INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Wagmüllerstraße 23 D–80538 München

Ludwig
Maximilians–
Universität
München

Ein Regelinterpretierer zur Vorbereitung eines Programmierpraktikums

Mathias Kettner

Fortgeschrittenenpraktikum

Beginn der Arbeit:	1. August 1995
Abgabe der Arbeit:	15. Februar 1996
Betreuer:	Dr. Norbert Eisinger
	Dipl.-Inform. Tim Geisler
	Dipl.-Inform. Sven Panne

Inhaltsverzeichnis

1 Ziel der Arbeit

Im Wintersemeter 1995/96 wird im Grundstudium ein Programmierpraktikum angeboten. Die Teilnehmer sollen ein größeres Programm entwerfen, spezifizieren und implementieren und dabei grundlegende Techniken der Informatik kennenlernen und anwenden. Thema dieses Praktikums ist die Implementation eines Regelinterpreters in der Art von SATCHMO [MB88]. Als Programmiersprache soll eine funktionale Teilsprache von SCHEME [CH91] verwendet werden.

Das Ziel meines Fortgeschrittenen-Praktikums ist die Entwicklung eines solchen Regelinterpretierers, der im Idealfall als Musterlösung für das Praktikum dienen könnte. Der Programmierstil ist funktional. Das Programm muß sauber strukturiert sein und in einzelne voneinander unabhängige Teile zerlegt werden können, deren Erstellung den Studenten als Teilaufgaben gestellt werden können.

Der Regelinterpretierer soll eine Sprache der Prädikatenlogik erster Stufe verstehen und mittels eines vorwärtsschließenden Kalküls bereichsbeschränkte Regeln[1] mit disjunktiven Köpfen verarbeiten können. Eine Optimierung der Datenbasis durch eine Baumstruktur wird den Studenten nicht als Aufgabe gestellt.

2 Beschreibung des Programmes

Für die Vorbereitung des Praktikums wurden verschiedene Entwicklungsschritte des Programmes festgehalten, deren Ergebnis eine nach Mächtigkeit gestaffelte Reihe von Beweisern ist. Die erste lauffähige Version verarbeitet nur variablenfreie Regeln ohne disjunktive Köpfe. Im nächsten Schritt werden zusätzlich Variablen eingeführt, was die Behandlung von Pattern-Matching und Substitutionen erfordert. Im dritten Schritt werden auch Regeln mit disjunktiven Köpfen zugelassen. In meiner Programmbeschreibung werde ich mich aus Gründen der Übersichtlichkeit lediglich mit der dritten Version beschäftigen.

2.1 Sprachbeschreibung

Zur Darstellung einer Spezifikation für den Beweiser wird für die Regeln eine Implikationsnormalform verwendet, wobei die Regelrümpfe auf der linken und die Regelköpfe auf der rechten Seite stehen. Sowohl Rumpf als auch Kopf dürfen leer sein. Ein vollständige Grammatik ist hier in EBNF angegeben. Um diese übersichtlich zu halten, habe ich auf eine Übereinstimmung der nichtterminalen Symbole mit den Datenstrukturen im Programm verzichtet.

```
Kleinbuchstabe ::= "a" | "b" | ... | "z"
Grossbuchstabe ::= "A" | "B" | ... | "Z"
Ziffer         ::= "0" | "1" | ... | "9"
```

[1] Eine Regel ist genau dann bereichsbeschränkt, wenn jede Variable des Kopfes auch im Rumpf auftritt.

```
Whitespace     ::= " " | <newline>

Kommentar      ::= <newline> | <beliebiges anderes Zeichen> Kommentar
Zwischenraum   ::= Whitespace | "%" Kommentar
BezeichnerRest ::= (Kleinbuchstabe | Grossbuchstabe | Ziffer | "_")*
Symbol         ::= Kleinbuchstabe BezeichnerRest
Varsymbol      ::= Grossbuchstabe BezeichnerRest
Zahl           ::= [ "+" | "-" ] Ziffer (Ziffer)*

Spezifikation  ::= (Regel)*
Regel          ::= Konjunktion "-->" Disjunktion "."
Konjunktion    ::= [ Atomformel ("&" Atomformel)* ]
Disjunktion    ::= [ Atomformel (";" Atomformel)* ]
Atomformel     ::= Symbol Termliste
Termliste      ::= [ "(" Term ("," Term)* ")" ]
Term           ::= Varsymbol | Symbol Termliste | Zahl
```

Beispiel für eine Spezifikation:

```
% Kommentare beginnen mit dem Prozentzeichen
gerade(X) --> ungerade(s(X)).
--> gerade(0).
gerade(X) & ungerade(X)-->.
baut(Person, mist) --> bleibt(Person, unentdeckt) ; wird_bestraft(Person).
-->. % Leere Regel ist Widerspruch. Kein Modell generierbar.
```

2.2 Grobe Struktur

Die Eingabedaten können entweder aus einem SCHEME-String oder aus einer Textdatei eingelesen werden und werden zunächst in einen von der Art der Eingabe unabhängigen Strom[2] von Zeichen umgewandelt (`zeichen-seq.scm`). Der Zeichenstrom wird von einem Scanner (`scanner.scm`) zu einem Strom [ASS85] von Token verarbeitet, der einem Parser (`parser.scm`) als Eingabe dient. Der Parser generiert aus der Spezifikation einen Strukturbaum in einer abstrakten Syntax. Die genannten drei Programmteile standen mir bereits zur Verfügung und waren nicht Teil meiner Arbeit.

Den eigentlichen Inferenzmotor enthält die Datei `inferenz.scm`. Er erzeugt einen Strom aller generierbaren Modelle. `praed.scm` enthält Hilfsfunktionen für den Inferenzmotor und sorgt für die schrittweise Instanziierung der Regelrümpfe

[2] Ein Strom ist ein Konstrukt, das semantisch mit Listen identisch ist bis auf zwei Unterschiede. Auch ein Strom ist eine Sequenz von Daten, die sich in Kopf und Rest teilt. Der erste Unterschied besteht darin, daß der Rest nicht in Form von Daten, sondern in Form einer Berechnungsfunktion für die Daten vorliegt, die erst bei einem Lesezugriff auf dem Rest ausgewertet wird. Daraus resultiert der zweite Unterschied, nämlich daß Ströme unendlich sein können.

und die Überprüfung der Anwendbarkeit von Regeln. `bereichs-beschr.scm` enthält Funktionen, die eine als Strukturbaum vorliegende Spezifikation auf Bereichsbeschränktheit prüfen.

Die Datei `datenbasis.scm` enthält die Implementierung einer Datenbasis, die während des Inferenzprozesses Fakten aufsammelt und überprüft. Für die arithmetischen Prädikate `plus` , `mal` und `positiv` werden Funktionen in der Datei `builtin.scm` definiert. `subst.scm` beschäftigt sich mit Substitutionen und deren Operationen, `match.scm` enthält einen Pattern-Matching-Algorithmus.

`gleichheit.scm` ist eine Sammlung von Gleichheitsprädikaten auf den im Programm eingeführten Datentypen für die abstrakte Syntax, und `abssyn.scm` führt diese Datentypen ein. In `trace.scm` sind Funktionen zur Ausgabe von Hilfsinformationen während des Progammlaufes. `unparse.scm` dient zur leserlichen Ausgabe verschiedener Datentypen. `util.scm` enthält kleine Hilfsfunktionen, hauptsächlich zur Behandlung von Listen. Der Lader `loader.scm` dient zum Laden aller Dateien in der richtigen Reihenfolge, und der Treiber `treiber.scm` stellt schließlich die Benutzerschnittstelle dar.

2.3 Datentypen für die abstrakte Syntax (`abssyn.scm`)

Hier werden die Datentypen für die Knoten des Strukturbaumes einer Spezifikation eingeführt. Es folgt eine abstrakte Syntax, die die Beziehungen der neun Typen untereinander beschreibt. Der Pfeil „`=>`“ steht für die Beziehung „besteht aus“. Die Bezeichung [*Typ*] steht für eine Liste über *Typ*. Der Typ `TERM` ist eine Vereinigung von `variable` und `funktion`.

```
spezifikation     => [regel]
regel             => rumpf kopf
rumpf             => [atomformel]
kopf              => [atomformel]
atomformel        => praedikatensymbol [TERM]
praedikatensymbol => bezeichner
variable          => bezeichner
funktion          => funktionssymbol [TERM]
funktionssymbol   => bezeichner
```

2.4 Eingabe einlesen (`zeichenseq.scm`)

Zu Beginn der Verarbeitung wird die Eingabe, die in Form eines Strings oder einer Textdatei vorliegen kann, einheitlich in einen Zeichenstrom umgewandelt.

2.5 Scanner (`scanner.scm`)

Der Scanner ist als endlicher Automat realisiert, dessen Zustände und Übergänge fest „verdrahtet“ im Scheme-Code implementiert sind. Eingabe des Scanners ist ein Strom von Zeichen. Ausgabe ist ein Strom von Token, der dem Parser als Eingabe dient.

2.6 Parser (`parser.scm`)

Der Parser verarbeitet einen Tokenstrom zu einem Strukturbaum in der oben beschriebenen abstrakten Syntax.

2.7 Gleichheit von Objekten der abstrakten Syntax (`gleichheit.scm`)

Für jeden Datentyp der abstrakten Synatx wird ein Gleichheitsprädikat definiert. Außerdem wird für den Typ `praedikatensymbol` noch ein Kleiner-Vergleich implementiert, der für die Datenbasis benötigt wird, um Atomformeln nach ihren Prädikatensymbolen sortieren zu können.

Die Definitionen der Funktionen sind rekursiv, falls die Komponenten des Typs wieder zur abstrakten Syntax gehören. Ansonsten werden SCHEME-Vergleichsoperationen verwendet.

2.8 Hilfsfunktionen (`util.scm`)

`util.scm` enthält eine Sammlung von Hilfsfunktionen, die im Programm an sehr verschiedenen Stellen verwendet werden.

`all` und `any` realisieren eine Und- bzw. Oder-Verknüpfung über einer Menge von Wahrheitswerten, die durch Anwendung eines einstelligen Prädikates auf die Elemente einer Liste von Argumenten ensteht.

`foldl` generiert aus einer Liste einen Ergebniswert. Dazu benötigt es eine zweistellige Funktion und einen Startwert. Im ersten Schritt wird der Startwert und das erste Listenelement in die Funktion eingesetzt. Das Ergebnis bildet den Startwert für die nächste Anwendung der Funktion. `(foldl + 0 '(1 2 3 4 5))` berechnet die Summe der ersten fünf natürlichen Zahlen gemäß des Ausdruckes $((((0+1)+2)+3)+4)+5$. Symmetrisch zu `foldl` ist die Funktion `foldr`, diese beginnt aber mit dem letzten Element der Liste in umgekehrter Richtung. `(foldr + 0 '(1 2 3 4 5))` berechnet ebenfalls die Summe der ersten fünf natürlichen Zahlen, aber gemäß des Ausdruckes $1+(2+(3+(4+(5+0))))$.

`filter` erzeugt aus einer Liste eine neue Liste, die alle Elemente der ersten Liste enthält, für die ein gegebenes einstelliges Prädikat zutrifft.

2.9 Substitutionen (`subst.scm`)

Eine Substitution ist eine Liste von Bindungen, eine Bindung ist ein Paar `(variable wert)`, wobei `variable` und `wert` Terme im Sinne von `abssyn.scm` sind, `wert` aber variablenfrei ist. Da wir nur mit bereichsbeschränkten Regeln arbeiten, wird diese Bedingung automatisch eingehalten. Als Schnittstellenoperationen haben wir folgende Funktionen.

`leere-substitution`: Erzeugt die leere (identische) Substitution.

`substitution-erweitern`: Erweitert eine Substitution um eine zusätzliche Bindung. Einen Test auf Inkonsistenz (ob einer Variable zwei verschiedene Werte zugewiesen werden) gibt es hier nicht.

`substitutionen-kombinieren`: Berechnet die allgemeinste gemeinsame Spezialisierung von zwei Substitutionen, allerdings für den Spezialfall variablendisjunkter Grundsubstitutionen. In diesem Spezialfall entfällt ein Test, ob die Bindungen verträglich sind.

`wert-von` wendet eine bestehende Substitution auf eine Variable an, um den gebundenen Wert zu bekommen, falls eine Bindung existiert. Zusätzlich zur Variable und zur Substitution müssen noch eine „Success-Continuation“ und eine „Failure-Continuation“ angegeben werden, also zwei Funktionen, zu denen im Erfolgs- bzw. im Fehlerfall verzweigt wird. Das Ergebnis der Funktion ist im Erfolgsfall der Wert der Success-Continuation, die als Parameter den gebundenen Wert bekommt.

`regel-instanziieren` wendet eine Substitution auf eine ganze Regel an und generiert so eine Instanz der Regel.

2.10 Pattern-Matching (`match.scm`)

Pattern-Matching ist ein Spezialfall von Unifikation. Auch hier besteht die Aufgabe darin, eine Substitution zu finden, die zwei Atomformeln zur Deckung bringt. Im Gegensatz zur Unifikation darf aber nur eine der beiden Atomformeln instanziiert werden. Gesucht ist eine Substitution σ, für die $\sigma(t_1) = t_2$ ist, wobei t_1 und t_2 gegebene Atomformeln sind[3].

`atomformel-matchen` bekommt zwei Atomformeln, von denen nur die erste Variablen enthalten darf. Außerdem gibt man noch eine Success-Continuation an, die im Falle des Findens einer Substitution mit dieser als Argument aufgerufen wird, und eine Failure-Continuation, die den Fall abdeckt, daß die Atomformeln nicht zur Deckung gebracht werden können.

Die Suche nach einer Substitution wird rekursiv durchgeführt. Eine Bindung wird genau dann erzeugt, wenn der erste Term eine bis dato ungebundene Variable ist. Diese wird dann mit dem zweiten Term gebunden und die bis dahin aufgebaute Substitution um die neue Bindung erweitert. Ist die Variable bereits gebunden, so wird statt der Variable der Wert dieser Bindung eingesetzt und fortgefahren.

2.11 Datenbasis (`datenbasis.scm`)

Während der Modellgenerierung werden Regeln „angewandt“, wodurch variablenfreie Atomformeln entstehen. Die Aufgabe der Datenbasis ist es, diese Atomformeln zu speichern. Eine Überprüfung auf ein mehrfaches Vorhandensein von Atomformeln in der Datenbasisfindet findet nicht statt. Auf die Korrektheit des Interpretierers hat dies jedoch keinen Einfluß.

Im ersten Ansatz wurde die Datenbasis als einfache Liste von Atomformeln realisiert. Das Suchen nach bestimmten Atomformeln und das Erzeugen aller Instanzen zu einer Atomformel (und die daraus resultierenden Substitutionen) hatte demnach lineare Komplexität in der Anzahl der gespeicherten Atomformeln. Bei etwas größeren Spezifikationen ist dieses Verhalten unbefriedigend. Bei der Suche nach einer Verbesserung stellt sich folgendes Problem. Will man die Atomformeln in einer sortierten Struktur wie in einem Suchbaum speichern, so muß man eine Ordnung auf den Atomformeln definieren.

[3] Bei der Unifikation muß $\sigma(t_1) = \sigma(t_2)$ sein.

Dies ist zwar möglich[4], und ein Test, ob eine grundinstanziierte Atomformel in der Datenbasis ist, läßt sich so effizienter durchführen als in einer unsortierten Liste. Bei der Suche nach allen Instanzen einer Atomformel mit Variablen (Hierfür wird das Pattern-Matching benötigt) kann jedoch die Ordnungsrelation nur noch bei Atomformeln mit unterschiedlichen Prädikatensymbolen berechnet werden[5]. Folglich müssen trotz der Sortierung alle Atomformeln mit dem gleichen Prädikatensymbol untersucht werden.

Aus diesen Gründen entschied ich mich für eine kombinierte Datenstruktur für die Datenbasis. Die Prädikatensymbole werden alphabetisch sortiert. Die Datenbasis ist ein Suchbaum, dessen Knoten genau diejenigen Prädikatensymbole sind, von denen mindestens eine Atomformel in der Datenbasis ist. Jeder dieser Knoten enthält eine Liste mit allen Atomformeln mit diesem Prädikatensymbol. Aus Gründen der Einfachheit wird der implementierte Baum nicht balanciert, was bedeutet, daß die Komplexität der Suche im schlechtesten Fall linear ist in der Anzahl der Prädikatensymbole. Im mittleren und besten Fall ist sie jedoch logarithmisch.

Bei einer festen Anzahl von Prädikatensymbolen ist die Komplexität der Suche aus oben genannten Gründen linear in der Anzahl der Atomformeln. In einer Spezifikation mit nur einem einzigen Prädikatensymbol ergibt sich kein Geschwindigkeitsvorteil gegenüber einer unsortierten Liste. Im Gegenteil macht sich der höhere Verwaltungsaufwand negativ bemerkbar.

Aber selbst bei einem entarteten Baum ergibt sich schon bei zwei verschiedenen Prädikatensymbolen im besten Fall eine Halbierung der Suchzeit gegenüber einer unsortierten Liste, weil bei der Suche nach allen Instanzen nur noch die Hälfte der Datenbasis durchsucht werden muß.

Im Folgenden gehe ich auf die Implementierung ein. Zu Beginn werden vier Datentypen eingeführt:

```
(define-record binbaum (knoten links rechts))
(define-record knoten (atomliste))
(define-record leerer-binbaum ())
(define-record datenbasis (datenbaum))
```

Der leere Baum wird als eigener Datentyp eingeführt, der keine Daten enthält und nur als Terminierungssymbol eingesetzt wird.

`datenbasis->liste` erzeugt eine Liste aller Atomformeln in der Datenbasis. Diese Funktion wird z.B. gebraucht, um am Ende der Berechnung das komplette Modell auslesen zu können, das in der Datenbasis aufgebaut wurde. Als angenehmen Nebeneffekt aus der Suchbaumstruktur erhält man eine Liste, die alphabetisch nach Prädikatensymbolen sortiert ist.

[4] Etwa durch eine lexikographische Ordnung auf den Prädikatensymbolen. Bei gleichen Prädikatensymbolen werden die Termlisten wiederum lexikographisch verglichen. Zwei Terme werden dann nach dem Funktionssymbol verglichen und bei gleichen Funktionssymbolen werden rekursiv deren Termlisten wiederum lexikographisch verglichen.

[5] Oben genannte Ordnung läßt sich nur auf variablenfreien Atomen definieren.

`unparse-datenbasis` erzeugt eine Bildschirmdarstellung des Inhaltes der Datenbasis.

`leere-datenbasis` erzeugt eine leere Datenbasis.

`atomformel-hinzufuegen` fügt eine neue Atomformel in die Datenbasis ein. Wenn schon ein Knoten mit dem passenden Prädikatensymbol existiert, so wird die Atomformel lediglich in die Liste dieses Knotens aufgenommen. Ansonsten wird ein neuer Knoten angelegt. Sehr einfach wird die Sache dadurch, daß aus der Datenbank niemals eine Atomformel entfernt werden muß. Dies ist der Fall, weil wir bei der Modellgenerierung ein Backtracking durch Ströme modellieren.

`atomformel-in-datenbasis?` ist ein Test, ob eine Atomformel in der bestehenden Datenbasis enhalten ist. Die Formel darf keine Variablen enthalten. Rückgabewert ist `#t` oder `#f`. Um Substitutionen für Atomformeln mit Variablen zu finden, gibt es eine weitere Funktion:

`passende-substitutionen` sucht in der Datenbasis nach allen Instanzen einer Atomformel A. Ergebnis ist eine Liste aller σ, für die $\sigma(A)$ in der Datenbasis enthalten ist. Die Liste ist leer, wenn keine Instanz der Atomformel in der Datenbasis ist. Bei der Suche nach den Instanzen wird zuerst der Knoten mit dem passenden Prädikatensymbol gesucht. Hierbei wird die Suchbaumeigenschaft ausgenutzt. Die im Knoten enthaltene Liste muß komplett durchsucht werden.

2.12 Eingebaute Prädikate und Arithmetik (`builtin.scm`)

Für manche Aufgaben eines Regelinterpretierers ist es wünschenswert, eine Arithmetik direkt im Interpretierer implementiert zu haben. Deshalb habe ich eine minimale Ganzzahlarithmetik implementiert, die drei eingebaute Prädikate `plus`, `mal` und `positiv` zur Verfügung stellt. Diese Prädikate werden von der Datenbasis auf besondere Weise behandelt. Der Wahrheitstest und die Instanzensuche wird bei diesen Prädikaten nicht anhand von gespeicherten Atomformlen durchgeführt, sondern vom Regelinterpretierer selbst berechnet. Dabei simuliert die Datenbasis das Vorhandensein unendlich vieler Fakten.

```
       .                .                .                .
       :                :                :                :
...    plus(-2,-2,-4)   plus(-2,-1,-3)   plus(-2,0,-2)    plus(-2,1,-1)    ...
...    plus(-1,-2,-3)   plus(-1,-1,-2)   plus(-1,0,-1)    plus(-1,1, 0)    ...
...    plus( 0,-2,-2)   plus( 0,-1,-1)   plus( 0,0, 0)    plus( 0,1, 1)    ...
...    plus( 1,-2,-1)   plus( 1,-1, 0)   plus( 1,0, 1)    plus( 1,1, 2)    ...
...    plus( 2,-2, 0)   plus( 2,-1, 1)   plus( 2,0, 2)    plus( 2,1, 3)    ...
       .                .                .                .
       :                :                :                :
```

```
         .              .              .            .
         :              :              :            :
...  mal(-2,-2, 4)  mal(-2,-1, 2)  mal(-2,0,0)  mal(-2,1,-2)  ...
...  mal(-1,-2, 2)  mal(-1,-1, 1)  mal(-1,0,0)  mal(-1,1,-1)  ...
...  mal( 0,-2, 0)  mal( 0,-1, 0)  mal( 0,0,0)  mal( 0,1, 0)  ...
...  mal( 1,-2,-2)  mal( 1,-1,-1)  mal( 1,0,0)  mal( 1,1, 1)  ...
...  mal( 2,-2,-4)  mal( 2,-1,-2)  mal( 2,0,0)  mal( 2,1, 2)  ...
         .              .              .            .
         :              :              :            :


     positiv(1)     positiv(2)     positiv(3)   positiv(4)  ...
```

Bei der Ermittlung aller Instanzen zu einem builtin-Atom müssen verschiedene Möglichkeiten behandelt werden, wie das Atom instanziiert sein kann.

`plus(1,3,5)` liefert keine Substitution.
`mal(2,2,4)` liefert die leere Substitution.
`plus(X,4,5)` liefert $\{X/1\}$.
`plus(2,A,B)` liefert $\{A/0, B/2\}, \{A/-1, B/1\}, \{A/1, B/3\}$... also unendlich viele Substitutionen.
`mal(X,X,X)` liefert $\{X/0\}$ und $\{X/1\}$.
`plus(4,X,5)` liefert $\{X/1\}$. Mit diesem Instanziierungsmuster kann man Differenzen berechnen.

Wie man sieht, gibt es Atomformeln mit unendlich vielen Instanzen. In diesen Fällen wird zur Laufzeit ein Instanziierungsfehler gemeldet und das Programm abgebrochen. Dies ist eine pragmatische Entscheidung[6]. Ebenso wurde auf eine korrekte Behandlung von Fällen wie `mal(P,Q,10)` verzichtet, in denen es zwar nur endlich viele Instanzen gibt, zu deren Berechnung aber eine Primfaktorzerlegung notwendig wäre.

Die Einbindung in den Regelinterpretierer geschieht an zwei Stellen. Zum einen wird in der Datenbasis in den Funktionen `passende-substitutionen` und `atomformel-in-datenbasis?` nach einer Fallunterscheidung zur Funktion `builtin?` in entsprechende Funktionen für eingebaute Prädikate verzweigt (`builtin?` entscheidet anhand des Prädikatensymbols und der Stelligkeit, ob es sich bei einer Atomformel um eines der eingebauten Prädikate handelt).

Weiterhin wird direkt im Inferenzmotor die Behandlung des Falles integriert, in dem eingebaute Prädikate in Köpfen auftauchen. Im Fall, daß es sich um eine erfüllte Atomformel handelt (z.B. `positiv(42)`), ist der Kopf und somit

[6]Anderfalls müßte das Programm in der Lage sein, mit Strömen von Substitutionen zu arbeiten, da Listen wegen ihrer Endlichkeit ausscheiden. Weitere Folge wäre, daß bei Regeln wie `plus(A,B,C) ---> q(A) ; q(B) ; q(C)` in einem Ableitungsschritt unendlich viele Regelinstanzen gleichzeitig anwendbar sind. Die Behandlung solcher Fälle verlangt eine komplette Umstrukturierung des Inferenzmotors und ist ohnehin nur für Widerspruchsbeweise interessant.

die ganze Regel ohnehin erfüllt und kann übergangen werden. Dieser Fall wird schon durch eine Integration der eingebauten Prädikate in die Datenbasis abgefangen. Für den Fall jedoch, daß eine unerfüllte Atomformel wie `plus(1,1,3)` im Kopf auftaucht, muß diese aus der Liste der im Kopf enthaltenen Alternativen gelöscht werden, was dazu führen kann, daß der Kopf leer wird und somit die Regel einen Widerspruch liefert. Der Inferenzmotor simuliert auf diese Weise das Vorhandensein unendlich vieler Widerspruchsregeln.

```
.
:
plus(0,0,1) -->.
plus(0,0,2) -->.
.
:
mal(4,5,7) -->.
.
:
positiv(-1) -->.
positiv(0) -->.
```

2.13 Bereichsbeschränktheitstest (`bereichs-beschr.scm`)

Da der Regelinterpretierer nur bereichsbeschränkte Regeln korrekt behandelt, muß diese Eigenschaft der Eingabespezifikation überprüft werden. Die Bereichsbeschränkung ist ein syntaktisches Kriterium und läßt sich daher statisch überprüfen. Eine dynamische Kontrolle zur Laufzeit ließe sich zwar knapper (z.B. im Pattern-Matching-Algorithmus) implementieren, hätte aber den Nachteil einer größeren Laufzeit des Programmes durch redundante Tests.

Die Funktion `bereichsbeschraenkt?` erwartet als Eingabe eine Spezifikation in Form eines abstrakten Strukturbaumes. Die Arbeitsweise der Funktion ist einfach: Die Spezifikation ist bereichsbeschränkt, wenn alle Regeln bereichsbeschränkt sind, und eine Regel ist bereichsbeschränkt, wenn jede Variable des Kopfes auch im Rumpf auftritt. Mit den Funktionen `variablen-in-kopf` und `variablen-in-rumpf` werden dazu die Variablen in Kopf und Rumpf ermittelt und anschließend eine Mengendifferenz gebildet, die genau dann leer ist, wenn die Regel bereichsbeschränkt ist.

2.14 Regeln anwenden und instanziieren (`praed.scm`)

Die in dieser Datei enthaltenen Funktionen dienen zum Testen der Anwendbarkeit von Regelinstanzen bei einer gegebenen Datenbasis. Eine Regelinstanz ist genau dann anwendbar, wenn alle Atome des Rumpfes in der Datenbasis enthalten sind, jedoch kein Atom aus dem Kopf. Außerdem werden die Köpfe der anwendbaren Regeln instanziiert.

2.15 Inferenzmotor (`inferenz.scm`)

Diese Datei besteht aus zwei Komponenten. Zum einen aus der Steuerung des Inferenzmotors, zum anderen aus Funktionen für eine Optimierung durch einen Subsumptionstest. Die Hauptfunktion des Inferenzmotors heißt `inferenzmotor` und erwartet als Eingabe eine Spezifikation und zwei boolsche Werte, mit deren erstem der Tracer und mit deren zweitem der Subsumptionstest eingeschaltet werden kann. Rückgabe ist ein Strom von Datenbasen. Jedes Element dieses Stromes ist ein Modell der Spezifikation. Einen leeren Strom als Ergebnis erhält man für inkonsistente Spezifikationen. Die Funktion ist als Schleife realisiert, die schrittweise den Strom der Datenbasen konstruiert. In jedem Durchlauf werden zunächst alle ableitbaren Köpfe[7] bezüglich der aktuellen Datenbasis ermittelt. Dann werden aus den Köpfen alle Atome von builtin-Prädikaten entfernt und, falls vom Benutzer angefordert, auf gegenseitige Subsumption getestet. Ein Kopf subsumiert einen anderen genau dann, wenn alle seine Atome auch im anderen Kopf vorkommen. In diesem Fall kann der subsumierte Kopf aus der Menge der zu behandelnden Köpfe gestrichen werden.

Sind nach der Subsumption keine Köpfe mehr zu behandeln, so ist der Inferenzprozeß an seinem Ende angelangt. Rückgabe ist der Strom, der lediglich aus der aktuellen Datenbasis besteht. Ansonsten werden die Köpfe in jede Datenbasis des aktuellen Stromes eingefügt. Da das Ergebnis dieses Einfügens wiederum ein Strom von Datenbasen ist, erhalten wir so einen Strom von Strömen von Datenbasen, der dann mit der Funktion `stream-flatten` wieder zu einem einfachen Strom von Datenbasen gemacht wird.

Das Einfügen der Köpfe in eine Datenbasis leistet die Funktion `koepfe-einfuegen`. Sie bekommt als Eingabe eine Datenbasis und eine Liste von Köpfen und liefert als Antwort einen Strom von Datenbasen, der für jede mögliche Kombination der Alternativen aus den Köpfen eine Datenbasis enthält. Ist einer der Köpfe leer (was einem Widerspruch entspricht), so gibt es auch keine Auswahlmöglichkeit und als Antwort folglich (korrekterweise) einen leeren Strom.

2.16 Treiber (`treiber.scm`)

Diese Datei enthält die Benutzerschnittstelle. Hier finden sich die Funktionen zum Einlesen einer Spezifikation und zum Starten des eigentlichen Inferenzmotors. Als Benutzerschnittstelle stehen die Funktionen `modell-erzeugen-aus-datei` und `modell-erzeugen-aus-string` zur Verfügung, mit denen der Inferenzmotor gestartet werden kann, wobei die Eingabe entweder aus einer Textdatei oder aus einem String kommen kann.

2.17 Zusammengesetzte Datentypen in SCHEME

Da die Sprache SCHEME von sich aus nur rudimentäre Konzepte zur Definition zusammengesetzter Datentypen anbietet, verwende ich eine Erweiterung [CH91] von SCHEME. Diese erlaubt den Aufruf `(define-record regel (rumpf kopf))`, welcher automatisch vier Funktionen generiert:

[7] Dies sind die Köpfe der anwendbaren Regelinstanzen wie in `praed.scm` definiert

```
(define make-regel (lambda (rumpf kopf) ... )) => regel
(define regel->rumpf (lambda (regel) ... ))    => objekt
(define regel->kopf (lambda (regel) ... ))     => objekt
(define regel? (lambda (regel) ... ))          => boolean
```

Dabei ist die Funktion `make-<typ>` ein Konstruktor, die Funktionen `<typ>-><bestandteil>` sind Zugriffsoperationen auf die einzelnen Komponenten und `<typ>?` ist ein Test, der ein Objekt auf Zugehörigkeit zum Typ prüft.

3 Bedienung des Programmes

Um das Programm zu starten, benötigt man Scheme-48 [KR92] mit einigen Zusätzen für die Verwendung der Funktionen zum Definieren von komplexen Datentypen und Strömen. Dieses Scheme wird an den Lehrstuhlrechnern mit dem Aufruf `pms-scheme` gestartet. Um das Programm zu laden, muß in Scheme die Datei `loader.scm` geladen werden, welche ihrerseits alle anderen Programmteile lädt.

Hat man eine kleine Spezifikation in Form eines Strings vorliegen, so kann man den Inferenzmotor mit der Funtion `modell-erzeugen-aus-string` starten.

```
Welcome to Scheme 48 0.36 (suspended image).
Copyright (c) 1993, 1994 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@martigny.ai.mit.edu.
Type ,? (comma question-mark) for help.
> ,load loader.scm
loader.scm util.scm ..........
abssyn.scm ......................................
gleichheit.scm ...........
unparse.scm ..........
subst.scm .....................
match.scm .....
zeichen-seq.scm ......
datenbasis.scm ................builtin.scm .......................
................................
scanner.scm ..........................................
parser.scm ..............................
bereichs-beschr.scm ...........
trace.scm ......
praed.scm .....
treiber.scm ............

> (modell-erzeugen-aus-string #f #f "-->f(a);f(b). f(X)&f(Y)-->g(X,Y).")
-----------------------------------------------------------
```

```
Folgendes Modell wurde gefunden:
f(a)
g(a,a)

Weitere Modelle berechnen (j/n)? j
------------------------------------------------------------
Folgendes Modell wurde gefunden:
f(b)
g(b,b)

Weitere Modelle berechnen (j/n)? j
Keine weiteren Modelle.
>
```

Mit der Funktion `modell-erzeugen-aus-datei` kann statt einem String der Name einer Asciidatei angegeben werden, die die Spezifikation enthält. Die beiden ersten Parameter der Funktionen `modell-erzeugen-aus...` entsprechen den beiden boolschen Parametern für den Inferenzmotor. Mit dem ersten kann ein Ablauftrace angefordert werden, wenn man ihn auf `#t` setzt. Der zweite schaltet die Subsumptionsoptimierung an.

4 Programmlisting

4.1 Abhängigkeiten der Module untereinander

Aus Abbildung 1 gehen die Aufrufabhängigkeiten der einzelnen Programmteile hervor. Aus Gründen der Übersichtlichkeit sind die Dateien `abssyn.scm`, `gleichheit.scm`, `unparse.scm`, `trace.scm`, `util.scm`, und `loader.scm`, nicht in der Grafik enthalten.

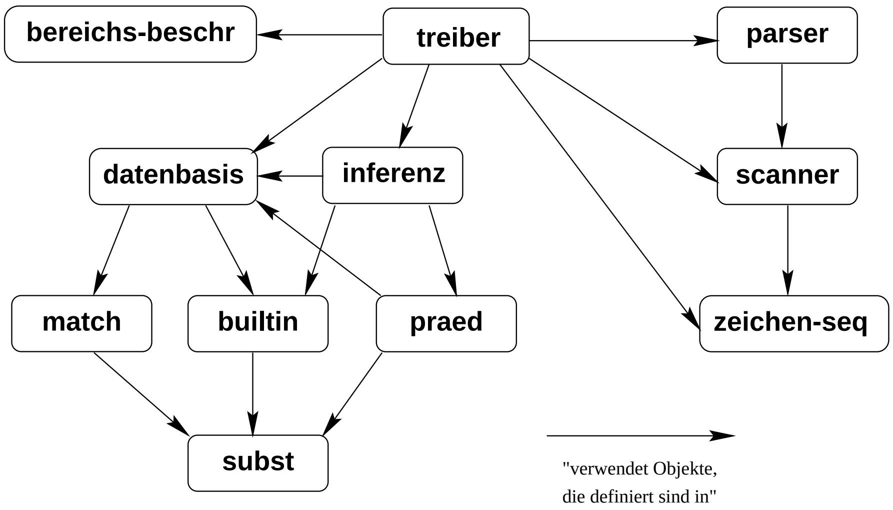

Abbildung 1: Abhängikeitsgraph der Programmteile

4.2 Programmteile in alphabetischer Reihenfolge

Im Anhang finden sich Listings aller Programmteile in alphabetischer Reihenfolge.

`abssyn.scm`	`bereichs-beschr.scm`
`builtin.scm`	`datenbasis.scm`
`gleichheit.scm`	`inferenz.scm`
`loader.scm`	`match.scm`
`parser.scm`	`praed.scm`
`scanner.scm`	`subst.scm`
`trace.scm`	`treiber.scm`
`unparse.scm`	`util.scm`
`zeichen-seq.scm`	

Literatur

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[KR92] Richard Kelsey and Jonathan Rees. *Scheme48*, June 1992. Documentation for the `scheme48` program.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In *9th Int. Conf. on Automated Deduction (CADE)*, Springer LNCS 310, pages 415–434, Argonne, IL, USA, 1988.

[MIT91] MIT-Press. *Revised^4 report on the algorithmic language scheme*, November 1991.