$ über die Funktion κ definiert. Die Funktion κ wird nirgendwo definiert. Deshalb ist die Menge $[\![\mathbf{\Gamma/N}]\!]_{\mathbf{c}}$ nicht genau definiert.

Es scheint, hier wurde nicht übliche Notation verwendet. ϕ in Semantischen Klammern ($[\![\phi]\!]_{\mathbf{c}}$) sagt üblicherweise nichts über die Bedeutung von ϕ aus. Die Semantischen Klammern sagen lediglich aus, daß es sich bei ϕ um einen Abstrakten Syntaxbaum handelt. (Schmidt [42] S.61 : *The double brackets surrounding the subtrees are used to clearly separate the syntax pieces from the semantic notion ... The valuation functions are applied to the tree (... to determine the meaning of the tree).* Ein Ausdruck in Semantischen Klammern hat also keine Bedeutung, wenn nicht eine Auswertungsfunktion auf ihn angewendet wird. Die Semantischen Klammern bezeichnen nur ein bestimmtes Argument einer Auswertungsfunktion.
Auch bei der Definition der Deskriptoren benutzt Keller keine Auswertungsfunktionen.
Weiteres Zitat von Keller [21] :
Now suppose that $f : U^* \rightarrow (U^* \rightarrow U^*)$ is the function associated with the node Γ/N in a given DATR interpretation. We can define a partial function $\Delta(f) : U^* \rightarrow U^*$ (the default interpretation of Γ/N) as follows. For each $v \in U^*$ set $\Delta(f)(v) = f(v_1)(v_2)$ where $v = v_1 v_2$ and v_1 is the longest prefix of v such that $f(v_1)$ is defined. ...
Hier wird gezeigt, daß Δf über f definiert werden kann. f ist als eine Klasse von Funktionen definiert. Wir hätten die Angabe einer Funktion aus der Klasse von Funktionen f als brauchbar empfunden. Es trat die Frage auf, ob f zu κ einen Bezug hat. Dieses ist möglich, da beide den gleichen Typ haben. In dem Fall, daß κ zu der Klasse der Funktionen f gehört, ist verwirrend, daß die Typangaben von der Syntax her verschieden sind. Falls ein Zusammenhang zwischen den beiden Funktionen κ und f besteht oder die beiden gleich sind (das könnte von dem Typ der Funktionen möglich sein) , so sollte dieses explizit dargestellt werden.
Wenn f und κ die gleichen Funktionen sind, so ist nicht klar, warum sie unterschiedliche Namen haben. Dann sind auch die Typangaben der beiden Funktionen verwirrend, die zwar formal die gleichen sind, jedoch unterschiedlich dargestellt sind.
Die Funktion *longest prefix* sollte auch formal beschrieben werden.
Keller [21] : *The function κ can be thought of as mapping global contexts onto (partial) functions from local contexts to values.* Wenn κ von einer partiellen Funktion abhängt, dann sollte diese Funktion Argument von κ sein. Es ist überhaupt nicht deutlich, wie die partielle Funktion mit κ zusammenhängt.
Die Semantik von DATR mit *defaults* wird nur am Beispiel eines Deskriptors beschrieben und darauf verwiesen, daß es bei den anderen Deskriptoren so ähnlich ist. Wenn Keller den Anspruch auf eine formale Beschreibung erhebt, so müssen auch diese formal definiert werden.
Vergleicht man die Semantik von Keller mit den am Anfang des Kapitels

beschriebenen Eigenschaften, die nach Watt [45] und Schmidt [42] eine Denotationale Semantik charakterisieren, so ergeben sich folgende Kritikpunkte:

- die Gleichungen geben keinen Algorithmus vor, wie die Bedeutung eines Deskriptors hergeleitet werden kann. Damit ist natürlich auch keine Übertragung in eine funktionale Sprache möglich. Weiterhin kann Kellers Semantik keinen Standard für eine Implementation bilden. Durch unzureichende Beschreibung kann zum Beispiel kein Laufzeitverhalten analysiert werden. Die Semantik kann kein Vorschlag für einen Prototypen sein.

- Keller schlägt kein Verfahren vor, wie ein Compiler aus seiner Semantik generiert werden kann.

- Als Manual für DATR eignet sich die Sprache nicht, weil die Funktion κ nicht ausreichend beschrieben wird und die Funktion f nicht in der Definition der Semantik der Deskriptoren vorkommt.

- die Phrasen sind nicht durch eine kontextfreie Grammatik definiert. Deshalb ist es schwierig die Semantik eines DATR Programms in konkreter Syntax abzuleiten.

- Es ist nicht klar, wie bei der Auswertung von Deskriptoren die Satzdeklarationen benutzt werden.

- Die Semantik einer Anfrage wird nicht definiert und damit wird auch nicht definiert, in welchen globalen Kontext eine Anfrage ausgewertet wird.

- Keller hält sich nicht an übliche Notation. Mit den Semantischen Klammern verbindet er implizit eine Auswertungsfunktion. Nach Schmidt [42] bezeichnen die Klammern an sich nur ein besonderes Argument (einen Syntaxbaum), der eine Bedeutung bekommt, indem eine Auswertungsfunktion auf ihn angewendet wird. Ein Ausdruck in Semantischen Klammern ohne Anwendung einer Auswertungsfunktion bezeichnet nicht die Semantik des Ausdrucks.

- Die drei Teile einer Denotationalen Semantik : Abstrakte Syntax, Auswertungsfunktionen und Semantische Algebra sind nicht übersichtlich dargestellt.

Uns war es nicht möglich, aus Kellers Semantik die Bedeutung eines DATR Source-Code abzuleiten.

4.3 Ergänzung von Kellers Semantik

In direkter Anlehnung an Kellers Semantik [21] ist im folgenden eine Denotationale Semantik entwickelt worden. Diese Semantik hat natürlich die Schwäche einer Denotationalen Semantik an sich (kein Vorschlag einer effizienten Implementierung).
Es wurde versucht, alle Funktionenen explizit darzustellen.
Hier wird nicht die Semantik einer externen globalen Variablendeklaration, einer Variablendeklaration, $defaults$ und Variablen beschrieben. Die abstrakte Syntax ist schon im vorherigen Kapitel beschrieben worden. Auf eine Wiederholung wird hier verzichtet. Deshalb wird sofort mit der Semantischen Algebra begonnen.

- U ist eine beliebige Menge.
  D ist die Menge, die alle Elemente beinhaltet, die durch die syntaktischen Variablen *constant* und *node* beschrieben werden.
  Die syntaktischen Variablen sind in der abstrakten Syntax beschrieben.
  $u_1 \cdots u_n$ bezeichnet die Sequenzen aus n Elementen $u_i, i \in 1 \dots n$.
  F ist eine beliebige bijektive Abbildung. $F : D \rightarrow U$.
  F wird wie bei Keller [21] auf Sequenzen erweitert:
  $\forall u_i \in U, i \in 1..n : F(< u_1 ... u_n >) =< F(u_1) ... F(u_n) >$.

- Γ ist die Theorie, die in der abstrakten Syntax durch *theory* beschrieben wird. ϕ ist ein beliebiger Deskriptor, der durch *desc* beschrieben werden kann, N wird durch *node* beschrieben und P durch *atomseq*.
  $NODE$ ist eine Menge, die alle Knoten beinhaltet, die durch das Nichtterminalsymbol *node* beschrieben werden können.
  $DESC$ ist die Menge aller Deskriptoren, deren Elemente durch die syntaktische Variable *desc* beschrieben werden.

$$\begin{array}{lcl} f_{N,\Gamma} & : & U^* \rightarrow DESC \\ f_{N,\Gamma} & = & \bigcup_{N:<P>==\phi\in\Gamma}\{(F(P),\phi)\} \\ \kappa_\Gamma & : & NODE \rightarrow (U^* \rightarrow DESC) \\ \kappa_\Gamma(N) & = & f_{N,\Gamma}, \text{ if } \exists N,P,\phi : (N :< P >== \phi) \in \Gamma \\ & & \text{undef otherwise} \end{array}$$

- U^* ist die Menge der Sequenzen von Symbolen aus U.
  Die Verkettung von n Sequenzen $u_i \in U^*, i \in 1 \dots n$, wird durch $u_1 \cdots u_n$ bezeichnet.

- $C = (U \times U^*)$,
  C ist die Menge aller globalen Kontexte. Ein globaler Kontext c ist ein Tupel bestehend aus einem Knoten und einen Pfad.

- δ ist die Auswertungsfunktion für Deskriptoren.
  $\delta : DESC \rightarrow C \rightarrow (NODE \rightarrow (U^* \rightarrow DESC)) \rightarrow U^*$.
- Elemente a der Menge von Atomen werden durch die die syntaktische Variable *atom* beschrieben.
  Deskriptoren werden mit d beschrieben und Knoten mit N .
- Die Semantik einer Anfrage wird durch die Funktion φ beschrieben :
  $\varphi : DESC \rightarrow \Gamma \rightarrow U^*$

$$\delta[\![(\mathbf{a})]\!]_{\mathbf{c},\kappa_\Gamma} = F(a)$$

$$\delta[\![\mathbf{N} : \langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle]\!]_{\mathbf{c},\kappa_\Gamma} = \begin{cases} \text{if} & \wedge \quad u_i = \delta[\![\mathbf{d_i}]\!]_{\mathbf{c},\kappa_\Gamma} \\ & \qquad \text{is defined for each } i \in (1 \leq i \leq n) \\ & \wedge \quad \kappa_\Gamma(N) = f_{N,\Gamma} \\ & \wedge \quad f_{N,\Gamma}(u_1 \cdots u_n) \text{ is defined} \\ \text{then} & \qquad \delta(f_{N,\Gamma}(u_1 \cdots u_n))_{c,\kappa_\Gamma} \\ \text{otherwise} & \qquad \text{undefined} \end{cases}$$

$$\delta[\![\text{"}\mathbf{N} : \langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle\text{"}]\!]_{\mathbf{c},\kappa_\Gamma} = \begin{cases} \text{if} & \wedge \quad u_i = \delta[\![\mathbf{d_i}]\!]_{\mathbf{c},\kappa_\Gamma} \\ & \qquad \text{is defined for each } i \in (1 \leq i \leq n) \\ & \wedge \quad \kappa_\Gamma(N) = f_{N,\Gamma} \\ & \wedge \quad f_{N,\Gamma}(u_1 \cdots u_n)_{c,\kappa_\Gamma} \text{ is defined} \\ & \wedge \quad c' = (F(N), u_1 \cdots u_n) \\ \text{then} & \qquad \delta(f_{N,\Gamma}(u_1 \cdots u_n))_{c',\kappa_\Gamma} \\ \text{otherwise} & \qquad \text{undefined} \end{cases}$$

$$\delta[\![\text{"}\langle \mathbf{d_1} \cdots \mathbf{d_n} \rangle\text{"}]\!]_{\mathbf{c},\kappa_\Gamma} = \begin{cases} \text{if} & \wedge \quad u_i = \delta[\![\mathbf{d_i}]\!]_{\mathbf{c},\kappa_\Gamma} \\ & \qquad \text{is defined for each } i \in (1 \leq i \leq n) \\ & \wedge \quad c = (N, p) \\ & \wedge \quad F(N1) = N \\ & \wedge \quad \kappa_\Gamma(N1) = f_{N1,\Gamma} \\ & \wedge \quad f_{N1,\Gamma}(u_1 \cdots u_n) \text{ is defined} \\ & \wedge \quad c' = (N, u_1 \cdots u_n) \\ \text{then} & \qquad \delta(f_{N1,\Gamma}(u_1 \cdots u_n))_{c',\kappa_\Gamma} \\ \text{otherwise} & \qquad \text{undefined} \end{cases}$$

$$\delta[\![\text{"}\mathbf{N}\text{"}]\!]_{\mathbf{c},\kappa_\Gamma} = \begin{cases} \text{if} & \wedge \quad \kappa_\Gamma(N) = f_{N,\Gamma} \\ & \wedge \quad c = (N2, u_1 \cdots u_n) \\ & \wedge \quad f_{N,\Gamma}(u_1 \cdots u_n) \text{ is defined} \\ & \wedge \quad N1 = F(N) \\ & \wedge \quad c' = (N1, u_1 \cdots u_n) \\ \text{then} & \qquad \delta(f_{N,\Gamma}(u_1 \cdots u_n))_{c',\kappa_\Gamma} \\ \text{otherwise} & \qquad \text{undefined} \end{cases}$$

$$\varphi[\![\mathbf{N} :< \mathbf{a_1} \cdots \mathbf{a_n} >]\!]_{\kappa_\Gamma} = \begin{cases} & \wedge \quad \kappa_\Gamma(N) = f_{N,\Gamma} \\ & \wedge \quad f_{N,\Gamma}(a_1 \cdots a_n) \text{ is defined} \\ & \wedge \quad c = (F(N), F(a_1 \cdots a_n)) \\ \text{then} & \qquad \delta(f_{N,\Gamma}(F(a_1 \cdots a_n)))_{c,\kappa_\Gamma} \end{cases}$$

Abbildung 4.3: Die Semantik von Deskriptoren

4.4 Natural Semantics

4.4.1 Begriffe der Natural Semantics

In diesem Abschnitt wird die Semantik eines DATR Programms im Natural Semantics Stil vorgestellt.
Im Natural Semantics Stil besteht die Semantik aus einer Menge von Regeln. Eine Regel hat einen Zähler und einen Nenner. Es können Variablen im Zähler und im Nenner auftreten. Die Variablen beschreiben, wann eine Regel angewendet werden kann. Der Zähler einer Regel besteht aus einer Menge von Formeln. Diese werden Prämisse der Regel genannt. Wenn alle Formeln der Prämisse wahr sind, so gilt der Zähler der Regel, der eine einzige Formel ist. Der Zähler der Regel wird auch Konklusion genannt.
Formeln haben zwei Teile: Folgerungen und Bedingungen. Die Konklusion einer Regel ist immer eine Folgerung. Bedingungen drücken immer eine Einschränkung der Anwendbarkeit der Regel aus. Eine Einschränkung ist zum Beispiel, wenn eine Variable ein bestimmtes Prädikat erfüllen muß , oder wenn eine bestimmte Relation zwischen zwei Variablen vorausgesetzt wird.
Eine Folgerung hat zwei Teile: den Antecedenten und den Konsequenten. Beide werden in den Regeln durch das Symbol $\vdash$ getrennt. Der Konsequent ist ein Prädikat.
Ein Axiom ist eine Regel ohne eine Folgerung in der Prämisse. Ein Axiom kann jedoch eine Bedingung haben.

4.4.2 Die Natural Semantics von DATR

Ein DATR Programm besteht aus einer externen globalen Variablendeklaration, Variablendeklarationen, Satzdeklarationen und einer Anfrage. Die Semantik einer Deklaration ist eine Änderung des Zustands. Der Zustand wird durch die Zustandsvariablen *ex,var,sen,con* beschrieben. Die Regeln (EXVARDECL), (VARDECL) und (SDECL) beschreiben die Semantik der Deklarationen.
Eine Anfrage an ein DATR Programm leitet eine Folge von Reduktionen ein, die einen Knoten und einen Pfad entsprechend einem gegebenen DATR Programm letzendlich zu einer Folge von Atomen reduzieren. Damit ist die Reduktionsfolge abgeschlossen. In den Regeln der Natural Semantics für DATR werden Variablen benutzt. Es sind die syntaktischen Variablen der BNF aus Abschnitt 3.3. Aus Gründen der Übersichtlichkeit sind im folgenden die syntaktischen Variablen nicht in spitzen Klammern eingeschlossen.
In den Regeln ist einigen syntaktischen Variablen ein Index oder ein Hoch-

kommata angefügt. Damit soll deutlich werden, daß eine syntaktische Variable mit verschiedenen Indizes bzw. mit einem Hochkommata für möglicherweise unterschiedliche Folgen von Terminalsymbolen steht. Eine syntaktischen Variable mit einem Stern (also zum Beispiel $constant^*$ oder $varname^*$) soll eine, möglicherweise leere, Liste von Terminalsymbolen bedeuten. Im Beispiel sind das Folgen von Terminalsymbolen die durch *constant* oder *varname* beschrieben werden.
Die leere Liste wird mit Nil bezeichnet. Ein Element wird mit $Cons$ in eine Liste eingetragen. Es werden in den Regeln folgende Definitionen verwendet:

$$\begin{aligned} con &\equiv con[gn], con[gp], con[def], con[binding] \\ ex &\equiv ex[n\_name], ex[node], ex[p\_name], ex[p] \end{aligned}$$

ex und *con* sind also beides 4-Tupel. Intuitiv soll *con* für den wechselnden Kontext stehen, in dem ein Deskriptor ausgewertet wird. Dieser Kontext besteht aus vier Variablennamen: $con[gn]$, $con[gp]$,$con[def]$ und $con[binding]$. Diese Variablennamen bezeichnen den globalen Knoten, den globalen Pfad, den default Pfad und die aktuelle Variablenbindung.
Wenn eine dieser Variablen an einen bestimmten Wert gebunden ist, so wird dieses durch einen Pfeil bezeichnet, zum Beispiel $con[gn] \rightarrow node$, wenn $con[gn]$ an den Wert *node* gebunden wird. *ex* soll intuitiv den externen globalen Kontext bezeichnen. Der externe globale Kontext wird durch den Namen des externen globalen Pfades, den Namen des externen globalen Knotens, den externen globalen Pfad und den externen globalen Knoten beschrieben. Die Elemente des externen globalen Kontexts bestehen aus den Variablennamen $ex[n\_name]$,$ex[node]$,$ex[p\_name]$,$ex[p]$.

4.4.3 Die Regeln

- (CONSTANT)

$$ex, var, sen, con \vdash Cons(constant, Nil) \rhd Cons(constant, Nil)$$

Die Regel (CONSTANT) ist ein Axiom und beschreibt die Auswertung von Deskriptoren, die durch die syntaktischen Variable *constant* beschrieben werden können. Die Auswertung von *constant* in der Umgebung *ex,var,sen,con* ist genau *constant*.

- (NIL)

$$ex, var, sen, con \vdash Nil \rhd Nil$$

- (VARIABLE)

$$\frac{ex, var, sen, con \vdash^{access\_var} varname \rightarrow constant}{ex, var, sen, con \vdash Cons(varname, Nil) \rhd Cons(constant, Nil)}$$

Die Regel VARIABLE beschreibt die Bedeutung eines Atoms,das durch *varname* beschrieben ist. Wenn der Variablenname *varname* an das Atom *constant* gebunden ist , so ist die Bedeutung von *varname* gleich *constant*. Die Regel *access_var* beschreibt das an *varname* gebundene Atom *constant* in der aktuellen Variablenbindung $con[binding]$.
Die syntaktischen Variablen *varname* und *constant* werden in den nächsten Regeln durch v und c abgekürzt.

- (ACCESS_VAR 1)

$$ex, var, sen, con[gn], con[gp], con[def], Nil \vdash^{access\_var} (v \rightarrow c)$$

- (ACCESS_VAR 2)

$$ex, var, sen, con[gn], con[gp], con[def], Cons((v \rightarrow c), con[binding]) \vdash^{access\_var} (v \rightarrow c)$$

- (ACCESS_VAR 3)

$$\frac{ex, var, sen, con[gn], con[gp], con[def], con[binding] \vdash^{access\_var} (v \rightarrow c)}{ex, var, sen, con[gn], con[gp], con[def], Cons((v' \rightarrow c'), con[binding]) \vdash^{access\_var} (v \rightarrow c)}$$

$$\textbf{if } v \neq v'$$

- (GLOBAL NODE 1)

$$\frac{\begin{array}{c} ex, var, sen, con \vdash con[gp] \rightarrow < constant^* > \\ ex, var, sen, con_1 \vdash node :< constant^* > \rhd x \end{array}}{ex, var, sen, con \vdash Cons('node', Nil) \rhd x}$$

$$con_1 \quad = \quad con[gn] \rightarrow node, con[gp], con[def], con[binding]$$

- (GLOBAL PATH 1)
  Der Wert eines globalen Pfades hängt davon ab, ob ein Pfad, ausgewertet zu einer Sequenz von Atomen, gleich dem Namen des externen globalen Pfades ist. Wenn dieses der Fall ist, so wird anschließend der externe globale Pfad ausgewertet. Der auszuwertende Pfad wird um den globalen Knoten ergänzt.

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc \rhd constant_1^* \\ ex, var, sen, con \vdash con[gn] \rightarrow node \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_2^* > \\ ex, var, sen, con_1 \vdash Cons(node :< constant_1^* >, Nil) \rhd x \end{array}}{ex, var, sen, con_1 \vdash Cons('< desc >', Nil) \rhd x},$$

$$\textbf{if } \; constant_1^* \neq constant_2^*$$

$$con_1 \quad = \quad con[gn], con[gp] \rightarrow < constant_1^* >, con[def], con[binding]$$

- (GLOBAL PATH 2)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc \rhd constant^* \\ ex, var, sen, con \vdash con[gn] \rightarrow node \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant^* > \\ ex, var, sen, con_1 \vdash Cons(node :< constant^* >, Nil) \rhd x \end{array}}{ex, var, sen, con_1 \vdash Cons('< desc >', Nil) \rhd x},$$

$$con_1 \quad = \quad con[gn], con[gp] \rightarrow < constant^* >, con[def], con[binding]$$

- (GLOBAL NODE-PATH 1)
  Bei der Auswertung eines globalen Deskriptors, der aus einem Knoten und einem Pfad besteht, können vier Fälle unterschieden werden:
  Der Knoten des Deskriptors ist gleich dem Namen des externen globalen Knotens aber der Pfad ist nicht gleich dem Namen des externen globalen Pfades, der Knoten des Deskriptors ist nicht gleich dem Namen des externen globalen Knotens aber der Pfad ist gleich dem Namen des externen globalen Pfades, der Knoten des Deskriptors ist nicht gleich dem Namen des externen globalen Knotens und der Pfad ist nicht gleich dem Namen des externen globalen Pfades, der Knoten des Deskriptors ist gleich dem Namen des externen globalen Knotens und der Pfad ist

gleich dem Namen des externen globalen Pfades.

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc \triangleright constant_1^* \\ ex, var, sen, con \vdash ex[n\_name] \rightarrow node_2 \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_2^* > \\ ex, var, sen, con_1 \vdash Cons(node_1 :< constant_1^* >, Nil) \triangleright x \end{array}}{ex, var, sen, con \vdash Cons('node_1 :< desc >', Nil) \triangleright x}$$

$$\textbf{if} \quad \begin{array}{ll} \wedge & (node_1 \neq node_2) \\ \wedge & (constant_1^* \neq constant_2^*) \end{array}$$

$$con_1 = con[gn] \rightarrow node, con[gp] \rightarrow < constant_1^* >, con[def], con[binding]$$

- (GLOBAL NODE-PATH 2)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc \triangleright constant_1^* \\ ex, var, sen, con \vdash ex[n\_name] \rightarrow node_1 \\ ex, var, sen, con \vdash ex[node] \rightarrow node_2 \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_2^* > \\ ex, var, sen, con_1 \vdash Cons(node_2 :< constant_1^* >, Nil) \triangleright x \end{array}}{ex, var, sen, con \vdash Cons('node_1 :< desc >', Nil) \triangleright x}$$

$$\textbf{if} \quad (constant_1^* \neq constant_2^*)$$

$$con_1 = con[gn] \rightarrow node, con[gp] \rightarrow < constant_1^* >, con[def], con[binding]$$

- (GLOBAL NODE-PATH 3)

$$\frac{\begin{array}{l} ex, var, sen, con' \vdash desc \rhd constant_1^* \\ ex, var, sen, con \vdash ex[n\_name] \rightarrow node_2 \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_1^* > \\ ex, var, sen, con \vdash ex[p] \rightarrow < constant_2^* > \\ ex, var, sen, con \vdash Cons(node_1 :< constant_2^* >, Nil) \rhd x \end{array}}{ex, var, sen, con_1 \vdash Cons('node_1 :< desc >', Nil) \rhd x}$$

if $(node_1 \neq node_2)$

$$con_1 = con[gn] \rightarrow node, con[gp] \rightarrow < constant_1^* >, con[def], con[binding]$$

- (GLOBAL NODE-PATH 4)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc \rhd constant_1^* \\ ex, var, sen, con \vdash ex[n\_name] \rightarrow node_1 \\ ex, var, sen, con \vdash ex[node] \rightarrow node_2 \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_1^* > \\ ex, var, sen, con \vdash ex[p] \rightarrow < constant_2^* > \\ ex, var, sen, con_1 \vdash Cons(node_2 :< constant_2^* >, Nil) \rhd x \end{array}}{ex, var, sen, con_1 \vdash Cons('node_1 :< desc >', Nil) \rhd x}$$

$$con_1 = con[gn] \rightarrow node_2, con[gp] \rightarrow < constant_2^* >, con[def], con[binding]$$

- (SEQUENCE)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc_1 \rhd constant_1^* \\ ex, var, sen, con \vdash desc_2 \rhd constant_2^* \\ \vdash^{concat} desc_1, desc_2, desc_3 \\ \vdash^{concat} constant_1^*, constant_2^*, constant_3^* \end{array}}{ex, var, sen, con \vdash desc_3 \rhd constant_3^*}$$

Die Semantik einer Sequenz von Deskriptoren ist die Sequenz, die sich durch Auswerten der einzelnen Deskriptoren der Sequenz ergibt.

- (LOCAL 1)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash desc_1 \rhd constant_1^* \\ ex, var, sen, con \vdash con[def] \rightarrow < constant_2^* > \\ \vdash^{concat} constant_1^*, constant_2^*, constant_3^* \\ ex, var, sen, con \vdash^{search} (node_1 :< constant_3^* >) \rightarrow (node_1 :< atom_1^* >= desc_2, b) \\ ex, var, sen, con \vdash ex[n\_name] \rightarrow node_2 \\ ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_5^* > \\ \vdash^{cut} atom_1^*, constant_3^*, constant_4^* \\ ex, var, sen, con' \vdash desc_2 \rhd x \end{array}}{ex, var, sen, con \vdash node_1 :< desc_1 > \rhd x},$$

$$\textbf{if} \quad \begin{array}{ll} \wedge & (node_1 \neq node_2) \\ \wedge & (constant_1^* \neq constant_5^*) \end{array}$$

$$con' \quad = \quad con[gn], con[gp], con'[def] \rightarrow < constant_4^* >, con'[binding] \rightarrow b$$

- (LOCAL 2)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash ex[n\_name] \rightarrow node_1 \\ ex, var, sen, con \vdash ex[node] \rightarrow node_2 \\ ex, var, sen, con \vdash node_2 :< constant^* > \rhd x \end{array}}{ex, var, sen, con \vdash node_1 :< constant^* > \rhd x}$$

  Wenn der Knoten $node_1$ des Anfragepfades $node_1 :< constant^* >$ gleich dem Namen des externen globalen Knotens ist, so wird der Knoten $node_1$ durch den externen globalen Knoten $node_2$ ersetzt.

- (LOCAL 3)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash ex[p\_name] \rightarrow < constant_1^* > \\ ex, var, sen, con \vdash ex[p] \rightarrow < constant_2^* > \\ ex, var, sen, con \vdash node :< constant_2^* > \rhd x \end{array}}{ex, var, sen, con \vdash node :< constant_1^* > \rhd x}$$

  Wenn der lokale Pfad $< constant_1^* >$ gleich dem Namen des externen globalen Pfades ist, so wird $< constant_1^* >$ durch den externen globalen Pfad ersetzt.

- (EXVARDECL)

$$ex, var, sen, con \vdash \texttt{qnode:}\ node\ \texttt{qpath:}\ < constant^* > \rhd ex_1, var, sen, con$$

$$ex_1[n\_name] \rightarrow node$$
$$ex_1[p\_name] \rightarrow < constant^* >$$

Eine externe globale Variablendeklaration definiert den Namen des externen globalen Knotens und den Namen des externen globalen Pfades.

- (VARDECL)

$$ex, var, sen, con \vdash \ \texttt{vars:} \quad varname\ constant^*$$
$$\rhd ex, Cons((varname, constant^*), var), sen, con$$

Eine Variablendeklaration erweitert die Liste *var* der Variablendeklarationen um ein Element.

- (SDECL)

$$ex, var, sen, con \vdash node :< atom^* >= desc \rhd$$
$$ex, var, Cons((node :< atom^* >= desc), sen), con$$

Eine Satzdeklaration verlängert die Liste der Satzdeklarationen *sen* um ein Element.

- (QUERY)

$$\frac{ex_1, var, sen, con_1 \vdash node : \texttt{<}desc^*\texttt{>} \rhd x}{ex, var, sen, con \vdash \texttt{query:}\ node\ :\ \texttt{<}\ desc^*\ \texttt{>}}$$

$$\begin{array}{l} ex_1[n\_name] \to node \\ ex_1[p\_name] \to desc^* \\ con_1[gn] \to node \\ con_1[gp] \to desc^* \\ con_1[def] \to Nil \\ con_1[binding] \to Nil \end{array}$$

Eine Anfrage an ein DATR Programm verändert den globalen Knoten, den globalen Pfad, den default Pfad, die aktuelle Variablenbindung sowie den externen globalen Knoten und den externen globalen Pfad. Die Anfrage, bestehend aus einem Knoten und einem Pfad, wird in dieser initialisierten Umgebung ausgewertet.

- (COMP)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash D_1 \rhd ex', var', sen', con' \\ ex', var', sen', con' \vdash D_2 \end{array}}{ex, var, sen, con \vdash D_1 D_2}$$

Die Regel (COMP) beschreibt die Hintereinanderausführung von Deklarationen D_1, D_2. D_1 ist eine Deklaration, zum Beispiel eine externen globale Variablendeklaration, eine Variablendeklaration oder eine Satzdeklaration. D_2 ist entweder eine Anfrage oder eine Sequenz von Deklarationen gefolgt von einer Anfrage.

- (PREFIX 1)

$$\frac{ex, var, sen, con \vdash^{prefix} (tail(constant^*), tail(atom^*)) \rhd b}{ex, var, sen, con \vdash^{prefix} (constant^*, atom^*) \rhd b}$$

$$\begin{array}{llll} & \wedge & head(constant^*) & = & head(atom^*) \\ \textbf{if} & \wedge & constant^* & \neq & Nil \\ & \wedge & atom^* & \neq & Nil \end{array}$$

Die Regeln PREFIX beschreiben, ob die Sequenz *as* ein Präfix von *bs* ist.

- (PREFIX 2)

$$\frac{\begin{array}{l} ex, var, sen, con' \vdash^{prefix} (tail(as), tail(bs)) \rhd b' \\ ex, var, sen, con \vdash^{legal} (head(as), head(bs), var, con[binding]) \end{array}}{ex, var, sen, con \vdash^{prefix} (as, bs) \rhd b}$$

$$\textbf{if} \quad \begin{array}{ll} \wedge & head(as) = constant \\ \wedge & head(bs) = variable \\ \wedge & as \neq Nil \\ \wedge & bs \neq Nil \\ \wedge & b' = Cons((constant, variable), b) \end{array}$$

$$\begin{array}{lcl} con[binding]' & = & Cons((head(as), head(bs)), con[binding]) \\ con' & = & con[gn], con[gp], con[def], con[binding]' \end{array}$$

- (PREFIX 3)

$$ex, var, sen, con \vdash^{prefix} (as, bs) \rhd b$$

$$\textbf{if} \quad (as = Nil)$$

- (SEARCH 1)

$$\frac{\begin{array}{l} ex, var, sen, con \vdash^{search} node :< constant_1^* >\to (node :< atom_2^* >= desc_3, b_3) \\ ex, var, sen, con' \vdash^{prefix} (atom_2^*, atom_1^*) \rhd b_2 \\ ex, var, sen, con \vdash^{prefix} (atom_1^*, constant_1^*) \rhd b_3 \end{array}}{\begin{array}{l} ex, var, Cons((node :< atom_1^* >= desc_1), sen), con \vdash^{search} \\ node :< constant_1^* >\to (node, < atom_2^* >= desc_3, b_1) \end{array}}$$

$$\frac{ex, var, sen, con \vdash^{search} node_2 :< constant_2^* >\to (node_2, < constant_3^* >= desc_3, b)}{\begin{array}{l} ex, var, Cons((node_1, < constant_1^* >, desc_1), sen), con \vdash^{search} \\ node_2 :< constant_2^* >\to (node_2, < constant_3^* >= desc_3, b) \end{array}}$$

, **otherwise**

- (SEARCH 2)

$$ex, var, Nil, con \vdash^{search} node :< constant_1^* > \rightarrow (node, < constant_2^* >, desc, Nil)$$

- (CUT 1)

$$\vdash^{cut} Nil, xs \rhd xs$$

- (CUT 2)

$$\frac{\vdash^{cut} as_1, as_2 \rhd xs}{\vdash^{cut} Cons(a_1, as_1), Cons(a_2, as_2) \rhd xs}$$

- (CONCAT 1)

$$\frac{\vdash^{concat} as_1, as_2, as_3}{\vdash^{concat} Cons(a_1, as_1), as_2, Cons(a_1, as_3)}$$

- (CONCAT 2)

$$\vdash^{concat} Nil, as, as$$

- (LEGAL)

$$\frac{\begin{array}{c}\vdash^{access} varname, constant, binding \\ \vdash^{inrange} varname, constant, vd\end{array}}{\vdash^{legal} varname, constant, vd, binding}$$

- (INRANGE 1)

$$\frac{\vdash^{isin} constant, atom^*}{\vdash^{inrange} varname, constant, Cons((varname, atom^*), vps)}$$

- (INRANGE 2)

$$\frac{\vdash^{inrange} varname_1, constant, vps}{\vdash^{inrange} varname_1, constant, Cons((varname_2, atom^*), vps)}$$

if $(varname_1 \neq varname_2)$

- (ISIN 1)

$$\vdash^{isin} constant_1, Cons((constant_1), cs)$$

- (ISIN 2)

$$\frac{\vdash^{isin} constant_1, cs}{\vdash^{isin} constant_1, Cons((constant_2), cs)}$$

if $(constant_1 \neq constant_2)$

- (BOUND 1)

$$\vdash^{bound} varname, constant, Cons((varname, constant), bs)$$

- (BOUND 2)

$$\frac{\vdash^{bound} varname_1, constant_1, bs}{\vdash^{bound} varname_1, constant_1, Cons((varname_2, constant_2), bs)}$$

if $(varname_1 \neq varname_2)$

4.4.4 Funktionen

$$\begin{array}{lcl} head(xs) & = & xs[1] \\ tail(xs) & = & [i \in 1..(Len(s)-1) \mapsto s[i+1]] \\ Len(s) & = & [s[1] \cdots s[n] \mapsto n] \end{array}$$

4.5 Beispiel für eine Ableitung

Als ein Beispiel für die Anwendung der Regeln soll der Mealy-Automat (Abbildung 4.4) dienen. Der Mealy-Automat soll die Zeichenfolge $(ab)^*$ erkennen.
Im folgenden wird abgeleitet, wie ein DATR-Programm als Mealy Automat

$S1$	Startzustand
$\{S1, S2, ERR\}$	Menge der Zustände
$\{a, b\}$	Eingabe-Alphabet
$\{error, ok, \epsilon\}$	Ausgabealphabet
δ	Übergangsfunktion
$\delta(S1, a)$	$= (S2, \epsilon)$
$\delta(S1, b)$	$= (ERR, error)$
$\delta(S2, a)$	$= (ERR, error)$
$\delta(S2, b)$	$= (S2, \epsilon\ )$
$\delta(ERR, a)$	$= (ERR, \epsilon)$
$\delta(ERR, b)$	$= (ERR, \epsilon)$

Abbildung 4.4: Ein Mealy Automat

```
S1:<>   = ok.
S1:<a>  = S2:<>.
S1:<b>  = error ERR:<>.
S2:<>   = error ERR:<>.
S2:<a>  = error ERR:<>.
S2:<b>  = S1:<>.
ERR:<a> = ERR:<>.
ERR:<b> = ERR:<>.
```

Abbildung 4.5: Ein DATR Programm als Mealy-Automat

(Abbildung 4.5) die Eingabe ab erkennt.
Um die Ableitung auf den interessanteren Teil einzuschränken, wird davon ausgegangen, daß die Umgebungsvariablen ex, var, sen, con durch Anwendung der Regeln (EXVARDECL),(VARDECL),(SDECL),(QUERY) und (COMP) auf das DATR Programm definiert sind.

- **Step 1**
$ex, var, sen, con_1 \vdash Cons(S1 :< Cons(a, Cons(b, Nil)) >, Nil) \triangleright Cons(ok, Nil)$

- **Step 1.1**
  $ex, var, sen, \vdash Cons(a, Cons(b, Nil)) \triangleright Cons(a, Cons(b, Nil))$

- **Step 1.2**
  $\vdash^{concat} Cons(a, Cons(b, Nil)), Nil, Cons(a, Cons(b, Nil))$

- **Step 1.3**
  $ex, var, sen, con \vdash^{search} S1 :< Cons(a, Cons(b, Nil)) > \rightarrow (S1 :< Cons(a, Nil) >$
  $, S2 :< Nil >, Nil)$

- **Step 1.4**
  $\vdash^{cut} Cons(a, Nil), Cons(a, Cons(b, Nil)), Cons(b, Nil)$

- **Step 1.5**
  $ex, var, sen, con' \vdash Cons(S2 :< Nil >, Nil) \triangleright Cons(ok, Nil)$

- **Step 1.6**
  Step 1 folgt aus **Step 1.1** - **Step 1.5** durch Anwenden der Regel (LOCAL 1).

- **Step 1.1.1**
  $ex, var, sen, con \vdash Cons(a, Nil) \triangleright Cons(a, Nil)$
  Folgt direkt aus (CONSTANT).

- **Step 1.1.2**
  $ex, var, sen, con \vdash Cons(b, Nil) \triangleright Cons(b, Nil)$
  Folgt direkt aus (CONSTANT).

- **Step 1.1.3**
  $\vdash^{concat} Cons(a, Nil), Cons(b, Nil), Cons(a, Cons(b, Nil))$

- **Step 1.1.4**
  $\vdash^{concat} Cons(a, Nil), Cons(b, Nil), Cons(a, Cons(b, Nil))$

- **Step 1.1.5**
  Step 1.1 folgt aus **Step 1.1.1-Step 1.1.4** durch SEQUENCE .

- **Step 1.2.1**
  $\vdash^{concat} Cons(b, Nil), Nil, Cons(b, Nil)$
  Folgt aus **Step 1.2** durch Anwenden von (CONCAT 1).

- **Step 1.2.2**
  $\vdash^{concat} Nil, Nil, Nil$
  Folgt durch Anwenden von (CONCAT 2).

- **Step 1.2.3**
  Step 1.2 folgt aus **Step 1.2.1** und **Step 1.2.2** durch Anwenden von (CONCAT 1) und (CONCAT 2).

- **Step 1.3.1**
  $ex, var, Cons((ERR :< Cons(b, Nil) >= Cons(ERR :< Nil >, Nil)), sen_1), con_1 \vdash^{search} (S_1 :< Cons(a, Cons(b, Nil)) >) \to (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.3.2**
  $ex, var, Cons((ERR :< Cons(a, Nil) >= Cons(ERR :< Nil >, Nil)), sen_2), con \vdash^{search} (S_1 :< Cons(a, Cons(b, Nil) >) \to (S_1 :< Cons(a, Nil) >, Cons(S2 =< Nil >, Nil), Nil)$

- **Step 1.3.3**
  $ex, var, Cons((S2 :< Cons(b, Nil) >= Cons(S2 :< Nil >, Nil)), sen_3), con \vdash^{search} (S_1 :< Cons(a, Cons(b), Nil) >) \to (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.3.4**
  $ex, var, Cons(S2 :< Cons(a, Nil) >= Cons(error, Cons(ERR :< Nil >, Nil)), sen_4), con \vdash^{search} (S_1 :< Cons(a, Cons(b, Nil)) >) \to (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.3.5**
  $ex, var, Cons((S2 :< Nil >= Cons(error, Cons(ERR :< Nil >, Nil)), sen_5), con \vdash^{search} (S_1 :< Cons(a, Cons(b), Nil) >) \to (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.3.6**
  $ex, var, Cons((S1 :< Cons(b, Nil) >= Cons(S2 :< Nil >, Nil)), sen_6), con \vdash^{search} (S_1 :< Cons(a, Cons(b, Nil)) >) \to (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.3.7**
$ex, var, Cons((S1 :< Cons(a, Nil) >= Cons(S1 :< Nil >, Nil)), sen_7), con \vdash^{search} (S_1 :< Cons(a, Cons(b, Nil)) >) \rightarrow (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil$

- **Step 1.3.8**
$ex, var, Cons((S1 :< Nil >= Cons(ok, Nil)), Nil), con \vdash^{search} (S_1 :< Cons(a, Cons(b, Nil)) >) \rightarrow (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.3.9**
$ex, var, Nil, con \vdash^{search} (S_1, < Cons(a, Cons(b, Nil)) >) \rightarrow (S_1 :< Cons(a, Nil) >= Cons(S2 :< Nil >, Nil), Nil)$

- **Step 1.4.1**
$\vdash^{cut} Nil, Cons(b, Nil), Cons(b, Nil)$
Folgt durch (CUT 1) aus **Step 1.4**.

- **Step 1.4.2**
Step 1.4 folgt aus (CUT 2) mit **Step 1.4.1**.

Die weiteren Schritte verlaufen entprechend.

4.6 Der Theorembeweiser Isabelle zur Ausführung von DATR

4.6.1 Nutzung von Isabelle zur Ausführung von DATR

Mit Hilfe eines Theorembeweisers kann eine formale Semantik in Logik direkt ausgeführt werden. Damit kann aus einer Semantik ein Interpreter oder ein Compiler gewonnen werden. In dieser Arbeit wurde der Theorembeweiser Isabelle verwendet, um ein DATR Programm auszuführen.
Isabelle hat verschiedene vordefinierte Logiken, zum Beispiel ZF (Zermelo-Fraenkel Set Theory), FOL (first order logic with natural deduction) und HOL (higher order logic of Church).
Neue Logiken können als Ergänzung einer bestehenden Logik definiert werden. Dadurch ist eine Form von modularen Aufbau einer Logik möglich.
Die Abbildung 4.9 zeigt, wie der Theorembeweiser Isabelle benutzt wurde, um ein DATR Programm auszuführen.
In der Datei *datr.thy* sind die Signatur und die Axiome von DATR gespeichert. Diese definieren die Semantik von DATR und werden von dem Theorembeweiser benutzt, um eine Anfrage an ein Programm zu einer Liste von Atomen zu reduzieren. Der Theorembeweiser Isabelle arbeitet nicht vollautomatisch bei der Ableitung oder beim Beweisen von Theoremen, sondern es muß eine Suchstrategie definiert werden.
Diese wird durch *tacticals* festgelegt. *tacticals* definieren wie *tactics* kombiniert werden sollen, um einen Beweis zu führen. Beispiele für *tactics* sind *resolution tactic* und *assumption tactic*. Mit diesen kann ein Beweis durch Resolution oder ein Beweis durch Annahme geführt werden. In der Datei *tacticals* ist definiert, welche *tacticals* der Theorembeweiser benutzen soll, um ein DATR Programm auszuführen. In dieser Arbeit wurde dazu eine Kombination von Tiefensuche und Resolution benutzt. Da die Datei *tacticals* aus ML Code besteht, kann ein beliebiger Suchalgorithmus verwendet werden. Damit kann der verwendete Suchalgorithmus optimal auf ein Problem abgestimmt werden, um eine möglichst effiziente Ausführung des Beweisens eines Theorems zu erreichen.
Bei der Benutzung von Isabelle zur Ausführung von DATR ist zwischen einer internen und einer externen Repräsentation eines DATR Programms zu unterscheiden. Die externe Repräsentation ist die Darstellung des Programms in der Syntax, die in den Abschnitten 3.2 und 3.3 vorgestellt wurde. Die interne Darstellung ist ein Abstrakter Syntaxbaum. Isabelle stellt einige Operationen zur Verfügung, um eine Eingabe in externer Repräsentation in eine interne Repräsentation zu wandeln. Aber es gibt hierbei Beschränkungen, beliebi-

ge Wandlungen sind nicht möglich. Die Operationen erscheinen umständlich und kompliziert für die Umwandlung eines größeren Programms. In DATR können einige Zeichen verwendet werden, die in Isabelle reservierte Symbole sind. Ein Beispiel sind Variablen in DATR. Sie beginnen in der konkreten Syntax mit dem Zeichen '$ '. In Isabelle muß jedoch jeder Identifier mit einem Buchstaben beginnen. Deshalb ist die direkte Eingabe von Variablen in konkreter DATR Syntax in den Theorembeweiser Isabelle nicht möglich. In dieser Arbeit wurde ein kleines Tool geschrieben, das ein DATR Programm in abstrakter Syntax in eine für Isabelle lesbare Form umwandelt. Dieses Tool wandelt also eine externe Repräsentation eines DATR Programms in die interne Repräsentation von Isabelle um. In der Abbildung 4.9 stellt die Datei *prog* die interne Repräsentation dar.
Im folgenden ist ein Beispiel für eine interne und eine externe Repräsentation eines DATR Programms gegeben:

```
Elephant: <color>      == grey.
Elephant: <legs>       == four.

RoyalElephant: <>      == Elephant:<>.
RoyalElephant: <color> == white.

Clyde: <>              == RoyalElephant:<>.

Tom: <>                == Elephant:<>.
Tom: <legs>            == three.
```

Abbildung 4.6: Ein Programm in der externen Repräsentation

Eine Liste ist in der internen Repräsentation über die leere Liste Nil und den Konstruktor Cons definiert. Satzdeklarationen, Variablendeklarationen und Deskriptoren werden als Listen dargestellt. Die Funktionen Exdecl,Sen,Const usw. beschreiben den Typ eines Datums (externen Variablendeklaration,Satzdeklaration, Deskriptor Const usw.)

```
goal D.thy "Query(Nil,Cons(Sen(Elephant,Cons(Const(color),Nil),
Cons(Const(grey),Nil)),Cons(Sen(Elephant,Cons(Const(legs),Nil),
Cons(Const(four),Nil)),Cons(Sen(RoyalElephant,Nil,
Cons(Local(Elephant,Nil),Nil)),Cons(Sen(RoyalElephant,
Cons(Const(color),Nil),Cons(Const(white),Nil)),Cons(Sen(Clyde,Nil,
Cons(Local(RoyalElephant,Nil),Nil)),Cons(Sen(Tom,Nil,
Cons(Local(Elephant,Nil),Nil)),Cons(Sen(Tom,Cons(Const(legs),Nil),
Cons(Const(three),Nil)),Nil))))))),Con(Elephant,
Cons(Const(color),Nil),Nil,Nil),Exdecl(QNODE,Elephant,
Cons(Const(qpath),Nil),Cons(Const(color),Nil)),
Cons(Local(Elephant,Cons(Const(color),Nil)),Nil),?x)"
```

Abbildung 4.7: Ein Programm in interner Repräsentation

4.6.2 Das HTML User Interface zu DATR für Isabelle

Um die Ausführung von DATR Programmen mit dem Theorembeweiser Isabelle zu veranschaulichen und den direkten Umgang mit Isabelle zu vermeiden, der ML-Kenntnisse voraussetzt, wurde ein HTML Interface entwickelt. In einem Menü sind vier DATR Programme vorgegeben. Zwei davon sind leichte Abwandlungen von Testprogrammen, die Gerald Gazdar entwickelt hat. In dem Programm *matrix* werden verschachtelte globale Deskriptoren und der default Mechanismus verwendet. Die richtige Auswertung von Anfragen an dieses Programm ist ein Indiz für die Richtigkeit der DATR Darstellung in Isabelle.
In dem HTML User Interface kann in einem Menü ein DATR Programm ausgewählt werden. Bei den vier vorgegebenen Programm kann der Code der internen und externen Repräsentation angezeigt werden. Es ist auch möglich den absoluten Pfad eines Programms anzugeben, das ausgeführt werden soll. Dieses Programm muß in abstrakter Syntax vorliegen. Neben der Auswahl des Programms muß eine Anfrage zu dem ausgewählten Programm formuliert werden. Mit dem Button *Permit* beginnt die Auswertung. Es wird ein CGI - Script in Perl gestartet. Dieses kopiert das ausgewählte Programm in externer Repräsentation und die Anfrage in eine Datei und ruft ein Tool auf, das die interne Repräsentation des Programms generiert. Danach wird der Theorembeweiser gestartet. In den Theorembeweiser wird die Datei mit den DATR Axiomen und der Signatur geladen. Außerdem wird die Datei *tacticals* geladen und die interne Repräsentation des ausgewählten Programms. Dann beginnt die Auswertung. Bei Beendigung der Auswertung wird eine HTML Seite generiert, die das vom Theorembeweiser gefundene Ergebnis darstellt.

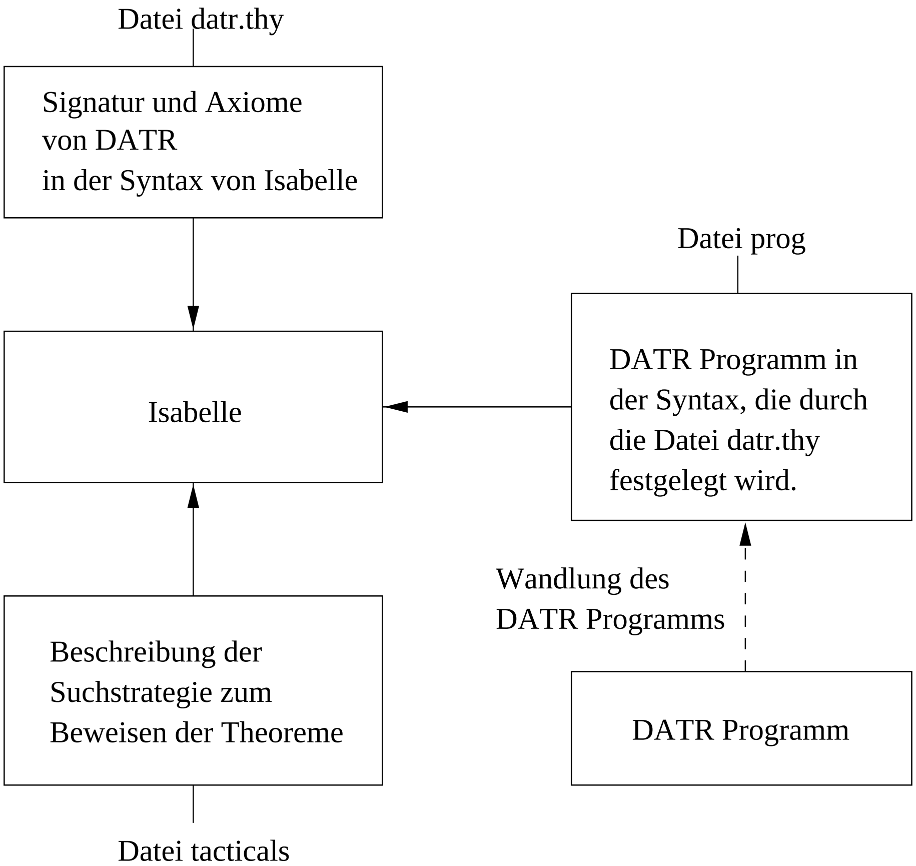

Abbildung 4.8: Der Theorembeweiser Isabelle zur Ausführung von DATR

Diese Seite ist die direkte Ausgabe des Theorembeweisers. Sie besteht aus der Anfrage und dem Ergebnis der Anfrage in abstrakter Syntax. Die unbekannte $?x$ der Anfrage ist hierbei in dem Ergebnis durch eine Liste von Atomen ersetzt.

4.6.3 Einschränkungen

Es werden nur DATR Programme akzeptiert, die der abstrakten Syntax gemäß Abschnitt 3.3 entsprechen mit der Ausnahme, daß Atome nicht mit einer Ziffer beginnen dürfen. Diese Einschränkung gilt, weil in Isabelle alle Identifier mit einem Buchstaben beginnen müssen. Um Atome mit Ziffern zuzulassen, müßten für die Eingabe in den Beweiser aus solchen Atomen neue Atome generiert werden. Damit die neu generierten Atome nicht schon vorhandenen gleichen, müßte weiterhin eine Symboltabelle angelegt werden.

Auf diese wurde verzichtet, da der Beweiser in erster Linie benutzt wurde, um zu veranschaulichen, wie eine in Logik definierte Semantik von DATR auf Plausibilität überprüft werden kann.
Bei der Definition der Axiome von DATR in Isabelle mußten einige Axiome eingeführt werden, die einzig und allein dazu dienen, daß die Unifikation vom Beweiser richtig ausgeführt wird. Diese *Hack* widerspricht zwar voll und ganz dem Wesen einer Semantik, war aber für die praktische Ausführung notwendig. Zu diesen Axiomen gehören: CHANGEDEFx,CHANGEGPx,DESCx und CONCAT3x und PREFIX2x.

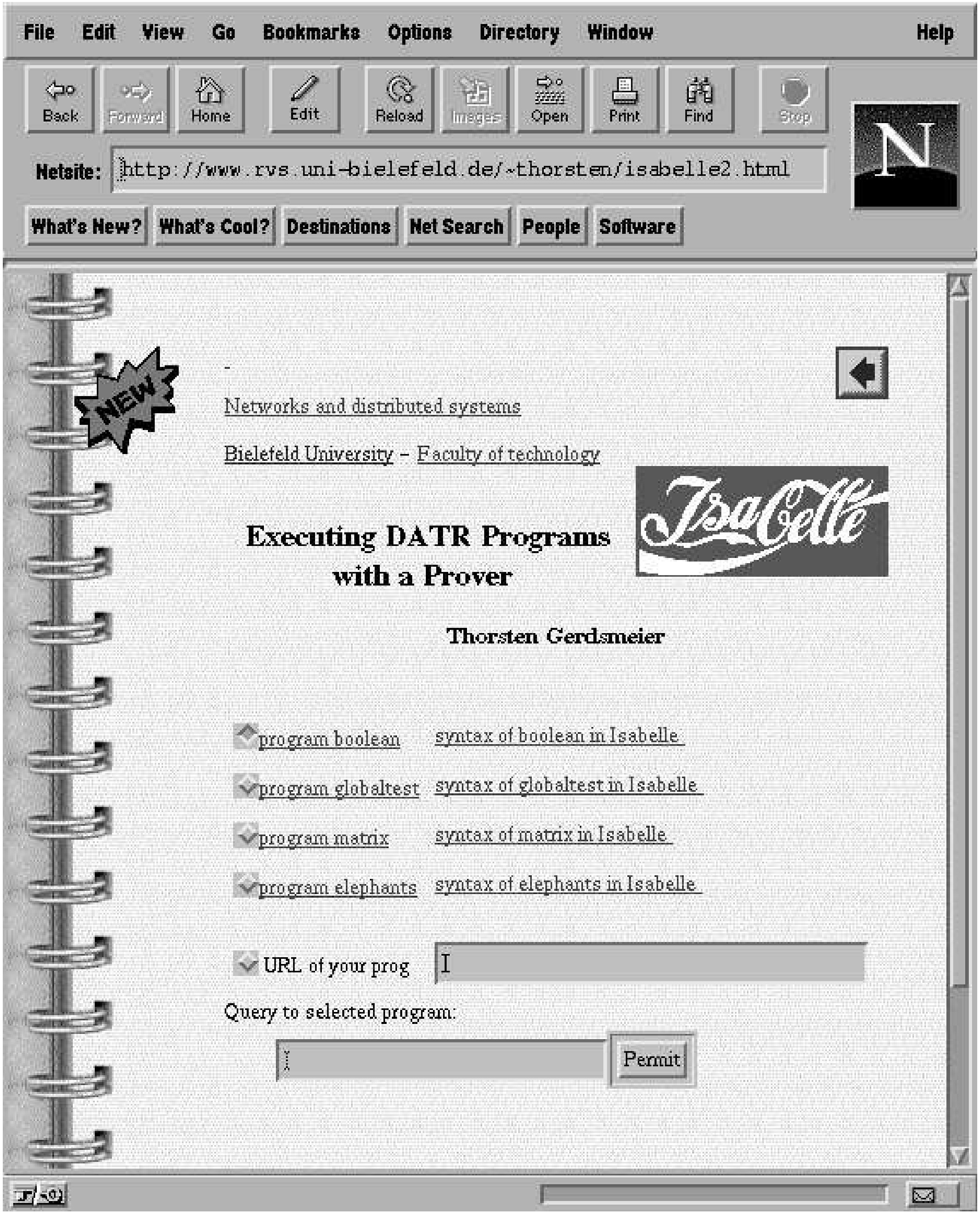

Abbildung 4.9: HTML User Interface für die Benutzung von DATR in Isabelle

Literaturverzeichnis

[1] R.J. Brachman. *What's in a Concept? Structural Foundations for Semantic Networks.*
Bolt,Beranck and Newman, Cambridge MA 1977

[2] Steven E. Brenner and Edwin Aoki. *Introduction to CGI/Perl : getting started with web scripts.*
M & T Books.

[3] Curry,H.B. , R.Feys und W. Craig. *Combinatory Logic, Vol. I, North Holland (1958)*

[4] Deryck F. Browm, Hermano Moura, David A. Watt. *ACTRESS: An Action Semantics Directed Compiler*
Workshop on Compiler Construction, Paderborn, Germany 1992

[5] Arie van Deursen,Peter D. Mosses. *ASD: The Action Semantic Description Tools,Users Guide.*

[6] Roger Evans, Gerald Gazdar. *DATR: A Language for Lexical Knowledge Representation*
Cognitive Science Research Paper,University of Brighton,1995

[7] Roger Evans, Gerald Gazdar. *The DATR Papers: February 1990*
Cognitive Science Research Paper, Serial No CSRP 139, University of Brighton

[8] Roger Evans, Gerald Gazdar. *Inference in DATR*
School of Cognitive and Computing Sciences, University of Sussex

[9] Roger Evans, Gerald Gazdar. *Semantics of DATR*
School of Cognitive and Computing Sciences, University of Sussex,

[10] Elfriede Fehr. *Semantik von Programmiersprachen*
Springer Verlag 1989

[11] Thorsten Gerdsmeier. *A Tool For Building and Analysing WB-graphs* March 1997, http://www.rvs.uni-bielefeld.de/CaliDATR.html

[12] T.Gerdsmeier,P.B. Ladkin and K.Loer. *Analysing the Cali Accident With a WB-Graph* http://www.rvs.uni-bielefeld.de Publications, January 1996. Also to appear in the Proceedings of the Glasgow Workshop on Human Error and System Development, March 1996.

[13] Dafydd Gibbon. *Generalised DATR inference for lexicon development and interfacing* Fakultät für Linguistik und Literaturwissenschaft, Universität Bielefeld,1993

[14] Hagen Langer, Dafydd Gibbon. *DATR as a graph representation language for ILEX speech oriented lexica* Fakultät für Linguistik und Literaturwissenschaft, Universität Bielefeld,1992

[15] Dafydd Gibbon. *Formal DATR application to bubble sort* January 1997,http://coral.lili.uni-bielefeld.de/DATR/

[16] Dafydd Gibbon. *DATR syntax : BNF* http://www.coral.lili.uni-bielefeld.de/DATR/datrinfo/

[17] Dafydd Gibbon. *Prolog Quellcode als DATR Semantik* http://www.coral.lili.uni-bielefeld.de/DATR/datrinfo/

[18] Helmut Herold, Werner Unger. *C Gesamtwerk* te-wi, 10. Auflage 1993

[19] Kyung Goo-Doh . *Action Semantics : A Tool for Developing Programming Languages* Technical Report, University of Aizu 1993

[20] G. Kahn. *Natural Semantics* in 4th Annual Symposium on Theoretical Aspects of Computer Science Springer Verlag 1987, S.23 - S.38

[21] Bill Keller. *DATR Theories and DATR Models* School of Cognitive and Computing Sciences, University of Sussex

[22] Peter B. Ladkin. *Lazy Cache Implements Complete Cache* Universität Bielefeld, Technische Fakultät, 1996 http://rvs.uni-bielefeld.de/ ladkin/abstracts.html#TLA

[23] Peter B. Ladkin. *Formal But Lively Buffers in TLA+*
Universität Bielefeld, Technische Fakultät, 1996
http://rvs.uni-bielefeld.de/ ladkin/abstracts.html#TLA

[24] Leslie Lamport. *Introduction to TLA* 1994

[25] Leslie Lamport. *TLA+* 1995

[26] Leslie Lamport. *The Temporal Logic of Actions*
SRC Research Report 79 ,1993

[27] Hagen Langer. *DATR Without Nodes and Global Inheritance*
Universität Bielefeld, Oktober 1993

[28] D. Lester. *The correctness of a G-machine compiler.*
MSc dissertation, Programming Research Group, Oxford 1985

[29] John R. Levine, Tony Mason & Doug Brown. *lex & yacc*
O'Reilly & Associates, Inc

[30] P.D. Mosses. *Compiler Generation Using Denotational Semantics*
Mathematical Foundations of Computer Science in Gdansk, Lecture Notes in Computer Science No.45,Springer Verlag (1976),436-441

[31] Peter D. Mosses, Martin Musicante. *An Action Semantics for ML Concurrency Primitives*
BRICS Report Series 94-20, 1994

[32] Peter D. Mosses. *A Tutorial on Action Semantics*
Notes for FME'94

[33] Hermano Moura. *An Implementation of Action Semantics*
Department of Computing Science,University of Glasgow

[34] Hermano Moura. *Action Notation Transformations*
Ph.D.Thesis 1993, University of Glasgow

[35] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*
Wiley Professional Computing,1992

[36] Peter Orbaek. *Analysis and Optimization of Actions*
Dissertation September 1993

[37] L.C. Paulson. *ML for the Working Programmer*
Cambridge University Press 1991

[38] L. C. Paulson. *The Isabelle Reference Manual*
Computer Laboratory, University of Cambridge 1997

[39] L.C. Paulson. *Introduction to Isabelle*
Computer Laboratory, University of Cambridge 1997

[40] L.C. Paulson. *Isabelle's Object Logics*
Computer Laboratory, University of Cambridge 1997

[41] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*
Prentice Hall,1987

[42] David A. Schmidt. *Denotational Semantics, A Methodology for Language Development*
Wm. C. Brown Publishers 1988

[43] David S. Touretzky. *The Mathematics of Inheritance Systems*
Morgan Kaufmann Publishers,Inc.,Los Altos,California

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

[45] David A. Watt *Programming Language Syntax and Semantics*
Prentice Hall,1991

[46] Ake Wikstroem. *Functional Programming Using Standard ML*
Prentice Hall 1987

[47] W.A. Woods. *What's in a Link? Foundations for Semantic Networks*
in B. Bobrow, A. Collins: Representation and Understanding, Academic Press, New York 1975, S35-82.