6.2 Operationale Semantik

Wir wollen nun die operationale Semantik von Prolog präzisieren, um genau zu sehen., wie ein Prolog-System in der Lage ist, die oben gezeigten Schlussfolgerungen zu ziehen. Prolog berechnet Schlussfolgerungen und beweist damit Aussagen mit Hilfe des sogenannten Resolutionsprinzips. Zunächst einmal betrachten wir dieses Prinzip in einer vereinfachten Form.

Definition 6.1 (**Vereinfachtes Resolutionsprinzip**). *Um ein Literal L zu beweisen, suche eine zu L passende Regel L :- $L_1, \ldots, L_n$ und beweise $L_1, \ldots, L_n$ (falls $n = 0$, d.h. die Regel ist ein Faktum, dann ist L bewiesen).*

Um dieses Prinzip zum Beweis von Literalen anzuwenden, müssen noch die folgenden Probleme gelöst werden:

1. Eine Regel passt eventuell nicht genau zu der Anfrage. Z.B. gibt es für das Anfrageliteral `vater(fritz,K)` keine direkt passende Regel oder Faktum, mit dem wir dies beweisen können.

2. Es passen eventuell mehrere Klauseln, um ein Literal zu beweisen. Prolog muss also mit Nichtdeterminismus umgehen können, wofür in Prolog immer eine automatische Suche eingebaut ist.

Für das erste Problem wird das vereinfachte Resolutionsprinzip um Unifikation erweitert. Hierbei bezeichnet **Unifikation** intuitiv das „Gleichmachen“ von Literalen oder Termen durch Substitution der Variablen in *beiden* Literalen/Termen. Dagegen ist das “pattern matching” aus funktionalen Sprachen nur die Anpassung *eines* Terms an einen anderen. Dies ist einer der wesentlichen Unterschiede zwischen funktionaler und logischer Programmierung: Beim Anwenden von Regeln wird in logischen Sprachen Unifikation statt Pattern Matching verwendet. Hierdurch wird das Finden von Lösungen und die bidirektionale Anwendung von Relationen ermöglicht. Wir wollen nun den Begriff des Unifikators formal definieren.

Definition 6.2 (Unifikator). *Ein Unifikator für zwei Terme t_1, t_2 ist eine Substitution σ mit $\sigma(t_1) = \sigma(t_2)$. In diesem Fall heißen t_1 und t_2* ***unifizierbar****.*
Ein ***allgemeinster Unifikator (most general unifier, mgu)*** *für t_1, t_2 ist ein Unifikator σ für t_1, t_2 mit: falls σ' auch ein Unifikator für t_1, t_2 ist, dann exisitiert eine Substitution φ mit $\sigma' = \varphi \circ \sigma$ („mgu subsumiert alle anderen Unifikatoren“).*

Beispiel 6.4 (Unifikatoren). Seien $t_1 = p(a, Y, Z)$ und $t_2 = p(X, b, T)$.
$\sigma_1 = \{X \mapsto a, Y \mapsto b, Z \mapsto c, T \mapsto c\}$ ist ein Unifikator, aber kein mgu.
$\sigma_2 = \{X \mapsto a, Y \mapsto b, Z \mapsto T\}$ ist ein mgu.

Bei der Verwendung von Unifikatoren stellen sich die folgenden Fragen:

- Existiert immer ein mgu für unifizierbare Terme?

- Wie können mgus berechnet werden?

Eine Antwort auf diese Fragen finden wir in (Robinson, 1965):

> Für unifizierbare Terme existiert immer ein mgu, der effektiv berechenbar ist.

6.2.1 Berechnung eines allgemeinsten Unifikators

Im Folgenden geben wir nicht den Algorithmus von Robinson an, sondern zeigen, wie ein mgu durch Transformation von Termgleichungen berechnet werden kann (diese Idee basiert auf (Martelli and Montanari, 1982)).

Definition 6.3 (mgu-Berechnung nach (Martelli and Montanari, 1982))**.** *Sei E eine Menge von Termgleichungen. Initial ist $E = \{t_1 = t_2\}$, falls wir t_1 und t_2 unifizieren wollen. Führe dann wiederholt die folgenden Transformation zur mgu-Berechnung durch (hier steht x für eine Variable):*

Eliminate

$$\{x = x\} \cup E \;\Rightarrow\; E$$

Decompose

$$\{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \cup E \;\Rightarrow\; \{s_1 = t_1, \ldots, s_n = t_n\} \cup E$$

Clash

$$\{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)\} \cup E \;\Rightarrow\; fail$$

falls $f \neq g$ oder $n \neq m$.

Swap

$$\{f(s_1, \ldots, s_n) = x\} \cup E \;\Rightarrow\; \{x = f(s_1, \ldots, s_n)\} \cup E$$

Replace

$$\{x = t\} \cup E \;\Rightarrow\; \{x = t\} \cup \sigma(E)$$

falls x eine Variable ist, die in E aber nicht in t vorkommt, und $\sigma = \{x \mapsto t\}$

Occur check

$$\{x = t\} \cup E \;\Rightarrow\; fail$$

falls x eine Variable ist, die in t vorkommt und $x \neq t$.

Die letzte Regel ist notwendig, um nicht unifizierbare Terme zu entdecken: Zum Beispiel ist $\{x = f(x)\}$ nicht unifizierbar. Mit der letzten Regel gilt: $\{x = f(x)\} \Rightarrow fail$. Über das Ergebnis des Algorithmus können wir die folgende Aussage treffen:

Satz 6.1. *Falls $\{t_1 = t_2\} \Rightarrow^* \{x_1 = s_1, \ldots, x_n = s_n\}$ gilt und keine weitere Regel anwendbar ist, dann ist $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ ein mgu für t_1 und t_2. Andernfalls gilt $\{t_1 = t_2\} \Rightarrow^* fail$ und dann sind t_1 und t_2 nicht unifizierbar.*

Hierbei wird durch die Notation $E \Rightarrow^* E'$ der reflexiv-transitive Abschluss von $\Rightarrow$ bezeichnet, der wie folgt definiert ist:

$$E \Rightarrow^* E' \;:\Leftrightarrow\; E \Rightarrow E_1 \Rightarrow E_2 \Rightarrow \cdots \Rightarrow E'$$

Beispiel 6.5 (mgu-Berechnung)**.** Sei $\{p(a, Y, Z) = p(X, b, X)\}$. Dann vollzieht der Algorithmus die folgenden Schritte:

- Decompose $\Rightarrow \{a = X, Y = b, Z = X\}$
- Swap $\Rightarrow \{X = a, Y = b, Z = X\}$
- Replace $\Rightarrow \{X = a, Y = b, Z = a\}$

Damit ist $\sigma = \{X \mapsto a, Y \mapsto b, Z \mapsto a\}$ ein mgu.

6.2.2 SLD-Resolution

Mit der Definition eines mgu können wir nun einen allgemeinen Resolutionsschritt definieren. Allerdings benötigen wir noch den Begriff einer Seleketionsregel:

Definition 6.4 (Selektionsregel)**.** *Eine Selektionsregel S ist eine Funktion, die aus einer nicht-leeren Anfrage ein Literal auswählt, d.h.*

$$S(\texttt{?-}\ L_1, \ldots, L_m\,.) \;=\; L_i$$

(falls $m > 0$) mit $1 \leq i \leq m$.

Z.B. verwendet Prolog die Selektionsregel $S(\texttt{?-}\ L_1, \ldots, L_m\,.) = L_1$, d. h. es wird immer das erste linke Literal ausgewählt.

Definition 6.5 (SLD-Resolutionsschritt)**.** *Sei S eine Selektionsregel und*

$\texttt{?-}\ L_1, \ldots, L_m\,.$

eine Anfrage und

$H\ \texttt{:-}\ B_1, \ldots, B_n\,.$

eine Klausel ($n \geq 0$), wobei die Selektionsregel S das Literal L_i auswählt und L_i und H mit einem mgu σ unifizierbar sind. Dann heißt die neue Anfrage

$\texttt{?-}\ \sigma(L_1, \ldots, L_{i-1}, B_1, \ldots, B_n, L_{i+1}, \ldots, L_m)\,.$

***Resolvente** aus der aktuellen Anfrage und der Klausel mit mgu σ.*

Der Begriff „SLD“ steht für folgende Eigenschaften:

S: Eine Selektionsregel S wählt ein L_i aus der Anfrage aus.

L: Lineare Resolution, d. h. verknüpfe immer eine Anfrage mit einer Klausel und erhalte eine neue Anfrage. Es gibt auch eine allgemeine Resolution, bei der auch Programmklauseln verknüpft werden, die eine allgemeinere Form haben können.

D: Definite Klauseln, d. h. die linke Seite enthält immer nur ein Literal.

Eine **SLD-Ableitung** ist eine Folge $A_1, A_2, \ldots$ von Anfragen, wobei A_{i+1} eine Resolvente aus A_i und einer **Variante** einer Programmklausel (mit neuen Variablen) ist. Das Betrachten von Varianten ist notwendig, denn sonst wäre bei einem Faktum „`p(X)`“ die Anfrage „`?- p(f(X)).`“ nicht beweisbar.

Eine SLD-Ableitung ist

- **erfolgreich**, wenn die letzte Anfrage leer ist (d.h. keine Literale enthält): dann ist alles bewiesen und die Substitution der Anfragevariablen ist die berechnete Lösung.
- **fehlgeschlagen**, wenn die letzte Anfrage nicht leer ist und keine weiteren Regeln anwendbar sind.
- **unendlich:**, wenn die SLD-Ableitung eine unendliche Folge ist. Dies entspricht einer Endlosschleife.

Beispiel 6.6 (SLD-Ableitung)**.**

```
?- grossvater(fritz,E).
   σ1 = {G ↦ fritz}
?- vater(fritz,V), vater(V,E).
   σ2 = {V ↦ thomas}
?- vater (thomas,E).
   σ3 = {E ↦ maria}
?- .
```

Damit ist die berechnete Lösung: `E = maria`.

Satz 6.2 (Korrektheit und Vollständigkeit der SLD-Resolution)**.** *Sei S eine beliebige Selektionsregel.*

1. ***Korrektheit:*** *Falls eine erfolgreiche SLD-Ableitung eine Substitution σ berechnet, dann ist σ eine „logisch korrekte“ Antwort.*
2. ***Vollständigkeit:*** *Falls σ eine „logisch korrekte“ Antwort für eine Anfrage ist, dann existiert(!) eine erfolgreiche SLD-Ableitung mit einer berechneten Antwort σ' und eine Substitution φ mit $\sigma = \varphi \circ \sigma'$, d. h. es werden unter Umständen allgemeinere Antworten berechnet.*

Problematisch hierbei ist, dass die Vollständigkeit nur eine Existenzaussage ist. Aber wie findet man die erfolgreichen SLD-Ableitungen unter allen SLD-Ableitungen? Eine

Möglichkeit ist es, alle möglichen SLD-Ableitungen zu untersuchen, aber in welcher Reihenfolge? Eine faire und sichere Strategie wäre, alle SLD-Ableitungen gleichzeitig zu untersuchen:

- parallel („ODER“-Parallelismus)
- Breitensuche im Baum aller Ableitungen (SLD-Baum)

Dies ist aber sehr aufwändig. Daher wird in Prolog auf diese sichere Strategie verzichtet. Stattdessen wird ein **Backtracking-Verfahren** verwendet, das einer Tiefensuche im SLD-Baum aller Ableitungen entspricht. Informell kann dieses Verfahren wie folgt beschrieben werden:

- Falls mehrere Klauseln anwendbar sind, nehme die textuell erste passende Klausel.
- Falls man in einer Sackgasse landet (wo keine Regel anwendbar ist), gehe zur letzten Alternative zurück und probiere die textuell nächste Klausel.

Ein Problem dieser Strategie ist, dass man eventuell keine Lösung bei existierenden endlosen Ableitungen erhält:

```
p :- p.
p.

?- p.   ⇒   ?- p.   ⇒   ?- p.   ⇒ ···
```

6.2.3 Formalisierung von Prologs Backtracking-Strategie

Wir wollen nun die soeben informell beschriebene Backtracking-Strategie präzise beschreiben. Dazu verwenden wir die folgenden Notationen:

- $l_1 \wedge \cdots \wedge l_n$: Anfrage mit Literalen $l_1, \ldots, l_n$
- *true*: leere Anfrage, hierbei entspricht ein Faktum einer Regel der Form `p :-` *true*.
- $+\!+$: Konkatenation auf Listen.
- [ ]: leere Liste.

Die Basissprache des Inferenzsystems für die Backtracking-Strategie enthält Aussagen der Form

$$cs, \sigma \vdash g : \overline{\sigma}$$

wobei gilt:

- cs ist eine Liste von Klauseln, die noch zum Beweis des ersten Literals von g benutzt werden können.
- σ ist die bisher berechnete Antwort.

- g ist die zu beweisende Anfrage.
- $\overline{\sigma}$ ist die Liste der berechneten Antworten für g.

Das Inferenzsystem besteht aus den folgenden Inferenzregeln. Hierbei bezeichnet P das gesamte Programm, also eine Folge von Klauseln.

Keine Klausel vorhanden

$$\frac{}{[\,], \sigma \vdash g : [\,]}$$

wobei $g \neq true$.

Anfrage bewiesen

$$\frac{}{cs, \sigma \vdash true : [\sigma]}$$

Literal bewiesen

$$\frac{cs, \sigma \vdash g : \overline{\sigma}}{cs, \sigma \vdash true \wedge g : \overline{\sigma}}$$

Anmerkung: auf Grund der anderen Regeln, insbesondere der Klauselanwendung, gilt hier die Invariante, dass die Klauseln cs immer das gesamte Programm (bzw. eine Variante davon) umfassen.

Klauselanwendung

$$\frac{P', \varphi \circ \sigma \vdash \varphi(b \wedge g) : \overline{\sigma_1} \qquad cs, \sigma \vdash l \wedge g : \overline{\sigma_2}}{[a \text{ :- } b] +\!+ cs, \sigma \vdash l \wedge g : \overline{\sigma_1} +\!+ \overline{\sigma_2}}$$

wobei φ ein mgu für a und l ist und P' ist eine Variante des Programms P (mit jeweils neuen Variablen).

Klausel nicht anwendbar

$$\frac{cs, \sigma \vdash l \wedge g : \overline{\sigma}}{[a \text{ :- } b] +\!+ cs, \sigma \vdash l \wedge g : \overline{\sigma}}$$

wobei a und l nicht unifizierbar sind.

Die Regel „Klauselanwendung“ formalisiert das „Backtracking“ bei einer Klauselanwendung: Es werden die Antworten mit der ersten Klausel berechnet und dahinter dann die Antworten mit den restlichen Klauseln.

Die Anwendung des Inferenzsystems für eine Anfrage G geschieht nach dem folgenden Muster: $\overline{\sigma}$ ist die Liste der berechneten Antworten für die Anfrage G, falls $P, \{\} \vdash G : \overline{\sigma}$ ableitbar ist.

Beispiel 6.7 (Backtracking)**.** Das Programm P enthalte folgende Klauseln:

```
p(a) :- true.
p(b) :- true.
q(X) :- p(X).
```

Dann ist $P, \{\} \vdash \texttt{q(Z)} : [\{\texttt{Z} \mapsto \texttt{a}\}, \{\texttt{Z} \mapsto \texttt{b}\}]$ ableitbar (wobei wir hier die Substitution der Variablen `X` weggelassen haben).

Probleme: Bezüglich dieses Inferenzsystems ist nichts ableitbar bei

- unendlich vielen Antworten
- endlich vielen Antworten gefolgt von einer unendlichen Berechnung

Eine mögliche Lösung dieser Probleme ist die Erweiterung des Inferenzsystems um einen Parameter n, sodass nur die maximal ersten n Antworten berechnet werden ($\rightarrow$ Übung).