Kapitel 5:

Logik-Programmierung

Abschnitt 5.1:

Einführung

Logik-Programmierung

Logik-Programmierung bezeichnet die Idee, Logik direkt als Programmiersprache zu verwenden.

Logik-Programmierung (in Sprachen wie Prolog) und die verwandte funktionale Programmierung (in Sprachen wie LISP, ML, Haskell) sind deklarativ, im Gegensatz zur imperativen Programmierung (in Sprachen wie Java, C, Perl).

Die Idee der deklarativen Programmierung besteht darin, dem Computer lediglich sein Wissen über das Anwendungsszenario und sein Ziel mitzuteilen und dann die Lösung des Problems dem Computer zu überlassen.

Bei der imperativen Programmierung hingegen gibt man dem Computer die einzelnen Schritte zur Lösung des Problems vor.

Prolog

- ist die wichtigste logische Programmiersprache,
- geht zurück auf Kowalski und Colmerauer (Anfang der 1970er Jahre, Marseilles),
- steht für (franz.) Programmation en logique.
- Mitte/Ende der 1970er Jahre: effiziente Prolog-Implementierung durch den von Warren (in Edinburgh) entwickelten Prolog-10 Compiler.

Prolog ist eine voll entwickelte und mächtige Programmiersprache, die vor allem für symbolische Berechnungsprobleme geeignet ist.

Aus Effizienzgründen werden in Prolog die abstrakten Ideen der logischen Programmierung nicht in Reinform umgesetzt, Prolog hat auch „nichtlogische" Elemente.

Dieses Kapitel

- setzt voraus, dass Sie bereits Grundkenntnisse der Programmiersprache *Prolog* besitzen, die beispielsweise im Buch „Learn Prolog Now!" von P. Blackburn, J. Bos und K. Striegnitz vermittelt werden, und die während des Semesters bereits im Übungsbetrieb behandelt wurden.
- gibt eine Einführung in die Grundlagen der Logik-Programmierung — *keine* Einführung in die Programmiersprache Prolog!

Auf einige der Hauptunterschiede zwischen allgemeiner Logik-Programmierung und Prolog werden wir im Laufe dieses Kapitels eingehen.

Alle in diesem Kapitel enthaltenen Beispiele von Logikprogrammen sind voll lauffähige Prologprogramme, aber in einigen Fällen unterscheidet sich die Semantik des Programms im Sinne der Logik-Programmierung von der Semantik des Programms im Sinne von Prolog.

Zunächst zwei Beispiele für Logikprogramme

Beispiel 5.1

Ein Logikprogramm zur Repräsentation natürlicher Zahlen in Unärdarstellung und der zugehörigen Arithmetik und der Kleiner-Relation.

Programm: unat.pl

```
unat(null).
unat(s(X)) :- unat(X).

plus(null, Y, Y).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).

minus(X, Y, Z) :- plus(Y, Z, X).

mal(null, Y, null).
mal(s(X), Y, Z) :- mal(X, Y, Z1), plus(Z1, Y, Z).

less(null, s(X)).
less(s(X), s(Y)) :- less(X, Y).
```

Beispiel 5.2

Abschnitt 5.2:

Syntax und deklarative Semantik von Logikprogrammen

Logikprogramme

Logikprogramme sind „Wissensbasen“, bestehend aus einer endlichen Menge von Fakten und Regeln.

Eine Berechnung eines Logikprogramms besteht aus der Ableitung der Konsequenzen, die aus den Fakten und den Regeln des Programms hergeleitet werden können.

Man führt ein Programm aus, indem man Anfragen an die Wissensbasis stellt.

Fakten beschreiben Relationen zwischen Objekten.

Beispiele: `vater(gerald,scarlett)`, `maennlich(rhett)`, `party`, `plus(s(null),s(s(null)),s(s(s(null))))`.

Relationen haben eine Stelligkeit $k \geqslant 0$.
Nullstellige Relationen sind einfach Aussagen (z.B. besagt „`party`“, dass die Party stattfindet).

Eine **Anfrage** ist eine durch Kommas getrennte Liste von Fakten; gefragt wird, ob diese Fakten in der Wissenbasis gelten, d.h., ob sie aus der Wissensbasis ableitbar sind.

Beispiele: Die Anfrage `?- schwester(scarlett, suellen)` fragt, ob Scarlett eine Schwester von Suellen ist.
Die Anfrage `?- mutter(scarlett, X), vater(ashley, X)` fragt, ob Scarlett und Ashley ein gemeinsames Kind haben.

Die Rolle der Terme

Terme sind in Logikprogrammen <u>die</u> universelle Datenstruktur.

Je nach Kontext spielen sie die Rolle von Fakten
oder von Objekten, über die die Fakten sprechen.

Die einfachste Art von Termen in Logikprogrammen sind die im Folgenden definierten Konstanten und Variablen.

Atome, Zahlen, Konstanten und Variablen der Logik-Programmierung

Definition 5.3

(a) Atome sind die Grundbausteine von Logikprogrammen. Sie werden bezeichnet durch Zeichenketten, die keins der Symbole „(“ und „)“ enthalten und die mit einem Kleinbuchstaben beginnen oder in einfachen Hochkommata stehen. Atome repräsentieren Individuen.

Beispiele: `scarlett`, `'Scarlett'`, `logikInDerInformatik`

(b) Zahlen in Logikprogrammen sind entweder ganze Zahlen oder reelle Zahlen in Gleitkommadarstellung.

Beispiele: `42`, `1.2e-3`

(c) Konstanten der Logik-Programmierung sind Atome oder Zahlen.

Definition 5.4

Variablen der Logik-Programmierung werden durch Zeichenketten bezeichnet, die

- mit einem Großbuchstaben beginnen oder
- oder einem Unterstrich beginnen und Länge $\geqslant 2$ haben,

und keins der Symbole „(“ und „)“ enthalten.

Eine Variable repräsentiert in einem Logikprogramm (ähnlich wie in der Logik erster Stufe) ein nicht-spezifiziertes Individuum.

Beispiele: `X`, `Mutter`, `_mutter`, `RUD26`

Die anonyme PROLOG-Variable _ hat eine besondere Bedeutung und kommt der Einfachheit halber nicht in der Syntax unserer Logikprogramme vor.

Terme der Logik-Programmierung

Definition 5.5

(a) Ein einfacher Term der Logik-Programmierung ist eine Konstante oder eine Variable (d.h., ein Atom, eine Zahl oder eine Variable der Logik-Programmierung).

(b) Die Menge T_{LP} der Terme der Logik-Programmierung ist rekursiv wie folgt definiert:

(1) Jeder einfache Term ist ein Term.

(2) Ist `f` ein Atom, ist $k \in \mathbb{N}$ mit $k \geqslant 1$ und sind $t_1, \ldots, t_k \in \mathrm{T}_{\mathrm{LP}}$ Terme, so ist

$$\mathtt{f(}t_1\mathtt{,}\ldots\mathtt{,}t_k\mathtt{)}$$

ein Term in T_{LP}.

(c) Terme in T_{LP}, die keine einfachen Terme sind, heißen *zusammengesetzte Terme* der Logik-Programmierung.

In einem zusammengesetzten Term der Form $\mathtt{f(}t_1\mathtt{,}\ldots\mathtt{,}t_k\mathtt{)}$ spielt das Atom `f` die Rolle eines k-stelligen Funktors, den wir mit `f/k` bezeichnen.

Spezialfall $k = 0$: Jedes Atom g wird als ein 0-stelliger Funktor betrachtet, der mit g/0 bezeichnet wird, und der ein (einfacher) Term ist.

Beispiele: `party`, `vater(gerald,scarlett)`, `s(s(s(null)))`,

```
vorlesung(name(logikInDerInformatik),
          zeit(Mi,9,11),
          ort(gebaeude(RUD26),raum(0110))).
```

Gleichheit von Termen

Zwei Terme t und t' der Logik-Programmierung werden nur dann als *gleich* bezeichnet, wenn sie syntaktisch, d.h. als Zeichenketten betrachtet, identisch sind.

Beispiel:
Die beiden Terme `plus(null,X,X)` und `plus(null,Y,Y)` sind nicht gleich.

Substitutionen

Notation

Für eine partielle Funktion f schreiben wir $\text{Def}(f)$ und $\text{Bild}(f)$ um den Definitionsbereich und den Bildbereich von f zu bezeichnen.
D.h. $\text{Def}(f)$ ist die Menge aller Objekte x, für die der Wert $f(x)$ definiert ist, und $\text{Bild}(f) = \{f(x) \ : \ x \in \text{Def}(f)\}$.

Definition 5.6

Eine Substitution ist eine partielle Abbildung von der Menge der Variablen auf die Menge der Terme.

Eine Substitution für eine Menge V von Variablen der Logik-Programmierung ist eine Substitution S mit $\text{Def}(S) \subseteq V$.

Beispiel:

$$S := \left\{ \mathtt{X} \mapsto \mathtt{c}, \ \ \mathtt{Y} \mapsto \mathtt{f(X,g(c))}, \ \ \mathtt{Z} \mapsto \mathtt{Y} \right\}$$

bezeichnet die Substitution mit Definitionsbereich $\text{Def}(S) = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$, für die gilt:
$S(\mathtt{X}) = \mathtt{c}, \ \ S(\mathtt{Y}) = \mathtt{f(X,g(c))}, \ \ S(\mathtt{Z}) = \mathtt{Y}$.

Anwendung von Substitutionen

Durch Anwenden einer Substitution S auf einen Term $t \in \mathrm{T}_{\mathrm{LP}}$ erhalten wir den Term $tS \in \mathrm{T}_{\mathrm{LP}}$, der aus t durch simultanes Ersetzen jeder Variablen $X \in \mathrm{Def}(S)$ durch den Term $S(X)$ entsteht.

Beispiel: Sei

$$t \quad := \quad \mathtt{h(f(X,X),\ Y,\ f(Y,g(Z)))}$$

und

$$S \;:=\; \{\,\mathtt{X} \mapsto \mathtt{c},\ \ \mathtt{Y} \mapsto \mathtt{f(X,g(c))},\ \ \mathtt{Z} \mapsto \mathtt{Y}\,\}.$$

Dann ist

$$tS \quad = \quad \mathtt{h(f(c,c),\ f(X,g(c)),\ f(f(X,g(c)),\ g(Y)))}.$$

Definition 5.7

Ein Term t' ist eine Instanz eines Terms t, wenn es eine Substitution S gibt, so dass $t' = tS$.

Grundterme

Definition 5.8

Ein Grundterm der Logik-Programmierung ist ein Term, der keine Variable(n) enthält.

Eine Grundinstanz eines Terms $t \in \mathrm{T}_{\mathrm{LP}}$ ist eine Instanz von t, die ein Grundterm ist.

Eine Grundinstanz eines Terms t entsteht also, indem jede in t vorkommende Variable durch einen Grundterm ersetzt wird.

Beispiele: `h(c,c,f(c))` und `h(f(f(c,c),g(d)),d,f(g(g(c))))` sind Grundinstanzen des Terms `h(X,Y,f(Z))`.

Bemerkung

Grundterme sind wichtig, weil sie in dem Modell, das dem Logikprogramm zu Grunde liegt, eine unmittelbare Bedeutung haben. Variablen hingegen haben keine direkte Bedeutung, sondern sind nur Platzhalter für Objekte.

Fakten der Logik-Programmierung

Definition 5.9

Ein Faktum der Logik-Programmierung ist ein Atom oder ein zusammengesetzter Term der Logik-Programmierung.

Fakten beschreiben Tatsachen bzw. Relationen zwischen Objekten.

Beispiele: Das Faktum `party` beschreibt, dass eine Party stattfindet.
Das Faktum `unat(s(s(null)))` beschreibt, dass der Term `s(s(null))` die Unärdarstellung einer natürlichen Zahl ist.
Das Faktum `mutter(scarlett, bonnie)` beschreibt, dass Scarlett die Mutter von Bonnie ist.

Fakten dürfen auch Variablen enthalten. Eine Variable in einem Faktum bedeutet, dass die entsprechende Aussage für *alle* Objekte, durch die die Variable ersetzt werden kann, gilt.
Beispiel: `plus(null,Y,Y)`

Regeln

Definition 5.10

Eine Regel der Logik-Programmierung besteht aus

- einem Faktum (dem so genannten Kopf der Regel),
- gefolgt von `:-`
  (in der Literatur wird an Stelle von „`:-`“ oft auch „$\leftarrow$“ geschrieben) und
- einer durch Kommas getrennten Liste von Fakten (dem so genannten Rumpf der Regel).

Wir interpretieren die Regel als Implikation:

Wenn alle Fakten im Rumpf gelten, dann gilt auch das Faktum im Kopf.

Beispiele:

```
minus(X,Y,Z) :- plus(Y,Z,X)
grossmutter(X,Z) :- mutter(X,Y), elternteil(Y,Z)
```

Logikprogramme

Definition 5.11

Ein Logikprogramm ist eine endliche Menge von Fakten und Regeln der Logik-Programmierung.

Es ist oft bequem, Fakten als spezielle Regeln mit leerem Rumpf aufzufassen. Dann besteht ein Logikprogramm nur aus Regeln.

In konkreten Beispielen stellen wir Logikprogramme meistens als Liste der in ihnen enthaltenen Fakten und Regeln dar, wobei das Ende jedes Eintrags dieser Liste durch einen Punkt markiert wird.

Beispiele: Das Programm `unat.pl` aus Beispiel 5.1 ist ein Logikprogramm im Sinne von Definition 5.11. Das Programm `vomWindeVerweht.pl` aus Beispiel 5.2 nicht, da dort Ungleichheitsprädikate der Form `X \== Y` vorkommen, die gemäß Definition 5.10 nicht im Rumpf von Regeln vorkommen können, da sie keine Fakten gemäß Definition 5.9 sind.

Ableitungen aus Logikprogrammen

Definition 5.12

Eine Ableitung aus einem Logikprogramm Π ist ein Tupel $(t_1, \ldots, t_\ell)$ von Termen, so dass $\ell \in \mathbb{N}$ mit $\ell \geqslant 1$ ist und für jedes $i \in [\ell]$ (mindestens) eine der beiden folgenden Aussagen zutrifft:

- t_i ist eine Instanz eines Faktums in Π.
- Es gibt eine Regel
$$\varphi \;\texttt{:-}\; \psi_1, \ldots, \psi_m$$
in Π, eine Substitution S und Indizes $i_1, \ldots, i_m \in \{1, \ldots, i{-}1\}$, so dass gilt: $t_i = \varphi S$ und $t_{i_j} = \psi_j S$ für jedes $j \in [m]$.

Eine Ableitung eines Terms t aus Π ist eine Ableitung $(t_1, \ldots, t_\ell)$ aus Π mit $t_\ell = t$.

Ein Term t ist ableitbar aus Π, wenn es eine Ableitung von t aus Π gibt.

Die im Kapitel über Automatisches Schließen eingeführte Kalkül-Schreibweise lässt sich dazu nutzen, eine elegante Darstellung des Begriffs der Ableitungen aus Logikprogrammen anzugeben.

Verwendung der Kalkül-Schreibweise für Ableitungen in Logikprogrammen

Sei Π ein Logikprogramm.

Gesucht: Ein Kalkül $\mathfrak{K}_\Pi$ über der Menge T_{LP}, so dass $\mathsf{abl}_{\mathfrak{K}_\Pi}$ genau die Menge aller aus Π ableitbaren Terme ist.

Darstellung von Ableitungen

- An Stelle von $(t_1, \ldots, t_\ell)$
  schreiben wir Ableitungen der besseren Lesbarkeit halber oft zeilenweise, also

$$\begin{array}{ll} (1) & t_1 \\ (2) & t_2 \\ \vdots & \\ (\ell) & t_\ell \end{array}$$

  und geben am Ende jeder Zeile eine kurze Begründung an.

- Ableitungen werden oft auch als Bäume dargestellt; man bezeichnet diese als Beweisbäume.

Beispiel

Betrachte das Programm `vomWindeVerweht1.pl`

Beispiel 5.13

Ableitung von `tante(suellen,bonnie)` aus dem Programm `vomWindeVerweht1.pl`:

(1)	`mutter(ellen,scarlett)`	Faktum in Zeile 3
(2)	`elternteil(ellen,scarlett)`	Regel in Zeile 25 und (1)
(3)	`mutter(ellen,suellen)`	Faktum in Zeile 3
(4)	`elternteil(ellen,suellen)`	Regel in Zeile 25 und (3)
(5)	`ungleich(suellen,scarlett)`	Regel in Zeile 32
(6)	`weiblich(suellen)`	Faktum in Zeile 16
(7)	`schwester(suellen,scarlett)`	Regel in Zeile 27 und (4),(2),(6),(5)
(8)	`mutter(scarlett,bonnie)`	Faktum in Zeile 4
(9)	`elternteil(scarlett,bonnie)`	Regel in Zeile 25 und (8)
(10)	`tante(suellen,bonnie)`	Regel in Zeile 27 und (9),(7)

Beweisbäume

Definition 5.14

Sei Π ein Logikprogramm und sei t ein Term.
Ein Beweisbaum für t aus Π ist ein endlicher Baum, dessen Knoten mit Termen beschriftet sind, so dass gilt:

- die Wurzel ist mit dem „Ziel“ t beschriftet,
- jedes Blatt ist mit einer Instanz eines Faktums in Π beschriftet, und
- für jeden inneren Knoten u und dessen Kinder $v_1, \ldots, v_m$ gilt:
  Es gibt eine Regel

  $$\varphi \;\texttt{:-}\; \psi_1, \ldots, \psi_m$$

  in Π und eine Substitution S, so dass für die Beschriftung t_u von u und die Beschriftungen $t_{v_1}, \ldots, t_{v_m}$ der Knoten $v_1, \ldots, v_m$ gilt:

  $$t_u = \varphi S, \quad t_{v_1} = \psi_1 S, \quad t_{v_2} = \psi_2 S, \quad \ldots, \quad t_{v_m} = \psi_m S.$$

Man sieht leicht, dass es genau dann einen Beweisbaum für t aus Π gibt, wenn t aus Π ableitbar ist (Details: Übung).

Deklarative Semantik von Logikprogrammen

Definition 5.15

Sei Π ein Logikprogramm.

Die Bedeutung von Π ist die Menge $\mathcal{B}(\Pi)$ aller Grundterme, die aus Π ableitbar sind.

Beispiel 5.16

Sei Π das folgende Logikprogramm `unat1.pl`.

Programm: `unat1.pl`

```
unat(null).
unat(s(X)) :- unat(X).
less(null, s(X)) :- unat(X).
less(s(X), s(Y)) :- less(X, Y).
```

Die Bedeutung von Π ist die Menge $\mathcal{B}(\Pi)$, und diese enthält u.a. die Terme

Beispiel: Wege in Digraphen (d.h., gerichteten Graphen)

Wir repräsentieren einen gerichteten Graphen G durch die Auflistung `node(v)` für alle Knoten `v` von G und `edge(v,w)` für alle Kanten `(v,w)` von G.

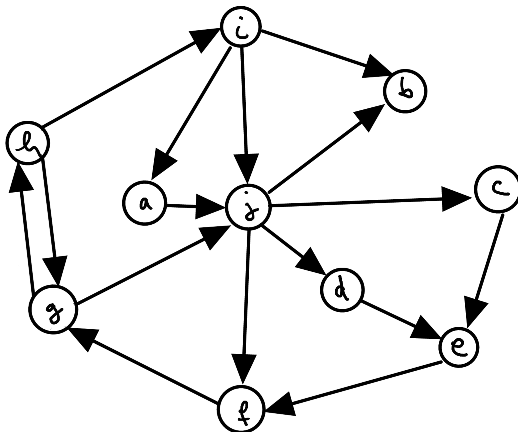
Ziel: `path(X,Y)` soll besagen, dass es in G einen Weg von Knoten `X` zu Knoten `Y` gibt.

Lösung:

```
path(X,X).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

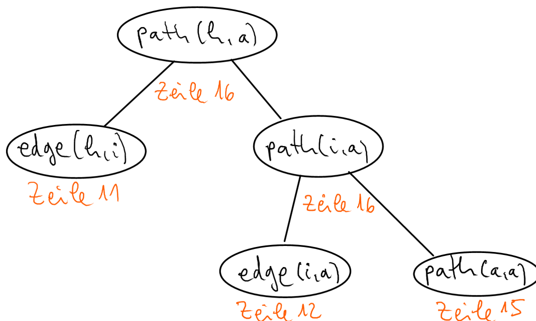
Im folgenden Programm `digraph.pl` ist dies zusammen mit einem Beispiel-Graphen gegeben.

Der in `digraph.pl` angegebene Graph sieht wie folgt aus:



Ein Beweisbaum für `path(a,g)` aus `digraph.pl`:

Ein Beweisbaum für `path(h,a)` aus `digraph.pl`:



Und was tut Prolog bei Eingabe von

```
?- consult(digraph).
?- path(a,g).
```

und bei Eingabe von

```
?- path(h,a).
```

?

Auf die Frage, ob `path(a,g)` gilt, antwortet Prolog mit „`true`“.

Auf die Frage, ob `path(h,a)` gilt, antwortet Prolog mit „`ERROR: Out of local stack`“.

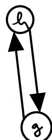
Was passiert hier?

Die Details zur Berechnung, die Prolog hier durchführt, können wir mit uns mit

```
?- trace.
?- path(h,a).
```

anschauen.

Dies zeigt, dass die Prolog-Suche nach einem Beweisbaum im Kreis



stecken bleibt.

Unterschied zwischen Theorie und Praxis

In der Theorie funktioniert die Pfadsuche aus `digraph.pl` für alle endlichen gerichteten Graphen.

In der Praxis funktioniert sie aber nur für azyklische Graphen.

Die operationelle Semantik von Prolog entspricht also nicht genau der deklarativen Semantik von Logikprogrammen!

Anfragen an Logikprogramme

Definition 5.17

Eine Anfrage der Logik-Progammierung besteht aus den Symbolen `?-` gefolgt von einem Faktum oder aus einer durch Kommas getrennten Liste von Fakten der Logik-Programmierung.

Die Antwort auf eine Anfrage α der Form

$$\texttt{?-}\ \alpha_1, \dots, \alpha_n$$

an ein Logikprogramm Π ist definiert als die Menge $[\![\alpha]\!]^{\Pi}$ aller Substitutionen S für die in α vorkommenden Variablen, so dass gilt:
$\alpha_1 S$, ..., $\alpha_n S$ sind Grundterme, die aus Π ableitbar sind.

Hier repräsentiert die leere Menge $\emptyset$ die Antwort „falsch".

Beachte: Eine Variable `X` in einer Anfrage fragt also nach einem bzw. allen Objekten, die die Anfrage erfüllen.

Beispiel 5.18

Betrachte die Anfrage

```
?- vater(gerald,X), mutter(ellen,X)
```

angewendet auf das Logikprogramm `vomWindeVerweht1.pl`.

Die Antwort auf diese Anfrage besteht aus den drei Substitutionen

$$\begin{aligned} S_1 &:= \{ \texttt{X} \mapsto \texttt{scarlett} \}, \\ S_2 &:= \{ \texttt{X} \mapsto \texttt{suellen} \}, \\ S_3 &:= \{ \texttt{X} \mapsto \texttt{carreen} \}. \end{aligned}$$

Beispiele von Anfragen an das Logikprogramm `unat.pl`:

```
?- plus(s(null),s(s(null)),X).
?- plus(X,Y,s(s(s(null)))).
```

Abschnitt 5.3:

Operationelle Semantik

Deklarative vs. Operationelle Semantik

- Die in Definition 5.15 festgelegte deklarative Semantik von Logikprogrammen beruht auf einer logischen Interpretation von Programmen (Regeln als Implikationen) und logischer Deduktion.

- Jetzt werden wir dieser deklarativen Semantik eine operationelle Semantik gegenüberstellen, indem wir einen Algorithmus angeben, der Programme ausführt (auf einem abstrakten, nichtdeterministischen Maschinenmodell). Dadurch legen wir ebenfalls die Antworten auf die Anfragen fest und weisen somit Programmen eine Bedeutung zu.

- Wir werden sehen, dass die deklarative Bedeutung von Logikprogrammen mit der operationellen übereinstimmt.

Semantik von Programmiersprachen im Allgemeinen

Generell unterscheidet man zwischen zwei Wegen, die Semantik von Programmiersprachen zu definieren:

- Die deklarative oder denotationelle Semantik ordnet Programmen Objekte in abstrakten mathematischen Räumen zu, in der Regel partielle Funktionen, oder im Fall von Logikprogrammen Mengen von Grundtermen.

  Zur Erinnerung: Die Bedeutung $\mathcal{B}(\Pi)$ eines Logikprogramms Π ist gemäß Definition 5.15 die die Menge aller Grundterme, die aus Π ableitbar sind.

- Die operationelle Semantik legt fest, wie Programme auf abstrakten Maschinenmodellen ausgeführt werden.

Notation

- LP := die Menge aller Logikprogramme

- A_{LP} := die Menge aller Atome der Logik-Programmierung

  V_{LP} := die Menge aller Variablen der Logik-Programmierung

  K_{LP} := die Menge aller Konstanten der Logik-Programmierung

  T_{LP} := die Menge aller Terme der Logik-Programmierung

  F_{LP} := die Menge aller Anfragen der Logik-Programmierung

  R_{LP} := die Menge aller Regeln der Logik-Programmierung

- Für jedes ξ aus $\mathsf{T}_{\mathsf{LP}} \cup \mathsf{F}_{\mathsf{LP}} \cup \mathsf{R}_{\mathsf{LP}} \cup \mathsf{LP}$ bezeichnet $\mathsf{Var}(\xi)$ die Menge aller Variablen, die in ξ vorkommen.

  Beispiel: Ist ρ die Regel `path(X,Y) :- edge(X,Z), path(Z,Y).`, dann ist $\mathsf{Var}(\rho) = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$.

- Ist S eine Substitution und $\alpha \in \mathsf{F}_{\mathsf{LP}}$ eine Anfrage der Form `?-` $\alpha_1, \ldots, \alpha_m$ ist, so bezeichnet αS die Anfrage `?-` $\alpha_1 S, \ldots, \alpha_m S$.

  Entsprechend definieren wir für jede Regel $\rho \in \mathsf{R}_{\mathsf{LP}}$ die Regel ρS.

Mehr über Substitutionen

- Zur Erinnerung: Eine Substitution ist eine partielle Abbildung S von V_{LP} nach T_{LP}. Den Definitionsbereich von S bezeichnen wir mit $\mathrm{Def}(S)$, den Bildbereich mit $\mathrm{Bild}(S)$.

- Die Verkettung zweier Substitutionen S und T ist die Substitution ST mit $\mathrm{Def}(ST) = \mathrm{Def}(S) \cup \mathrm{Def}(T)$ und $\mathtt{X}(ST) := (\mathtt{X}S)T$ für alle $\mathtt{X} \in \mathrm{Def}(ST)$.

- Die Einschränkung einer Substitution S auf eine Menge V von Variablen ist die Substitution $S|_V$ mit $\mathrm{Def}(S|_V) = \mathrm{Def}(S) \cap V$ und $\mathtt{X}S|_V := \mathtt{X}S$ für alle $\mathtt{X} \in \mathrm{Def}(S) \cap V$.

- Die leere Substitution bezeichnen wir mit I. Es gilt:
  - $tI = t$ für alle Terme $t \in \mathrm{T}_{\mathrm{LP}}$, und
  - $IS = SI = S$ für alle Substitutionen S.

Beispiel 5.19

Für die Substitutionen

$$\begin{aligned} S &:= \{ \texttt{X} \mapsto \texttt{good(c,Y)},\ \ \texttt{Y} \mapsto \texttt{rainy(d)} \}, \\ T &:= \{ \texttt{Y} \mapsto \texttt{sunny(d)},\ \ \texttt{Z} \mapsto \texttt{humid(e)} \}. \end{aligned}$$

gilt:

$$\begin{aligned} ST &= \{ \texttt{X} \mapsto \texttt{good(c,sunny(d))},\ \ \texttt{Y} \mapsto \texttt{rainy(d)},\ \ \texttt{Z} \mapsto \texttt{humid(e)} \} \\ TS &= \{ \texttt{X} \mapsto \texttt{good(c,Y)},\ \ \texttt{Y} \mapsto \texttt{sunny(d)},\ \ \texttt{Z} \mapsto \texttt{humid(e)} \}. \end{aligned}$$

Umbennungen

- Eine Umbenennung ist eine injektive partielle Abbildung von $\mathrm{V_{LP}}$ nach $\mathrm{V_{LP}}$.

  Wegen $\mathrm{V_{LP}} \subseteq \mathrm{T_{LP}}$, sind Umbenennungen spezielle Substitutionen.

- Eine Umbenennung für eine Menge V von Variablen ist eine Umbennung U mit $\mathrm{Def}(U) = V$.

- Ist U eine Umbennung, so bezeichnet U^{-1} ihre Umkehrung.

Beispiel: $U := \{\mathtt{X} \mapsto \mathtt{Y},\ \mathtt{Y} \mapsto \mathtt{Z}\}$ ist eine Umbenennung für $\{\mathtt{X}, \mathtt{Y}\}$.
$U^{-1} = \{\mathtt{Y} \mapsto \mathtt{X},\ \mathtt{Z} \mapsto \mathtt{Y}\}$ ist die Umkehrung von U.

Ein einfacher Interpreter für Logikprogramme

Algorithmus $\textsc{Antwort}(\Pi, \alpha)$

% Eingabe: Programm $\Pi \in \mathsf{LP}$, Anfrage ?- $\alpha \in \mathsf{F_{LP}}$ mit $\alpha = \alpha_1, \ldots, \alpha_m$

% Ausgabe: eine Substitution S für $\mathsf{Var}(\alpha)$ oder das Wort „gescheitert“.

1. Wähle ein $i \in [m]$ *% α_i ist das nächste „Ziel“*
2. Wähle eine Regel ρ aus Π. Sei φ :- $\psi_1, \ldots, \psi_n$ die Form von ρ.
   % Fakten fassen wir als Regeln ohne Rumpf auf
3. Sei U eine Umbennung für $\mathsf{Var}(\rho)$, so dass $\mathsf{Var}(\rho U) \cap \mathsf{Var}(\alpha) = \emptyset$.
4. Wähle eine Substitution T, so dass $\alpha_i T = \varphi U T$. Wenn dies nicht möglich ist, gib „gescheitert“ aus und halte an.
5. Wenn $m = 1$ und $n = 0$, gib $T|_{\mathsf{Var}(\alpha)}$ aus und halte an.
6. Setze $\alpha' := \alpha_1 T, \ldots, \alpha_{i-1} T, \psi_1 U T, \ldots, \psi_n U T, \alpha_{i+1} T, \ldots, \alpha_m T$.
7. Setze $T' := \textsc{Antwort}(\Pi, \alpha')$
8. Wenn T' eine Substitution ist, gib $(TT')|_{\mathsf{Var}(\alpha)}$ aus und halte an.
9. Gib „gescheitert“ aus und halte an.

Zum Nichtdeterminismus des Interpreters

- Das Programm ANTWORT ist nichtdeterministisch. Wir sprechen von verschiedenen Läufen des Programms, die durch die Auswahlen in den Zeilen 1–4 bestimmt sind.

- Ein Lauf heißt akzeptierend, wenn die Ausgabe eine Substitution ist.

- Von den nichtdeterministischen Auswahlschritten in den Zeilen 1–4 ist die Wahl der Substitution in Zeile 4 am problematischsten, weil hier ein Element einer unendlichen Menge ausgewählt wird, und weil nicht klar ist, wie man so ein Element überhaupt finden kann.

- Die Wahl der Umbennung in Zeile 3 hingegen ist unwesentlich. Jede Umbennung U, für die $\text{Var}(\rho U) \cap \text{Var}(\alpha) = \emptyset$ gilt, führt zum gleichen Ergebnis, und es ist leicht, eine solche Umbennung zu finden.

Korrektheit und Vollständigkeit des Interpreters

Satz 5.20

Seien $\Pi \in \mathsf{LP}$ *ein Logikprogramm, sei* ?- $\alpha \in \mathsf{F_{LP}}$ *eine Anfrage mit* $\alpha = \alpha_1, \ldots, \alpha_m$*, und sei* S *eine Substitution für* $\mathrm{Var}(\alpha)$*. Dann sind folgende Aussagen äquivalent:*

(a) *Die Terme* $\alpha_1 S, \ldots, \alpha_m S$ *sind aus* Π *ableitbar.*

(b) *Es gibt einen Lauf von* $\textsc{Antwort}(\Pi, \alpha)$*, der* S *ausgibt.*

Die Richtung *„(b)* $\Longrightarrow$ *(a)“* wird *Korrektheit des Interpreters* genannt; die Richtung *„(a)* $\Longrightarrow$ *(b)“ Vollständigkeit*.

Für den Spezialfall, dass $m = 1$ und α ein Grundterm ist, erhalten wir das folgende Korollar.

Korollar 5.21

Sei $\Pi \in \mathsf{LP}$ *ein Programm und sei* α *ein Grundterm. Dann gilt:*
$\alpha \in \mathcal{B}(\Pi) \iff$ *es gibt einen akzeptierenden Lauf von* $\textsc{Antwort}(\Pi, \alpha)$.

Nächstes Ziel:
Auflösen des Nichtdeterminismus in Zeile 4

Als ein Hauptproblem des nichtdeterministischen Interpreters ANTWORT haben wir die Wahl der Substitution T in Zeile 4 identifiziert.

Mit Hilfe der im Folgenden vorgestellten Unifikatoren können die richtigen Substitutionen auf deterministische Art gefunden werden.

Unifikation

Definition 5.22

Seien $t, s \in \mathrm{T}_{\mathrm{LP}}$ Terme der Logik-Programmierung.

(a) Ein Unifikator für t und s ist eine Substitution S, so dass $tS = sS$.

(b) t und s sind unifizierbar, wenn es einen Unifikator für t und s gibt.

Beispiel 5.23

$t := \texttt{mal(s(X), Y, s(Z))}$ und $s := \texttt{mal(s(s(null)), Y, Y)}$ sind unifizierbar.
Ein Unifikator ist

$$S \; := \; \{\, \texttt{X} \mapsto \texttt{s(null)}, \;\; \texttt{Y} \mapsto \texttt{s(Z)} \,\}.$$

Die entstehende gemeinsame Instanz ist
$tS \; = \; \texttt{mal(s(s(null)), s(Z), s(Z))} \; = \; sS.$

Ein weiterer Unifikator für t und s ist

$$S' \; := \; \{\, \texttt{X} \mapsto \texttt{s(null)}, \;\; \texttt{Y} \mapsto \texttt{s(null)}, \;\; \texttt{Z} \mapsto \texttt{null} \,\}.$$

Die entstehende gemeinsame Instanz ist
$tS' \; = \; \texttt{mal(s(s(null)), s(null), s(null))} \; = \; sS'.$

Eine Ordnung auf den Substitutionen

Definition 5.24

Zwei Substitutionen S und T sind äquivalent (kurz: $S \equiv T$), wenn für alle Variablen $\mathtt{X} \in \mathrm{V_{LP}}$ gilt: $\mathtt{X}S = \mathtt{X}T$.

Beobachtung:

S und T sind genau dann äquivalent, wenn $\mathtt{X}S = \mathtt{X}T$ für alle $\mathtt{X} \in \mathrm{Def}(S) \cap \mathrm{Def}(T)$ und $\mathtt{X}S = \mathtt{X}$ für alle $\mathtt{X} \in \mathrm{Def}(S) \setminus \mathrm{Def}(T)$ und $\mathtt{X}T = \mathtt{X}$ für alle $\mathtt{X} \in \mathrm{Def}(T) \setminus \mathrm{Def}(S)$.

Definition 5.25

Seien S und T Substitutionen. S ist allgemeiner als T (wir schreiben $S \leqq T$), wenn es eine Substitution S' gibt, so dass $SS' \equiv T$.

Beobachtung:

I ist eine allgemeinste Substitution, d.h. für jede Substitution T gilt $I \leqq T$.

Allgemeinste Unifikatoren (kurz: mgu, für „most general unifier“)

Definition 5.26

Seien $t, s \in \mathrm{T}_{\mathrm{LP}}$. Ein allgemeinster Unifikator für t und s ist ein Unifikator S für t und s, so dass gilt: $S \leqq T$ für alle Unifikatoren T für t und s.

Das folgende Lemma besagt, dass allgemeinste Unifikatoren bis auf Umbenennung von Variablen eindeutig sind.

Lemma 5.27

Seien $t, s \in \mathrm{T}_{\mathrm{LP}}$, und seien S, T allgemeinste Unifikatoren für t und s. Dann gibt es eine Umbennenung U, so dass $SU \equiv T$.

Ein Unifikationsalgorithmus

Algorithmus $\mathrm{MGU}(t, s)$

% Eingabe: zwei Terme $t, s \in \mathsf{T}_{\mathsf{LP}}$.

% Ausgabe: eine Substitution S oder die Worte „nicht unifizierbar"

1. Wenn $t = s$, dann gib I aus und halte an.
2. Wenn $t = \mathtt{X} \in \mathsf{V}_{\mathsf{LP}}$
3. Wenn $\mathtt{X} \in \mathrm{Var}(s)$, dann gib „nicht unifizierbar" aus und halte an.
4. Gib $\{\mathtt{X} \mapsto s\}$ aus und halte an.
5. Wenn $s = \mathtt{X} \in \mathsf{V}_{\mathsf{LP}}$
6. Wenn $\mathtt{X} \in \mathrm{Var}(t)$ dann gib „nicht unifizierbar" aus und halte an.
7. Gib $\{\mathtt{X} \mapsto t\}$ aus und halte an.
8. Wenn $t = \mathtt{f}(t_1, \ldots, t_k)$ und $s = \mathtt{f}(s_1, \ldots, s_k)$
   für ein Atom $\mathtt{f} \in \mathsf{A}_{\mathsf{LP}}$ und eine Stelligkeit $k \in \mathbb{N}$ mit $k \geqslant 1$
9. Setze $S_1 := I$.
10. Für $i = 1, \ldots, k$ tue Folgendes:
11. Setze $T_i := \mathrm{MGU}(t_i S_i, s_i S_i)$.
12. Wenn $T_i =$ „nicht unifizierbar" dann gib „nicht unifizierbar"
    aus und halte an.
13. Setze $S_{i+1} := S_i T_i$.
14. Gib S_{k+1} aus und halte an.
15. Gib „nicht unifizierbar" aus und halte an.

Korrektheit des Unifikationsalgorithmus

Satz 5.28

Für alle Terme $t, s \in \mathsf{T}_{\mathsf{LP}}$ gilt:

(a) *Sind t und s unifizierbar, so gibt $\mathrm{MGU}(t, s)$ einen allgemeinsten Unifikator für t und s aus.*

(b) *Sind t und s nicht unifizierbar, so gibt $\mathrm{MGU}(t, s)$ die Worte „nicht unifizierbar" aus.*

(Hier ohne Beweis)

Korollar 5.29

Sind zwei Terme unifizierbar, so gibt es für diese Terme einen allgemeinsten Unifikator.

Beispiele 5.30

(a) Ein allgemeinster Unifikator für

$$t \ := \ \texttt{g(f(X,Y),f(V,W))} \qquad \text{und} \qquad s \ := \ \texttt{g(V,f(Z,g(X,Y)))}$$

ist

$$\begin{aligned} S \ &:= \ \{ \, \texttt{V} \mapsto \texttt{f(X,Y)}, \ \ \texttt{Z} \mapsto \texttt{f(X,Y)}, \ \ \texttt{W} \mapsto \texttt{g(X,Y)} \, \} \\ &= \ \{ \texttt{V} \mapsto \texttt{f(X,Y)} \} \ \{ \texttt{Z} \mapsto \texttt{f(X,Y)} \} \ \{ \texttt{W} \mapsto \texttt{g(X,Y)} \}, \end{aligned}$$

und es gilt $tS = sS = \texttt{g(f(X,Y),f(f(X,Y),g(X,Y)))}$.

(b) $\texttt{g(f(X,Y),Y)}$ und $\texttt{g(c,Y)}$ sind nicht unifizierbar.

(c) Seien $n \geqslant 1$ und seien $X_0, \ldots, X_n \in \mathrm{V}_{\mathrm{LP}}$ paarweise verschieden. Sei

$$\begin{aligned} t_n \ &:= \ \texttt{f(}X_1\texttt{,}X_2\texttt{...,}X_n\texttt{)} \\ s_n \ &:= \ \texttt{f(g(}X_0\texttt{,}X_0\texttt{),g(}X_1\texttt{,}X_1\texttt{),...,g(}X_{n-1}\texttt{,}X_{n-1}\texttt{))} \ . \end{aligned}$$

Dann sind t_n und s_n unifizierbar durch einen allgemeinsten Unifikator S, für den gilt: – siehe Tafel –

Es gilt: Für jeden Unifikator T für t_n und s_n ist der Term $T(X_n)$ exponentiell groß in n, und jede gemeinsame Instanz von t_n und s_n ist exponentiell lang in n.

Auflösen des Nichtdeterminismus in Zeile 4

Wir können nun den Nichtdeterminismus in Zeile 4 unseres einfachen Interpreters für Logikprogramme, $\textsc{Antwort}(\Pi, \alpha)$, auflösen, indem wir als Substitution T einen allgemeinsten Unifikator von α_i und φU wählen, und zwar den allgemeinsten Unifikator, der vom Algorithmus $\mathrm{MGU}(\alpha_i, \varphi U)$ ausgegeben wird.

Dadurch erhalten wir den folgenden Algorithmus $\mathrm{U}\textsc{Antwort}(\Pi, \alpha)$.

Interpreter für Logikprogramme mit allgemeinsten Unifikatoren

Algorithmus $\text{UANTWORT}(\Pi, \alpha)$

% Eingabe: Programm $\Pi \in \mathsf{LP}$, Anfrage ?- α $\in \mathsf{F_{LP}}$ mit $\alpha = \alpha_1, \ldots, \alpha_m$

% Ausgabe: eine Substitution $\tilde{S}$ für $\mathsf{Var}(\alpha)$ oder das Wort „gescheitert".

1. Wähle ein $i \in [m]$ *% α_i ist das nächste „Ziel"*
2. Wähle eine Regel ρ aus Π. Sei φ :- $\psi_1, \ldots, \psi_n$ die Form von ρ.
   % Fakten fassen wir als Regeln ohne Rumpf auf
3. Sei U eine Umbennung für $\mathsf{Var}(\rho)$, so dass $\mathsf{Var}(\rho U) \cap \mathsf{Var}(\alpha) = \emptyset$.
4. Setze $\tilde{T} := \text{MGU}(\alpha_i, \varphi U)$
   % $\tilde{T}$ soll ein allgemeinster Unifikator von α_i und φU sein
5. Wenn $\tilde{T}$ = „nicht unifizierbar", gib „gescheitert" aus und halte an.
6. Wenn $m = 1$ und $n = 0$, gib $\tilde{T}|_{\mathsf{Var}(\alpha)}$ aus und halte an.
7. Setze $\tilde{\alpha}' := \alpha_1 \tilde{T}, \ldots, \alpha_{i-1} \tilde{T}, \psi_1 U \tilde{T}, \ldots, \psi_n U \tilde{T}, \alpha_{i+1} \tilde{T}, \ldots, \alpha_m \tilde{T}$.
8. Setze $\tilde{T}' := \text{UANTWORT}(\Pi, \tilde{\alpha}')$
9. Wenn $\tilde{T}'$ eine Substitution ist, gib $(\tilde{T}\tilde{T}')|_{\mathsf{Var}(\alpha)}$ aus und halte an.
10. Gib „gescheitert" aus und halte an.

Korrektheit und Vollständigkeit des Interpreters

Satz 5.31

Sei $\Pi \in \mathsf{LP}$ ein Logikprogramm, sei $\texttt{?-}\ \alpha \in \mathsf{F}_{\mathsf{LP}}$ eine Anfrage mit $\alpha = \alpha_1, \ldots, \alpha_m$, und sei S eine Substitution für $\mathsf{Var}(\alpha)$. Dann sind folgende Aussagen äquivalent:

(a) *Die Terme $\alpha_1 S, \ldots, \alpha_m S$ sind aus Π ableitbar.*

(b) *Es gibt einen Lauf von $\textrm{UAntwort}(\Pi, \alpha)$, der eine Substitution $\tilde{S}$ für $\mathsf{Var}(\alpha)$ mit $\tilde{S} \leqq S$ ausgibt.*

Korollar 5.32

Sei $\Pi \in \mathsf{LP}$ ein Logikprogramm und sei α ein Grundterm. Dann gilt:
$\alpha \in \mathcal{B}(\Pi) \iff$ *es gibt einen akzeptierenden Lauf von $\textrm{UAntwort}(\Pi, \alpha)$.*

Für den Beweis der Richtung „*(a)* $\Longrightarrow$ *(b)*“ von Satz 5.31 verwenden wir:

Lemma 5.33

Sei $\Pi \in \mathsf{LP}$ und sei $\texttt{?-}\ \alpha \in \mathsf{F}_{\mathsf{LP}}$ mit $\alpha = \alpha_1, \ldots, \alpha_m \in \mathsf{F}_{\mathsf{LP}}$, und sei S' eine Substitution für α. Dann gibt es zu jedem Lauf von $\textrm{Antwort}(\Pi, \alpha S')$, der eine Substitution S ausgibt, einen Lauf von $\textrm{UAntwort}(\Pi, \alpha)$, der eine Substitution $\tilde{S}$ mit $\tilde{S} \leqq S'S$ ausgibt.

Bemerkungen

- Indem wir das nichtdeterministische Auswählen einer Substitution im Algorithmus ANTWORT im Algorithmus UANTWORT durch das deterministische Berechnen eines allgemeinsten Unifikators ersetzt haben, sind wir einen entscheidenden Schritt in Richtung „praktische Ausführbarkeit“ gegangen.

- Es bleiben aber immer noch die nichtdeterministischen Auswahlschritte eines Ziels in Zeile 1 und einer Regel in Zeile 2. Diese müssen bei einer praktischen Implementierung durch eine systematische Suche durch alle Möglichkeiten ersetzt werden.

  (Die Wahl der Umbennenung in Zeile 3 unproblematisch.)

- Verschiedene logische Programmiersprachen unterscheiden sich in den verwendeten Suchstrategien.

- Prolog verwendet Tiefensuche.

Abschnitt 5.4:

Logik-Programmierung und Prolog

Reines Prolog

Reines Prolog ist das Fragment der Programmiersprache Prolog, dessen Programme gerade die Logikprogramme in LP sind.

Insbesondere enthält reines Prolog keine speziellen Prolog-Operatoren wie Cut „!“, arithmetische Prädikate oder Ein-/Ausgabe-Prädikate (d.h. Prädikate mit Seiteneffekten).

Die Semantik von reinem Prolog stimmt nicht mit der deklarativen Semantik der Logik-Programmierung überein.

Die erste vom Prolog-Interpreter ausgegebene Antwort wird gemäß dem folgenden Interpreter PErsteAntwort ermittelt.

Ein Prolog-Interpreter

Algorithmus $\textsc{PErsteAntwort}(\Pi, \alpha)$

% Eingabe: Programm $\Pi \in$ LP, Anfrage ?- $\alpha \in \mathsf{F}_{\mathsf{LP}}$ mit $\alpha = \alpha_1, \ldots, \alpha_m$

% Ausgabe: eine Substitution S für Var(α) oder das Wort „false“

1. Betrachte alle Regeln ρ in Π in der Reihenfolge ihres Vorkommens in Π und tue Folgendes: *% Fakten fassen wir als Regeln ohne Rumpf auf*
2. Sei φ :- $\psi_1, \ldots, \psi_n$ die Form von ρ
3. Sei U eine Umbennung für Var(ρ), so dass Var(ρU) $\cap$ Var(α) $= \emptyset$
4. Setze $T := \mathrm{MGU}(\alpha_1, \varphi U)$
5. Wenn T eine Substitution ist
6. Wenn $m = 1$ und $n = 0$, gib $T|_{\mathsf{Var}(\alpha)}$ aus und halte an
7. Setze $\alpha' := \psi_1 UT, \ldots, \psi_n UT, \alpha_2 T, \ldots, \alpha_m T$
8. Setze $T' := \textsc{PErsteAntwort}(\Pi, \alpha')$
9. Wenn T' eine Substitution ist, gib $(TT')|_{\mathsf{Var}(\alpha)}$ aus und halte an
10. Gib „false“ aus und halte an

Vergleich zur deklarativen Semantik

$\textsc{PErsteAntwort}(\Pi, \alpha)$ gibt *höchstens eine* Substitution aus, kann u.U. aber auch in eine Endlosschleife gelangen und nicht terminieren.

Der folgende Satz besagt, dass im Falle der Terminierung die ausgegebene Antwort korrekt ist.

Satz 5.34

Sei $\Pi \in \mathsf{LP}$ ein Logikprogramm und sei ?- $\alpha \in \mathsf{F}_{\mathsf{LP}}$ mit $\alpha = \alpha_1, \ldots, \alpha_m$ eine Anfrage. Dann gilt:

(a) *Wenn $\textsc{PErsteAntwort}(\Pi, \alpha)$ eine Substitution S ausgibt, dann sind die Terme $\alpha_1 S, \ldots, \alpha_m S$ aus Π ableitbar.*

(b) *Wenn $\textsc{PErsteAntwort}(\Pi, \alpha)$ das Wort „false" ausgibt, dann gibt es keine Substitution S, so dass die Terme $\alpha_1 S, \ldots, \alpha_m S$ aus Π ableitbar sind.*

(Hier ohne Beweis)

Terminierung

Intuitiv besagt Satz 5.34, dass im Falle der Terminierung die vom Prolog-Interpreter bei Eingabe eines Logikprogramms Π und einer Anfrage `?-` α gegebene erste Antwort korrekt ist.

Möglicherweise hält der Prolog-Interpreter aber gar nicht an, obwohl es laut Definition der deklarativen Semantik korrekte Antworten gibt.

Es ist Aufgabe des Programmierers, dies zu verhindern!

Typische Probleme dabei sind Dummheit und linksrekursive Regeln.

Beispiel:

```
vorfahre(X,Y)  :-  vorfahre(X,Z), elternteil(Z,Y)
```

Unterschied zwischen Theorie und Praxis

Beispiel 5.35

Die folgenden Logikprogramme `myplus1.pl`, `myplus2.pl`, `myplus3.pl` haben die gleiche Bedeutung hinsichtlich der deklarativen Semantik im folgenden Sinne:

Aus allen drei Programmen können genau dieselben Grundterme der Form `myplus(...)` abgeleitet werden.

Alle drei Programme erzeugen jedoch unterschiedliche Ausgaben in Prolog.

Programm: `myplus1.pl`

```
myplus(X,Y,Z) :- myplus(Y,X,Z).
myplus(0,X,X).
                myplus(1,1,2).  myplus(1,2,3).  myplus(1,3,4).
                                myplus(2,2,4).  myplus(2,3,5).
                                                myplus(3,3,6).
```

Programm: `myplus2.pl`

```
myplus(0,X,X).
                myplus(1,1,2).  myplus(1,2,3).  myplus(1,3,4).
                                myplus(2,2,4).  myplus(2,3,5).
                                                myplus(3,3,6).
myplus(X,Y,Z) :- myplus(Y,X,Z).
```

Programm: `myplus3.pl`

```
myplusH(0,X,X).
                myplusH(1,1,2).  myplusH(1,2,3).  myplusH(1,3,4).
                                 myplusH(2,2,4).  myplusH(2,3,5).
                                                  myplusH(3,3,6).
myplus(X,Y,Z) :- myplusH(X,Y,Z).
myplus(X,Y,Z) :- myplusH(Y,X,Z).
```

Aus Sicht des Prolog-Interpreters (und des Interpreters PErsteAntwort) ist das Programm `myplus1.pl` idiotisch und liefert auf <u>keine</u> Anfrage der Form „`myplus(...)`“ eine Antwort, da die Auswertung des Programms stets mit der ersten Regel in eine Endlosschleife gerät.

Das Programm `myplus2.pl` ist besser, hält aber auch bei „falschen“ Anfragen wie z.B. „`myplus(1,1,3)`“ <u>nicht</u> an, da die Auswertung des Programms dann mit der letzten Regel in eine Endlosschleife gerät.

Das Programm `myplus3.pl` leistet das, was es soll.

Beweisbäume vs. Suchbäume

Beweisbäume

sind in Definition 5.14 definiert. Ein Beweisbaum ist eine graphische Darstellung einer Ableitung eines Terms $t \in \mathsf{T}_{\mathsf{LP}}$ aus einem Logikprogramm $\Pi \in \mathsf{LP}$.

Somit stellt ein Beweisbaum eine einzelne Ableitung dar. Diese entspricht einem erfolgreichen Lauf unseres nichtdeterministischen Interpreters ANTWORT.

Suchbäume

stellen die vollständige Suche des Prolog-Interpreters bei Eingabe eines Logikprogramms Π und einer Anfrage ?- α dar. Insbesondere enthält der Suchbaum Informationen über alle erfolgreichen Läufe des nichtdeterministischen Interpreters ANTWORT.

Unifikation in Prolog

In Prolog testet der Ausdruck $t = s$ testet nicht, ob die Terme t und s gleich sind, sondern ob sie unifizierbar sind.

Der in den meisten Prologimplementierungen verwendete Unifikationsalgorithmus testet aus Effizienzgründen bei der Unifikation einer Variablen X mit einem Term t nicht, ob X in t vorkommt.

Diesen Test bezeichnet man als Occurs-Check, er findet in den Zeilen 3 und 6 unseres Unifikationsalgorithmus MGU statt.

In Prolog ist es eine Aufgabe des Programmierers, sicherzustellen, dass niemals eine Variable mit einem Term unifiziert wird, der diese Variable enthält.