# Logikprogrammierung

Jürgen Giesl

Sommersemester 2015

Lehr- und Forschungsgebiet Informatik 2

RWTH Aachen

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Für Informatiker ist die Kenntnis verschiedener Familien von Programmiersprachen aus mehreren Gründen nötig:

- Die Vertrautheit mit unterschiedlichen Konzepten von Programmiersprachen ermöglicht es, eigene Ideen bei der Entwicklung von Software besser auszudrücken.
- Das Hintergrundwissen über verschiedene Programmiersprachen ist nötig, um in konkreten Projekten jeweils die am besten geeignete Sprache auszuwählen.
- Wenn man bereits einige konzeptionell verschiedene Programmiersprachen erlernt hat, ist es sehr leicht, sich später weitere Programmiersprachen schnell anzueignen.
- Es ist auch die Aufgabe von Informatikern, neue Programmiersprachen zu entwerfen. Dies kann nur auf der Grundlage der bereits entwickelten Sprachen geschehen.

Generell unterscheidet man zwischen *imperativen* und *deklarativen* Programmiersprachen (wobei sich deklarative Sprachen weiter in funktionale und logische Sprachen unterteilen). In imperativen Sprachen setzen sich die Programme aus einer Folge von nacheinander ausgeführten Anweisungen zusammen, die die Werte der Variablen im Speicher verändern. Die meisten der heute verwendeten Programmiersprachen beruhen auf diesem Prinzip, das auch einen direkten Zusammenhang zu der klassischen Rechnerarchitektur besitzt, die auf John von Neumann zurückgeht.

In der deklarativen Programmierung bestehen die Programme hingegen aus einer Spezifikation dessen, *was* berechnet werden soll. Die Festlegung, *wie* die Berechnung genau verlaufen soll, wird dem Interpreter bzw. dem Compiler überlassen. Deklarative Programmiersprachen sind daher problemorientiert statt maschinenorientiert.

Einerseits sind die verschiedenen Programmiersprachen alle "gleichmächtig", d.h., jedes Programm lässt sich prinzipiell in jeder der üblicherweise verwendeten Sprachen schreiben. Andererseits sind die Sprachen aber unterschiedlich gut für verschiedene Anwendungsbereiche geeignet. So werden imperative Sprachen wie C beispielsweise für schnelle maschinennahe Programmierung eingesetzt, da dort der Programmierer direkt die Verwaltung des Speichers übernehmen kann (und muss). In anderen Programmiersprachen wird diese Aufgabe automatisch (vom Compiler oder vom Laufzeitsystem) durchgeführt. Dies erlaubt eine schnellere Programmentwicklung, die weniger fehleranfällig ist. Andererseits sind die dabei

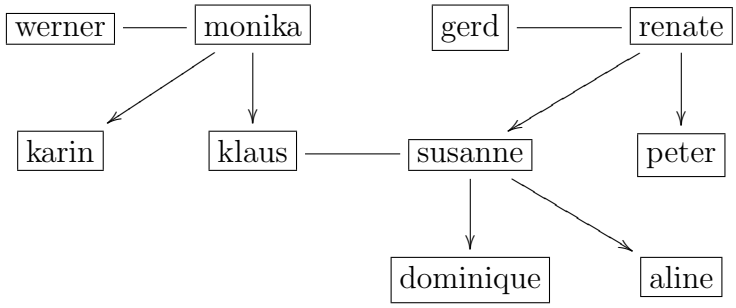
entstehenden Programme meist auch weniger effizient (d.h., sie benötigen mehr Zeit und Speicherplatz).

Während in der funktionalen Programmierung ein Programm eine Funktion realisiert (und auch in der imperativen Programmierung ein Programm implizit einer Funktion entspricht, die die Werte der Variablen verändert), bestehen logische Programme aus Regeln zur Definition von Relationen (d.h., ein logisches Programm ist eine Sammlung von Aussagen über den Zusammenhang zwischen verschiedenen Objekten). Bei der Ausführung des Programms werden diese Aussagen benutzt, um Anfragen zu beantworten und zu lösen.

Wir werden als Beispiel für die Logikprogrammierung die Sprache Prolog betrachten. Sie wird vor allem in der künstlichen Intelligenz eingesetzt (z.B. für Expertensysteme, Sprachverarbeitung, deduktive Datenbanken, etc.).

Im Folgenden werden die Grundprinzipien von Prolog kurz erklärt, um den Inhalt und die Struktur der Vorlesung zu motivieren. Es sind mehrere Implementierungen von Prolog verfügbar. Eine freie Implementierung, die wir empfehlen, ist SWI-Prolog, erhältlich unter `http://www.swi-prolog.org`.

Das Prinzip des logischen Programmierens beruht darauf, dass der Programmierer nur die logischen Zusammenhänge eines zu lösenden Problems beschreibt, d.h. die *Wissensbasis*. Hierzu ist kein Wissen über maschinennahe Details des Computers notwendig. Als Beispiel betrachten wir die folgende Wissensbasis über verwandtschaftliche Zusammenhänge.




Die Pfeile gehen hierbei von den Müttern zu ihren Kindern und waagerechte Striche deuten verheiratete Personen an. Hierbei gehen wir davon aus, dass es keine Scheidungen, keine unehelichen Kinder und keine gleichgeschlechtlichen Ehen gibt.

### Fakten und Anfragen

Zur Darstellung dieser Wissensbasis wird in Prolog die Sprache der *Prädikatenlogik* verwendet. In der Tat steht "Prolog" für "Programming in Logic". Die Wissensbasis besteht also aus lauter logischen Formeln (sogenannten *Klauseln*). Genauer gibt es zwei Arten von Klauseln in Programmen: *Fakten*, die Aussagen über Objekte treffen, und *Regeln*, die einem erlauben, aus bekannten Fakten auf neue Fakten zu schließen. In unserem Beispiel besteht die Wissensbasis (zunächst) nur aus Fakten:

```
weiblich(monika).
weiblich(karin).
weiblich(renate).
weiblich(susanne).
```

```
weiblich(aline).

maennlich(werner).
maennlich(klaus).
maennlich(gerd).
maennlich(peter).
maennlich(dominique).

verheiratet(werner, monika).
verheiratet(gerd, renate).
verheiratet(klaus, susanne).

mutterVon(monika, karin).
mutterVon(monika, klaus).
mutterVon(renate, susanne).
mutterVon(renate, peter).
mutterVon(susanne, aline).
mutterVon(susanne, dominique).
```

Ein Faktum besteht immer aus dem Namen der Eigenschaft oder Relation (dem sogenannten *Prädikatssymbol*) und direkt danach (ohne Leerzeichen) folgen die Objekte, die diese Eigenschaft haben. Danach kommt ein Punkt. Das nächste Faktum steht in der nächsten Zeile (oder zumindest kommt ein Leerzeichen vorher.) In Prolog beginnen Objekte (wie `monika`) und Eigenschaften (wie `mutterVon`) mit Kleinbuchstaben. Die Anzahl der Argumente wird als *Stelligkeit* des Prädikatssymbols bezeichnet. Ein Prädikat entspricht einer Funktion, deren Ergebnis "wahr" oder "falsch" ist. Man erkennt auch, dass die Relationen gerichtet sind. Wenn also ein Objekt 1 in einer Relation mit Objekt 2 steht, so gilt dies nicht notwendigerweise umgekehrt. Beispielsweise gilt `mutterVon(monika,karin)` aber nicht `mutterVon(karin,monika)`. Kommentare in Prolog beginnen mit einem `%` und gehen bis zum Zeilenende oder sie sind in `/*` und `*/` eingeschlossen.

Wie erwähnt, bedeutet die Ausführung eines logischen Programms, dass der Anwender *Anfragen* an das Programm stellt. Es handelt sich also um eine "dialogorientierte" Programmiersprache. Eine mögliche Frage wäre zum Beispiel: "Ist `gerd maennlich`?". In Prolog beginnen Anfragen mit "`?-`". Danach kommen die Aussagen (Formeln), die bewiesen werden sollen. Hierbei kann man auch mehrere Formeln auf einmal beweisen lassen, indem man die Formeln durch Kommas trennt. Zum Schluss der Anfrage kommt ein Punkt. In Prolog würde man also hierzu die Anfrage

```
?- maennlich(gerd).
```

eingeben. Prolog erzeugt die Antwort, indem es einen logischen Beweis auf Basis des vorhandenen Wissens (d.h. aufgrund der vom Programmierer gegebenen Problembeschreibung) durchführt. Die Aufgabe des Computers ist somit die Lösungsfindung, d.h., der Computer ist die *Inferenzmaschine* und "Rechnen" bedeutet bei Prolog "Beweisen". Diese Beweise werden mit Hilfe der Techniken "*Unifikation*" und "*Resolution*" durchgeführt. Bei der obigen Anfrage würde der Rechner mit `true` antworten. Auf die Anfrage

```
?- verheiratet(gerd, monika).
```

erhält man hingegen die Antwort `false`. Der Rechner geht nämlich davon aus, dass seine Wissensbasis das gesamte relevante Wissen über die Welt enthält. Falls also eine bestimmte Aussage nicht aus seinem Wissen *folgt*, so wird sie als *falsch* betrachtet. Aus diesem Grund wird die Aussage `verheiratet(gerd, monika)` also als falsch angesehen.

Um Anfragen an ein Programm stellen zu können, muss man das Programm natürlich vorher laden. Wenn das Programm in einer Datei mit dem Namen "`datei.pl`" steht, so gibt man im Prolog-Interpreter oder -Compiler hierzu "`consult(datei).`" ein (oder kurz "`[datei].`"). Danach steht das Wissen aus dieser Wissensbasis zur Verfügung. Die Datei muss hierbei mit einem Kleinbuchstaben anfangen.

### Variablen in Programmen

Die Wissensbasis kann auch *Variablen* enthalten. Hierbei beginnen Variablen in Prolog mit Großbuchstaben oder einem Unterstrich. Als Beispiel ergänzen wir die obige Wissensbasis um das folgende Faktum:

```
mensch(X).
```

Variablen in der Wissensbasis stehen für *alle* möglichen Belegungen, d.h., dieses Faktum bedeutet: "Alle Objekte sind Menschen". Stellt man nun die Anfrage

```
?- mensch(gerd).
```

so erhält man wie erwartet die Antwort `true`. Allerdings würde auch die Anfrage "`?- mensch(5).`" zum Ergebnis `true` führen.

Gleiche Variablen in gleichen Fakten bedeuten Gleichheit. Das Faktum "`mag(X,Y).`" bedeutet also: "Jeder mag jeden". Hingegen bedeutet "`mag(X,X).`": "Jeder mag sich selbst." Gleiche Variablen in verschiedenen Klauseln haben hingegen nichts miteinander zu tun.

### Variablen in Anfragen

Es ist auch möglich, das Programm selbst Lösungen berechnen zu lassen. Hierzu schreibt man Variablen in die Anfragen. Als Beispiel betrachten wir die Anfrage

```
?- mutterVon(X, susanne).
```

Dies entspricht der Frage "Wer ist die Mutter von `susanne`?" bzw. "Gibt es eine Belegung der Variablen `X`, so dass `X` die Mutter von `susanne` ist?". Der Rechner antwortet nun nicht (nur) mit `true`, sondern er sucht nach einer solchen Belegung von `X`. Die Antwort ist daher `X = renate`. Variablen in der Wissensbasis sind also *allquantifiziert* und Variablen in Anfragen sind *existenzquantifiziert*.

Als weiteres Beispiel betrachten wir die Anfrage

```
?- mutterVon(renate, Y).
```

d.h., die Frage "Welche Kinder hat Renate?". Der Rechner würde nun mit "`Y = susanne`" antworten.

Dies ist aber nicht die einzige mögliche Lösung. Will man weitere Lösungen berechnen lassen, so muss man ein Semikolon eingeben. Der Rechner sucht nun nach weiteren Lösungen und man erhält "`Y = peter`". Man erkennt also, dass `mutterVon` keine Funktion ist, bei der die Ein- und Ausgabe festliegt, sondern eine Relation. Was hierbei die Eingabe und was die Ausgabe ist, ist nicht festgelegt, sondern es hängt von der Anfrage ab. Man kann also ein- und dasselbe Programm benutzen, um zu einer Frau alle ihre Kinder berechnen zu lassen und um zu einem Kind seine Mutter berechnen zu lassen. Natürlich kann man auch "`?- mutterVon(X,Y).`" fragen. Es bleibt also dem Benutzer des Programms überlassen, welche Werte er bei der Anfrage vorgibt und welche Werte er vom Programm berechnen lassen möchte.

Zur Abarbeitung solcher Anfragen, durchsucht Prolog die Wissensbasis von vorne nach hinten und liefert die erste Antwort, die passt. *Die Klauseln der Wissensbasis werden also von oben nach unten bearbeitet.*

Als weiteres Beispiel betrachten wir die Anfrage

```
?- mensch(X).
```

Jede mögliche Instantiierung der Variablen `X` wäre jetzt eine Lösung. Der Rechner berechnet hier die "allgemeinste" Lösung `true`. Der Grund ist, dass die Aussage für jede Instantiierung von `X` wahr ist. Das Programm versucht stets, die *allgemeinsten* Antworten auf die Anfragen zu finden.

### Kombination von Fragen

Wie bereits erwähnt, kann man auch mehrere Aussagen gleichzeitig beweisen lassen (d.h., man kann Fragen kombinieren). Ein Beispiel ist die folgende Anfrage:

```
?- verheiratet(gerd,F), mutterVon(F,susanne).
```

Die Frage ist also, ob es eine Frau `F` gibt, die mit `gerd` verheiratet ist und auch Mutter von `susanne` ist. Die Variable `F` muss dabei in der ganzen Anfrage gleich instantiiert werden. Das Vorgehen von Prolog ist, zunächst das erste Ziel "`verheiratet(gerd,F)`" zu lösen. Dabei findet man eine Lösung für `F`. Mit dieser Lösung wird dann versucht, "`mutterVon(F,susanne)`" zu lösen. Falls dies nicht gelingt, setzt man zurück (*backtracking*) und versucht, eine andere Lösung für `F` zu finden, die auch "`verheiratet(gerd,F)`" erfüllt. *Aussagen in einer Anfrage werden also von links nach rechts bearbeitet.*

Als weiteres Beispiel betrachten wir die Frage, wer die Großmutter von `aline` ist:

```
?-  mutterVon(Oma,Mama), mutterVon(Mama,aline).
```

Hier wird erst `Oma = monika, Mama = karin` probiert. Dann wird zurückgesetzt, bis man schließlich die Lösung `Oma = renate, Mama = susanne` findet. Hätte man die beiden Fragen `mutterVon(Oma,Mama)` und `mutterVon(Mama,aline)` hingegen vertauscht, hätte man die Lösung schneller (ohne Zurücksetzen) gefunden.

### Regeln

Neben *Fakten* kann die Wissensbasis auch *Regeln* enthalten. Regeln dienen dazu, aus bekanntem Wissen neues Wissen herzuleiten. Als Beispiel betrachten wir die Vater-Kind-Beziehung. Diese Beziehung könnte man natürlich eigens für alle Objekte definieren, aber es ist wesentlich kürzer und übersichtlicher, dies mit einer allgemeinen Regel zu formulieren. (Insbesondere wäre dies bei unendlich vielen Objekten oder bei einer später wachsenden oder schrumpfenden Objektmenge gar nicht anders möglich.) Die folgende Regel besagt: "Eine Person `V` ist Vater eines Kindes `K`, falls er mit einer Frau `F` verheiratet ist, und diese Mutter des Kindes `K` ist."

```
vaterVon(V,K) :- verheiratet(V,F), mutterVon(F,K).
```

Das Zeichen "`:-`" bedeutet also "falls" und Regeln formulieren "wenn - dann" Beziehungen. Wenn die Voraussetzungen auf der rechten Seite der Regel wahr sind, dann ist auch die Aussage auf der linken Seite wahr. Die linke Seite heißt *Kopf* der Regel und die rechte Seite wird *Rumpf* bezeichnet. Die Voraussetzungen im Rumpf werden mit Kommas getrennt und zum Schluss kommt ein Punkt. Die Bedeutung einer Regel `p :- q,r.` ist: Wenn `q` und `r` gelten, dann gilt auch `p`.

Bei der Abarbeitung (d.h. bei Beweisen) in Prolog werden Regeln rückwärts angewendet. Um zu zeigen, dass die linke Seite einer Regel gilt, muss gezeigt werden, dass die rechte Seite gilt (*backward chaining*). Als Beispiel betrachten wir die Anfrage

```
?- vaterVon(gerd,susanne).
```

Um diese Aussage zu beweisen, muss man aufgrund der Regel für `vaterVon` ein `F` finden, so dass die Aussagen `verheiratet(gerd,F), mutterVon(F,susanne)` wahr sind. Dies entspricht also gerade der Anfrage `?- verheiratet(gerd,F), mutterVon(F,susanne)`. Prolog antwortet daher mit "`true`". Analog dazu würde die Anfrage

```
?- vaterVon(gerd,Y).
```

die Antworten `Y = susanne` und `Y = peter` ergeben.

### Mehrere Regeln für ein Prädikat

Bisher haben wir eine Regel definiert, bei der der Kopf gilt, wenn die *Konjunktion* der Voraussetzungen gilt. Nun wollen wir auch den Fall betrachten, dass der Kopf aus der *Disjunktion* von zwei Voraussetzungen folgt. Hierzu verwenden wir mehrere Regeln für dasselbe Prädikatssymbol. Als Beispiel betrachten wir ein Prädikat `elternteil`, wobei `elternteil(X,Y)` wahr sein soll, wenn `X` Mutter oder Vater von `Y` ist. Die Regeln hierzu sind folgende:

```
elternteil(X,Y) :- mutterVon(X,Y).
elternteil(X,Y) :- vaterVon(X,Y).
```

Stellt man nun die Anfrage

```
?- elternteil(X, susanne).
```

so ergeben sich die Antworten `X = renate` und `X = gerd`. Man erkennt, dass die Reihenfolge der Klauseln auch die Suche und die Reihenfolge der Lösungen beeinflusst.

### Rekursive Regeln

*Rekursion* ist eine wichtige Programmiertechnik in Prolog. Als Beispiel definieren wir ein Prädikat `vorfahre`. Hierbei ist `V` Vorfahre von `X`, wenn `V` ein Elternteil von `X` ist oder wenn es ein `Y` gibt, so dass `V` Elternteil von `Y` ist (d.h. `V` hat ein Kind `Y`) und `Y` Vorfahre von `X` ist. Die Übersetzung dieser Regel in Prolog ergibt:

```
vorfahre(V,X) :- elternteil(V,X).
vorfahre(V,X) :- elternteil(V,Y), vorfahre(Y,X).
```

Die Anfrage

```
?- vorfahre(X, aline).
```

bedeutet nun: "Wer sind die Vorfahren von `aline`?". Hierzu findet Prolog die folgenden Antworten:

```
X = susanne;
X = klaus;
X = monika;
X = renate;
X = werner;
X = gerd
```

### Kennzeichen logischer Programme

Insgesamt ergeben sich die folgenden Eigenschaften logischer Programme:

1. (Rein) logische Programme besitzen keine Kontrollstrukturen zur Steuerung des Programmablaufs. Die Programme sind lediglich Sammlungen von Fakten und Regeln, die von oben nach unten (bzw. von links nach rechts) abgearbeitet werden.

2. Das logische Programmieren ist aus dem automatischen Beweisen entstanden und bei der Ausführung eines logischen Programms wird versucht, eine Anfrage zu beweisen. Dabei werden auch Lösungen für Variablen in der Anfrage berechnet. Dies bedeutet, dass bei einem logischen Programm Ein- und Ausgabevariablen nicht festliegen.

3. Logische Programme sind besonders gut für Anwendungen der künstlichen Intelligenz geeignet. Beispielsweise lassen sich damit sehr gut Expertensysteme realisieren, wobei die Regeln des Programms aus dem Wissen der Experten gebildet werden. Weitere Hauptanwendungsgebiete sind deduktive Datenbanken und das Rapid Prototyping.

Der Aufbau der Vorlesung ist wie folgt: Da Logikprogramme aus prädikatenlogischen Formeln bestehen und da zur Ausführung logischer Programme prädikatenlogische Beweise geführt werden, werden in Kapitel 2 die benötigten Grundlagen der Prädikatenlogik eingeführt und in Kapitel 3 führen wir das in der Logikprogrammierung verwendete Beweisprinzip der *Resolution* ein. In Kapitel 4 betrachten wir anschließend die Syntax, Semantik und Ausdrucksstärke von (reinen) Logikprogrammen wie oben und gehen auf die Strategie zur Ausführung von Logikprogrammen ein. In Kapitel 5 behandeln wir dann die Programmiersprache Prolog und betrachten insbesondere die Eigenschaften von Prolog, die über die

reinen Logikprogramme hinausgehen. Schließlich stellen wir in Kapitel 6 eine Erweiterung von Logikprogrammen um Constraints vor.

Ich danke Peter Schneider-Kamp und René Thiemann für ihre konstruktiven Kommentare und Vorschläge beim Korrekturlesen des Skripts.

# Kapitel 2

# Grundlagen der Prädikatenlogik

In diesem Kapitel werden wir die Sprache der Prädikatenlogik 1. Stufe einführen, die zur Formulierung von Logikprogrammen verwendet wird. Hierzu rekapitulieren wir in den Abschnitten 2.1 und 2.2 die Syntax und Semantik der Prädikatenlogik, insbesondere auch, um die im Folgenden verwendeten Notationen einzuführen. Zur Abarbeitung eines Logikprogramms muss untersucht werden, ob eine Formel (die *Anfrage*) aus einer Menge von Formeln (den *Fakten* und *Regeln* des *Programms*) folgt.

## 2.1 Syntax der Prädikatenlogik

Die *Syntax* legt fest, aus welchen Zeichenreihen die Ausdrücke einer Sprache bestehen. Zunächst definieren wir das Alphabet, aus dem Ausdrücke der Prädikatenlogik gebildet werden.

**Definition 2.1.1 (Signatur)** *Eine* Signatur $(\Sigma, \Delta)$ *ist ein Paar mit* $\Sigma = \bigcup_{n\in\mathbb{N}} \Sigma_n$ *und* $\Delta = \bigcup_{n\in\mathbb{N}} \Delta_n$. $\Sigma$ *und* $\Delta$ *sind hierbei die Vereinigung von paarweise disjunkten Mengen* $\Sigma_n$ *und* $\Delta_n$. *Jedes* $f \in \Sigma_n$ *heißt* $n$*-stelliges* Funktionssymbol *und jedes* $p \in \Delta_n$ *heißt* $n$*-stelliges* Prädikatssymbol. *Die Elemente von* $\Sigma_0$ *werden auch* Konstanten *genannt. Wir verlangen hierbei stets* $\Sigma_0 \neq \varnothing$.

**Beispiel 2.1.2** *Als Beispiel betrachten wir die folgende Signatur* $(\Sigma, \Delta)$ *mit* $\Sigma = \Sigma_0 \cup \Sigma_3$ *und* $\Delta = \Delta_1 \cup \Delta_2$. *Sie entspricht der Signatur des Logikprogramms aus Kapitel 1, wobei* $\Sigma$ *zusätzlich das dreistellige Funktionssymbol* datum *und* $\Delta$ *das zweistellige Prädikatssymbol* geboren *enthält.*

$$
\begin{aligned}
\Sigma_0 &= \mathbb{N} \cup \{\mathsf{monika}, \mathsf{karin}, \mathsf{renate}, \mathsf{susanne}, \mathsf{aline}, \mathsf{werner}, \mathsf{klaus}, \mathsf{gerd}, \mathsf{peter}, \mathsf{dominique}\} \\
\Sigma_3 &= \{\mathsf{datum}\} \\
\Delta_1 &= \{\mathsf{weiblich}, \mathsf{maennlich}, \mathsf{mensch}\} \\
\Delta_2 &= \{\mathsf{verheiratet}, \mathsf{mutterVon}, \mathsf{vaterVon}, \mathsf{elternteil}, \mathsf{vorfahre}, \mathsf{geboren}\}
\end{aligned}
$$

Nun definieren wir, wie Datenobjekte in der Sprache der Prädikatenlogik repräsentiert werden.

**Definition 2.1.3 (Terme)** *Sei $(\Sigma, \Delta)$ eine Signatur und $\mathcal{V}$ eine Menge von* Variablen*, so dass $\mathcal{V} \cap \Sigma = \varnothing$ gilt. Dann bezeichnet $\mathcal{T}(\Sigma, \mathcal{V})$ die Menge aller* Terme *(über $\Sigma$ und $\mathcal{V}$). Hierbei ist $\mathcal{T}(\Sigma, \mathcal{V})$ die kleinste Menge mit*

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ *und*
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$*, falls $f \in \Sigma_n$ und $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ für ein $n \in \mathbb{N}$.*

*$\mathcal{T}(\Sigma)$ steht für $\mathcal{T}(\Sigma, \varnothing)$, d.h., für die Menge aller variablenfreien Terme (oder* Grundterme*). Für einen Term $t$ ist $\mathcal{V}(t)$ die Menge aller Variablen in $t$.*

Um Variablen von Funktions- und Prädikatssymbolen (insbesondere von Konstanten) unterscheiden zu können, verwenden wir (wie in Prolog) die Konventionen, dass Variablen mit Großbuchstaben und Funktions- und Prädikatssymbole mit Kleinbuchstaben beginnen.

**Beispiel 2.1.4** *Wir betrachten wieder die Signatur $\Sigma$ aus Bsp. 2.1.2. Wenn $\mathcal{V} = \{X, Y, Z, Mama, Oma, \ldots\}$ ist, dann erhalten wir z. B. die folgenden Terme in $\mathcal{T}(\Sigma, \mathcal{V})$:* $\mathsf{monika}$, $\mathsf{42}$, $\mathsf{datum}(\mathsf{15}, \mathsf{10}, \mathsf{1966})$, $X$, $\mathsf{datum}(X, Oma, \mathsf{datum}(Y, \mathsf{monika}, \mathsf{101}))$, *...*

Nun können wir festlegen, wie Aussagen in der Sprache der Prädikatenlogik gebildet werden.

**Definition 2.1.5 (Formeln)** *Sei $(\Sigma, \Delta)$ eine Signatur und $\mathcal{V}$ eine Menge von Variablen. Die Menge der* atomaren Formeln *über $(\Sigma, \Delta)$ und $\mathcal{V}$ ist definiert als $\mathcal{A}t(\Sigma, \Delta, \mathcal{V}) = \{p(t_1, \ldots, t_n) \mid p \in \Delta_n$ für ein $n$, $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}$.*

*$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ bezeichnet die Menge aller* Formeln *über $(\Sigma, \Delta)$ und $\mathcal{V}$. Hierbei ist $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ die kleinste Menge mit*

- $\mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- *wenn $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, dann $\neg\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$*
- *wenn $\varphi_1, \varphi_2 \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, dann $(\varphi_1 \wedge \varphi_2), (\varphi_1 \vee \varphi_2), (\varphi_1 \to \varphi_2), (\varphi_1 \leftrightarrow \varphi_2) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$*
- *wenn $X \in \mathcal{V}$ und $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, dann $(\forall X\ \varphi), (\exists X\ \varphi) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$*

*Für eine Formel $\varphi$ bezeichnet $\mathcal{V}(\varphi)$ die Menge aller Variablen in $\varphi$. Eine Variable $X$ ist* frei *in einer Formel $\varphi$ gdw.*

- *$\varphi$ ist eine atomare Formel und $X \in \mathcal{V}(\varphi)$ oder*
- *$\varphi = \neg\varphi_1$ und $X$ ist frei in $\varphi_1$ oder*
- *$\varphi = (\varphi_1 \cdot \varphi_2)$ mit $\cdot \in \{\wedge, \vee, \to, \leftrightarrow\}$ und $X$ ist frei in $\varphi_1$ oder in $\varphi_2$ oder*
- *$\varphi = (Q\, Y\ \varphi_1)$ mit $Q \in \{\forall, \exists\}$, $X$ ist frei in $\varphi_1$ und $X \neq Y$.*

*Eine Formel $\varphi$ heißt* geschlossen *gdw. es keine freie Variable in $\varphi$ gibt. Eine Formel heißt* quantorfrei*, wenn sie nicht die Zeichen $\forall$ oder $\exists$ enthält.*

$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ ist also eine formale Sprache, in der mathematische Sachverhalte, aber auch Fakten des täglichen Lebens formuliert werden können. Im Folgenden lassen wir überflüssige Klammern meist weg (d.h., wir schreiben "$\forall X\ \mathsf{mensch}(X)$" statt "$(\forall X\ \mathsf{mensch}(X))$").

**Beispiel 2.1.6** *Sei* $(\Sigma, \Delta)$ *wieder die Signatur aus Bsp. 2.1.2. Dann kann man z. B. Formeln wie folgt bilden.*

$$\begin{aligned}
\mathsf{weiblich}(\mathsf{monika}) &\in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) && (2.1)\\
\mathsf{mutterVon}(X, \mathsf{susanne}) &\in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) && (2.2)\\
\mathsf{geboren}(\mathsf{monika}, \mathsf{datum}(15, 10, 1966)) &\in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) && (2.3)\\
\forall F\ (\mathsf{verheiratet}(\mathsf{gerd}, F) \wedge \mathsf{mutterVon}(F, K)) &\in \mathcal{F}(\Sigma, \Delta, \mathcal{V}) && (2.4)\\
\mathsf{verheiratet}(\mathsf{gerd}, F) \wedge \neg(\forall F\ \mathsf{mutterVon}(F, K)) &\in \mathcal{F}(\Sigma, \Delta, \mathcal{V}). && (2.5)
\end{aligned}$$

*In der Formel aus Zeile (2.2) ist die Variable $X$ frei. Für die Formel $\varphi$ in Zeile (2.4) gilt $\mathcal{V}(\varphi) = \{F, K\}$, aber nur die Variable $K$ ist hier frei. In der Formel der Zeile (2.5) ist hingegen sowohl $F$ als auch $K$ frei.*

Wir kürzen Formeln der Art "$\forall X_1\ (\ldots(\forall X_n\ \varphi)\ldots)$" oft durch "$\forall X_1, \ldots, X_n\ \varphi$" ab. Analog dazu ist "$\exists X_1, \ldots, X_n\ \varphi$" eine Abkürzung für "$\exists X_1\ (\ldots(\exists X_n\ \varphi)\ldots)$".

**Beispiel 2.1.7** *Das Logikprogramm aus Kapitel 1 entspricht der folgenden Formelmenge über der Signatur von Bsp. 2.1.2. Wie erwähnt sind Variablen im Logikprogramm hierbei* allquantifiziert*:*

$$\begin{aligned}
& && \mathsf{weiblich}(\mathsf{monika})\\
& && \mathsf{weiblich}(\mathsf{karin})\\
& && \mathsf{weiblich}(\mathsf{renate})\\
& && \mathsf{weiblich}(\mathsf{susanne})\\
& && \mathsf{weiblich}(\mathsf{aline})\\
& && \\
& && \mathsf{maennlich}(\mathsf{werner})\\
& && \mathsf{maennlich}(\mathsf{klaus})\\
& && \mathsf{maennlich}(\mathsf{gerd})\\
& && \mathsf{maennlich}(\mathsf{peter})\\
& && \mathsf{maennlich}(\mathsf{dominique})\\
& && \\
& && \mathsf{verheiratet}(\mathsf{werner}, \mathsf{monika})\\
& && \mathsf{verheiratet}(\mathsf{gerd}, \mathsf{renate})\\
& && \mathsf{verheiratet}(\mathsf{klaus}, \mathsf{susanne})\\
& && \\
& && \mathsf{mutterVon}(\mathsf{monika}, \mathsf{karin})\\
& && \mathsf{mutterVon}(\mathsf{monika}, \mathsf{klaus})\\
& && \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne})\\
& && \mathsf{mutterVon}(\mathsf{renate}, \mathsf{peter})\\
& && \mathsf{mutterVon}(\mathsf{susanne}, \mathsf{aline})\\
& && \mathsf{mutterVon}(\mathsf{susanne}, \mathsf{dominique})\\
& && \\
\forall X & && \mathsf{mensch}(X)\\
& && \\
\forall V, F, K & \mathsf{verheiratet}(V, F) \wedge \mathsf{mutterVon}(F, K) & \rightarrow & \mathsf{vaterVon}(V, K)
\end{aligned}$$

$$
\begin{array}{lrcl}
\forall X, Y & \mathsf{mutterVon}(X, Y) & \rightarrow & \mathsf{elternteil}(X, Y) \\
\forall X, Y & \mathsf{vaterVon}(X, Y) & \rightarrow & \mathsf{elternteil}(X, Y) \\
 & & & \\
\forall V, X & \mathsf{elternteil}(V, X) & \rightarrow & \mathsf{vorfahre}(V, X) \\
\forall V, Y, X & \mathsf{elternteil}(V, Y) \wedge \mathsf{vorfahre}(Y, X) & \rightarrow & \mathsf{vorfahre}(V, X)
\end{array}
$$

Wir führen schließlich noch den Begriff der Substitution ein. Substitutionen erlauben die Ersetzung (oder "Instantiierung") von Variablen durch Terme.

**Definition 2.1.8 (Substitution)** *Eine Abbildung $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ heißt* Substitution *gdw. $\sigma(X) \neq X$ nur für endlich viele $X \in \mathcal{V}$ gilt. $DOM(\sigma) = \{X \in \mathcal{V} \mid \sigma(X) \neq X\}$ heißt der* Domain *von $\sigma$ und $RANGE(\sigma) = \{\sigma(X) \mid X \in DOM(\sigma)\}$ heißt der* Range *von $\sigma$. Da $DOM(\sigma)$ endlich ist, ist eine Substitution $\sigma$ als die endliche Menge von Paaren $\{X/\sigma(X) \mid X \in DOM(\sigma)\}$ darstellbar. Eine Substitution $\sigma$ ist eine* Grundsubstitution *auf $DOM(\sigma)$ gdw. $\mathcal{V}(\sigma(X)) = \varnothing$ für alle $X \in DOM(\sigma)$.*[1] *Eine Substitution $\sigma$ ist eine* Variablenumbenennung *gdw. $\sigma$ injektiv ist und $\sigma(X) \in \mathcal{V}$ für alle $X \in \mathcal{V}$ gilt.*

*Substitutionen werden homomorph zu Abbildungen $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$ erweitert, d.h. $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. Analog kann man Substitutionen auch auf Formeln erweitern:*

*(1)* $\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$

*(2)* $\sigma(\neg\varphi) = \neg\sigma(\varphi)$

*(3)* $\sigma(\varphi_1 \cdot \varphi_2) = \sigma(\varphi_1) \cdot \sigma(\varphi_2)$ *für* $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$

*(4)* $\sigma(Q\, X\, \varphi) = Q\, X\, \sigma(\varphi)$, *für* $Q \in \{\forall, \exists\}$, *falls* $X \notin \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma)$

*(5)* $\sigma(Q\, X\, \varphi) = Q\, X'\, \sigma(\delta(\varphi))$, *für* $Q \in \{\forall, \exists\}$, *falls* $X \in \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma)$. *Hierbei ist $X'$ eine neue Variable mit $X' \notin \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma) \cup \mathcal{V}(\varphi)$ und $\delta = \{X/X'\}$.*

*Die Bedingung in Fall (4) bedeutet, dass die Anwendung von $\sigma$ auf die Formel $\forall X\ \varphi$ nur dann direkt möglich ist, wenn die durch den Quantor gebundene Variable $X$ nicht ersetzt wird und wenn dadurch keine andere Variable in $\varphi$ durch einen Term ersetzt wird, der $X$ enthält. Anderenfalls muss (im Fall (5)) zunächst $X$ in eine neue Variable $X'$ umbenannt werden und anschließend wird die Substitution $\sigma$ angewendet. Solch eine Umbenennung von quantifizierten Variablen bezeichnet man als* gebundene Umbenennung.

*Eine Instanz $\sigma(t)$ eines Terms $t$ (bzw. eine Instanz $\sigma(\varphi)$ einer quantorfreien Formel $\varphi$) bezeichnet man als* Grundinstanz, *falls $\mathcal{V}(\sigma(t)) = \varnothing$ (bzw. $\mathcal{V}(\sigma(\varphi)) = \varnothing$) ist.*

[1] Wir sprechen oft nur von einer "Grundsubstitution" und gehen davon aus, dass $DOM(\sigma)$ groß genug gewählt ist, um alle jeweils benötigten Variablen auf Grundterme abzubilden.

**Beispiel 2.1.9** *Wir betrachten wieder die Signatur von Bsp. 2.1.2. Ein Beispiel für eine Substitution wäre dann* $\sigma = \{X/\mathsf{datum}(X,Y,Z), Y/\mathsf{monika}, Z/\mathsf{datum}(Z,Z,Z)\}$. *Man erhält dann*

$$\begin{aligned} \sigma(\mathsf{datum}(X,Y,Z)) &= \mathsf{datum}(\mathsf{datum}(X,Y,Z), \mathsf{monika}, \mathsf{datum}(Z,Z,Z)) \\ \sigma(\forall Y\ \mathsf{verheiratet}(X,Y)) &= \forall Y'\ \mathsf{verheiratet}(\mathsf{datum}(X,Y,Z), Y') \end{aligned}$$

*Wir schreiben auch statt "*$\sigma(\mathsf{datum}(X,Y,Z))$*" oft*

$$\mathsf{datum}(X,Y,Z)\ [X/\mathsf{datum}(X,Y,Z),\ Y/\mathsf{monika},\ Z/\mathsf{datum}(Z,Z,Z)].$$

## 2.2 Semantik der Prädikatenlogik

Unser Ziel ist, mit Formeln aus $\mathcal{F}(\Sigma,\Delta,\mathcal{V})$ Aussagen über die Welt formal zu repräsentieren. Bislang haben wir nur festgelegt, wie Formeln aus $\mathcal{F}(\Sigma,\Delta,\mathcal{V})$ gebildet werden. Wir haben jedoch noch nicht definiert, was solche Formeln eigentlich *aussagen*. Dazu muss man den Formeln eine *Semantik*, d.h. eine Bedeutung, zuordnen. Dann kann man auch festlegen, welche Formeln (d.h. welche *Anfragen*) aus einer gegebenen Formelmenge (d.h. aus einem *Logikprogramm*) folgen.

Zur Definition der Semantik verwendet man sogenannte *Interpretationen*, die eine Menge von Objekten $\mathcal{A}$ festlegen, jedem (syntaktischen) Funktionssymbol $f$ eine Funktion $\alpha_f$, jedem (syntaktischen) Prädikatssymbol $p$ eine Relation (oder Teilmenge) $\alpha_p$ und jeder Variablen $X \in \mathcal{V}$ ein Objekt aus $\mathcal{A}$ zuordnen. Eine Interpretation bildet jeden Term auf ein Objekt aus $\mathcal{A}$ ab und jede Formel ist bezüglich einer Interpretation wahr oder falsch.

**Definition 2.2.1 (Interpretation, Struktur, Erfüllbarkeit, Modell)** *Für eine Signatur* $(\Sigma,\Delta)$ *ist eine* Interpretation *ein Tripel* $I = (\mathcal{A},\alpha,\beta)$. *Die Menge* $\mathcal{A}$ *heißt der* Träger *der Interpretation, wobei wir* $\mathcal{A} \neq \varnothing$ *verlangen. Weiter ist* $\alpha$ *eine Abbildung, die jedem Funktionssymbol* $f \in \Sigma_n$ *eine Funktion* $\alpha_f : \mathcal{A}^n \to \mathcal{A}$ *und jedem Prädikatssymbol* $p \in \Delta_n$ *mit* $n \geq 1$ *eine Menge (bzw. Relation)* $\alpha_p \subseteq \mathcal{A}^n$ *zuordnet. Für* $p \in \Delta_0$ *gilt* $\alpha_p \in \{TRUE, FALSE\}$. *Die Funktion* $\alpha_f$ *bzw. die Relation* $\alpha_p$ *heißen die* Deutung *des Funktionssymbols* $f$ *bzw. des Prädikatssymbols* $p$ *unter der Interpretation* $I$. *Die Abbildung* $\beta : \mathcal{V} \to \mathcal{A}$ *heißt* Variablenbelegung *für die Interpretation* $I$.

*Zu jeder Interpretation* $I$ *erhält man eine Funktion* $I : \mathcal{T}(\Sigma,\mathcal{V}) \to \mathcal{A}$ *wie folgt:*

$$\begin{aligned} I(X) &= \beta(X) \text{ für alle } X \in \mathcal{V} \\ I(f(t_1,\ldots,t_n)) &= \alpha_f(I(t_1),\ldots,I(t_n)) \text{ für alle } f \in \Sigma_n \text{ und } t_1,\ldots,t_n \in \mathcal{T}(\Sigma,\mathcal{V}) \end{aligned}$$

*Man nennt* $I(t)$ *die* Interpretation des Terms $t$ *unter der Interpretation* $I$.

*Für* $X \in \mathcal{V}$ *und* $\mathbf{a} \in \mathcal{A}$ *ist* $\beta[\![X/\mathbf{a}]\!]$ *die Variablenbelegung mit* $\beta[\![X/\mathbf{a}]\!](X) = \mathbf{a}$ *und* $\beta[\![X/\mathbf{a}]\!](Y) = \beta(Y)$ *für alle* $Y \in \mathcal{V}$ *mit* $Y \neq X$. $I[\![X/\mathbf{a}]\!]$ *bezeichnet die Interpretation* $(\mathcal{A},\alpha,\beta[\![X/\mathbf{a}]\!])$.

*Eine Interpretation* $I = (\mathcal{A}, \alpha, \beta)$ erfüllt *eine Formel* $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$*, geschrieben "*$I \models \varphi$*", gdw.*

| | | | |
|---|---|---|---|
| | $\varphi$ | $= p(t_1, \ldots, t_n)$ | *und* $p \in \Delta_n$ *mit* $(I(t_1), \ldots, I(t_n)) \in \alpha_p$ *für* $n \geq 1$ |
| *oder* | $\varphi$ | $= p$ | *und* $p \in \Delta_0$ *mit* $\alpha_p = TRUE$ |
| *oder* | $\varphi$ | $= \neg\varphi_1$ | *und* $I \not\models \varphi_1$ |
| *oder* | $\varphi$ | $= \varphi_1 \wedge \varphi_2$ | *und* $I \models \varphi_1$ *und* $I \models \varphi_2$ |
| *oder* | $\varphi$ | $= \varphi_1 \vee \varphi_2$ | *und* $I \models \varphi_1$ *oder* $I \models \varphi_2$ |
| *oder* | $\varphi$ | $= \varphi_1 \rightarrow \varphi_2$ | *und falls* $I \models \varphi_1$*, dann auch* $I \models \varphi_2$ |
| *oder* | $\varphi$ | $= \varphi_1 \leftrightarrow \varphi_2$ | *und* $I \models \varphi_1$ *gdw.* $I \models \varphi_2$ |
| *oder* | $\varphi$ | $= \forall X\ \varphi_1$ | *und* $I[\![X/\mathbf{a}]\!] \models \varphi_1$ *für alle* $\mathbf{a} \in \mathcal{A}$ |
| *oder* | $\varphi$ | $= \exists X\ \varphi_1$ | *und* $I[\![X/\mathbf{a}]\!] \models \varphi_1$ *für ein* $\mathbf{a} \in \mathcal{A}$ |

*Eine Interpretation* $I$ *heißt* Modell *von* $\varphi$ *gdw.* $I \models \varphi$*.* $I$ *ist Modell einer Menge von Formeln* $\Phi$ *("*$I \models \Phi$*") gdw.* $I \models \varphi$ *für alle* $\varphi \in \Phi$ *gilt. Zwei Formeln* $\varphi_1$ *und* $\varphi_2$ *heißen* äquivalent *gdw. für alle Interpretationen* $I$ *gilt:* $I \models \varphi_1$ *gdw.* $I \models \varphi_2$*.*

*Eine Formel bzw. eine Formelmenge heißt* erfüllbar*, wenn sie ein Modell besitzt, und* unerfüllbar*, wenn sie kein Modell besitzt. Sie heißt* allgemeingültig*, wenn jede Interpretation ein Modell ist.*

*Eine Interpretation ohne Variablenbelegung* $S = (\mathcal{A}, \alpha)$ *wird als* Struktur *bezeichnet.*[2] *Eine Struktur besitzt also eine Trägermenge, wobei den Funktionssymbolen aus* $\Sigma$ *Funktionen auf dieser Trägermenge zugeordnet werden und den Prädikatssymbolen aus* $\Delta$ *Relationen auf der Trägermenge zugeordnet werden.*

*Sofern wir nur geschlossene Formeln betrachten, so kann man daher* Erfüllbarkeit *und* Modell *auch schon mit Hilfe von Strukturen definieren. Eine Struktur* $S$ *erfüllt dann eine geschlossene Formel* $\varphi$ *(d.h., "*$S \models \varphi$*" bzw.* $S$ *ist Modell von* $\varphi$*) gdw.* $I \models \varphi$ *für eine Interpretation der Form* $I = (\mathcal{A}, \alpha, \beta)$ *gilt. Die Variablenbelegung* $\beta$ *spielt hierbei keine Rolle, da* $\varphi$ *ja keine freien Variablen enthält. Ebenso kann man bei Grundtermen* $t$ *auch bereits* $S(t)$ *definieren (die Interpretation des Terms* $t$ *unter der Struktur* $S$*).*

**Beispiel 2.2.2** *Wir betrachten wieder die Signatur aus Bsp. 2.1.2. Eine Interpretation für diese Signatur ist beispielsweise* $I = (\mathcal{A}, \alpha, \beta)$ *mit*

$$
\begin{aligned}
\mathcal{A} &= \mathbb{N} \\
\alpha_n &= n \text{ für alle } n \in \mathbb{N} \\
\alpha_{\mathsf{monika}} &= 0 \\
\alpha_{\mathsf{karin}} &= 1 \\
\alpha_{\mathsf{renate}} &= 2 \\
&\vdots \\
\alpha_{\mathsf{datum}}(n_1, n_2, n_3) &= n_1 + n_2 + n_3 \text{ für alle } n_1, n_2, n_3 \in \mathbb{N} \\
\alpha_{\mathsf{weiblich}} &= \{n \mid n \text{ ist gerade}\} \\
\alpha_{\mathsf{maennlich}} &= \{n \mid n \text{ ist ungerade}\}
\end{aligned}
$$

[2] Sofern es keine Prädikatssymbole außer der Gleichheit gibt, spricht man auch von einer *Algebra*.

$$\begin{aligned}
\alpha_{\mathsf{mensch}} &= \mathbb{N} \\
\alpha_{\mathsf{verheiratet}} &= \{(n,m) \mid n > m\} \\
&\vdots \\
\beta(X) &= 0 \\
\beta(Y) &= 1 \\
\beta(Z) &= 2 \\
&\vdots
\end{aligned}$$

*Dann ist* $I(\mathsf{datum}(1, X, \mathsf{karin})) = \alpha_{\mathsf{datum}}(\alpha_1, \beta(X), \alpha_{\mathsf{karin}}) = 1 + 0 + 1 = 2$. *Somit gilt z.B.*

$$I \models \mathsf{verheiratet}(\mathsf{datum}(1, X, \mathsf{karin}), \mathsf{karin}).$$

*Ebenso gilt*

$$I \models \forall X\ \mathsf{weiblich}(\mathsf{datum}(X, X, \mathsf{monika})),$$

*denn für alle* $\mathbf{a} \in \mathcal{A}$ *ist* $I[\![X/\mathbf{a}]\!](\mathsf{datum}(X, X, \mathsf{monika})) = \mathbf{a} + \mathbf{a} + 0$ *eine gerade Zahl. Da die Formel* $\forall X\ \mathsf{weiblich}(\mathsf{datum}(X, X, \mathsf{monika}))$ *geschlossen ist, ist auch schon die Struktur* $S = (\mathcal{A}, \alpha)$ *ein Modell der Formel, d.h.*

$$S \models \forall X\ \mathsf{weiblich}(\mathsf{datum}(X, X, \mathsf{monika})).$$

*Die obigen Formeln sind somit alle erfüllbar, aber sie sind nicht allgemeingültig, da sie nicht von jeder Interpretation erfüllt werden. Beispiele für allgemeingültige Formeln sind* $\varphi \vee \neg\varphi$ *für alle Formeln* $\varphi$. *Beispiele für unerfüllbare Formeln sind* $\varphi \wedge \neg\varphi$ *für alle Formeln* $\varphi$.

Der Zusammenhang zwischen dem syntaktischen Begriff "Substitution" und dem semantischen Begriff "Variablenbelegung" wird durch folgendes Lemma beschrieben.

**Lemma 2.2.3 (Substitutionslemma)** *Sei* $I = (\mathcal{A}, \alpha, \beta)$ *eine Interpretation für eine Signatur* $(\Sigma, \Delta)$, *sei* $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ *eine Substitution. Dann gilt:*

*(a)* $I(\sigma(t)) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](t)$ *für alle* $t \in \mathcal{T}(\Sigma, \mathcal{V})$

*(b)* $I \models \sigma(\varphi)$ *gdw.* $I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!] \models \varphi$ *für alle* $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

**Beweis.**

(a) Der Beweis wird durch strukturelle Induktion über den Aufbau des Terms $t$ geführt. Im Induktionsanfang ist $t$ entweder eine Variable oder eine Konstante (d.h., ein Funktionssymbol aus $\Sigma_0$).

Falls $t$ eine Variable $X_i$ ist, so gilt

$$I(\sigma(X_i)) = I(t_i) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](X_i).$$

Ist $t$ eine Variable $Y$, die verschieden von allen $X_1, \ldots, X_n$ ist, so erhält man

$$I(\sigma(Y)) = I(Y) = \beta(Y) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](Y).$$

Hat $t$ die Gestalt $f(s_1, \ldots, s_k)$ (wobei $k = 0$ möglich ist), so gilt

$$I(\sigma(t)) = I(\sigma(f(s_1, \ldots, s_k))) = \alpha_f(I(\sigma(s_1)), \ldots, I(\sigma(s_k))).$$

Aus der Induktionshypothese folgt $I(\sigma(s_i)) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](s_i)$. Man erhält daher

$$\begin{array}{ll} & \alpha_f(I(\sigma(s_1)), \ldots, I(\sigma(s_k))) \\ = & \alpha_f(I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](s_1), \ldots, I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](s_k)) \\ = & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](f(s_1, \ldots, s_k)) \\ = & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](t). \end{array}$$

(b) Wir verwenden eine strukturelle Induktion über den Aufbau der Formel $\varphi$.

Falls $\varphi = p(s_1, \ldots, s_m)$ ist, so gilt

$$\begin{array}{lll} I \models \sigma(\varphi) & \text{gdw.} & I \models p(\sigma(s_1), \ldots, \sigma(s_m)) \\ & \text{gdw.} & (I(\sigma(s_1)), \ldots, I(\sigma(s_m))) \in \alpha_p \\ & \text{gdw.} & (I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!](s_1), \ldots, \\ & & \quad I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!](s_m)) \in \alpha_p \end{array}$$

nach Teil (a). Also erhält man

$$\begin{array}{ll} & (I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!](s_1), \ldots, I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!](s_m)) \in \alpha_p \\ \text{gdw.} & I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!] \models p(s_1, \ldots, s_m) \\ \text{gdw.} & I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!] \models \varphi. \end{array}$$

Die Fälle $\varphi = \neg\varphi_1$ oder $\varphi = \varphi_1 \cdot \varphi_2$ mit $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ sind sehr einfach. Betrachten wir noch den Fall $\varphi = \forall X\ \varphi_1$ (der Fall $\varphi = \exists X\ \varphi_1$ ist analog). O.B.d.A. betrachten wir nur den Fall $X \notin \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma)$. (Ansonsten wird die Formel zuerst in $\forall X'\ \varphi_1[X/X']$ überführt, die offensichtlich äquivalent zu $\varphi$ ist.) Es gilt $\sigma(\varphi) = \forall X\ \sigma(\varphi_1)$ und

$$I \models \sigma(\varphi) \quad \text{gdw.} \quad I[\![X/\mathbf{a}]\!] \models \sigma(\varphi_1) \text{ für alle } \mathbf{a} \in \mathcal{A}.$$

Sei $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ (wobei nach Voraussetzung $X$ keine der Variablen $X_i$ ist). Nach der Induktionshypothese ergibt sich

$$\begin{array}{ll} & I[\![X/\mathbf{a}]\!] \models \sigma(\varphi_1) \text{ für alle } \mathbf{a} \in \mathcal{A} \\ \text{gdw.} & I[\![X/\mathbf{a}]\!][\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!] \models \varphi_1 \text{ für alle } \mathbf{a} \in \mathcal{A}. \end{array}$$

Da $X$ verschieden von allen $X_1, \ldots, X_n$ ist, gilt $I[\![X/\mathbf{a}]\!][\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!] = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!][\![X/\mathbf{a}]\!]$ und wir erhalten somit

$$\begin{array}{ll} & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!][\![X/\mathbf{a}]\!] \models \varphi_1 \text{ für alle } \mathbf{a} \in \mathcal{A} \\ \text{gdw.} & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!] \models \forall X\ \varphi_1 \\ \text{gdw.} & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!] \models \varphi. \end{array}$$

□

**Beispiel 2.2.4** *Wir betrachten die Interpretation $I$ aus Bsp. 2.2.2, die Substitution $\sigma = \{X/\mathsf{datum}(1, X, \mathsf{karin})\}$ und den Term $t = \mathsf{datum}(X, Y, Z)$. Dann gilt*

$$\begin{aligned} I(\sigma(t)) &= I(\mathsf{datum}(\mathsf{datum}(1, X, \mathsf{karin}), Y, Z)) \\ &= \alpha_1 + \beta(X) + \alpha_{\mathsf{karin}} + \beta(Y) + \beta(Z) \\ &= 1 + 0 + 1 + 1 + 2 \\ &= 5. \end{aligned}$$

*Ebenso gilt*

$$\begin{aligned} I[\![X/I(\mathsf{datum}(1, X, \mathsf{karin}))]\!](t) &= I[\![X/2]\!](\mathsf{datum}(X, Y, Z)) \\ &= 2 + \beta(Y) + \beta(Z) \\ &= 2 + 1 + 2 \\ &= 5. \end{aligned}$$

Nun definieren wir, wann aus einer Formelmenge eine andere Formel folgt. Dies ist genau die Frage, die bei der Ausführung von Logikprogrammen untersucht wird.

**Definition 2.2.5 (Folgerbarkeit)** *Aus einer Formelmenge* $\Phi$ folgt *die Formel* $\varphi$ *(abgekürzt "*$\Phi \models \varphi$*") gdw. für alle Interpretationen $I$ mit $I \models \Phi$ gilt $I \models \varphi$. Sofern $\Phi$ und $\varphi$ keine freien Variablen enthalten, ist dies gleichbedeutend mit der Forderung, dass für alle Strukturen $S$ mit $S \models \Phi$ gilt $S \models \varphi$. (Das Zeichen "$\models$" steht also sowohl für Erfüllbarkeit durch eine Interpretation als auch für Folgerbarkeit aus einer Formelmenge. Was jeweils gemeint ist, erkennt man daran, ob links des Zeichens "$\models$" eine Interpretation oder eine Formelmenge steht.) Anstelle von "$\varnothing \models \varphi$" schreibt man meist "$\models \varphi$" (d.h., die Formel $\varphi$ ist allgemeingültig).*

**Beispiel 2.2.6** *Sei $\Phi$ die Formelmenge aus Bsp. 2.1.7, die dem Logikprogramm aus Kapitel 1 entspricht. Wenn man nun eine Anfrage wie*

```
?- maennlich(gerd).
```

*stellt, bedeutet das, dass man versucht $\Phi \models \mathsf{maennlich}(\mathsf{gerd})$ zu beweisen. Dies gilt natürlich im obigen Beispiel, da die Formel $\mathsf{maennlich}(\mathsf{gerd})$ in $\Phi$ enthalten ist. Ebenso gilt auch $\Phi \models \mathsf{mensch}(\mathsf{gerd})$, da $\Phi$ die Formel $\forall X\ \mathsf{mensch}(X)$ enthält.*

*Die Anfrage*

```
?- mutterVon(X,susanne).
```

*bedeutet, dass man untersuchen will, ob $\Phi \models \exists X\ \mathsf{mutterVon}(X, \mathsf{susanne})$ gilt. Wie erwähnt sind Variablen $X$ im Logikprogramm allquantifiziert und in Anfragen existenzquantifiziert. Da $\Phi$ die Formel $\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne})$ enthält, gilt $\Phi \models \exists X\ \mathsf{mutterVon}(X, \mathsf{susanne})$. Hierbei muss die Variable $X$ jeweils so belegt werden, wie die Deutung von* renate.

Zur Ausführung von Logikprogrammen muss man also für eine Formelmenge $\Phi$ (das Logikprogramm) und eine Formel $\varphi$ (die Anfrage) herausfinden, ob $\Phi \models \varphi$ gilt. Im folgenden Kapitel werden wir zeigen, wie diese Frage automatisch untersucht werden kann.

# Kapitel 3

# Resolution

Der Begriff der "Folgerbarkeit" ist auf semantische Weise definiert. Diese Definition eignet sich aber nicht zu einer rechnergestützten Überprüfung, denn um $\Phi \models \varphi$ zu untersuchen, müssten wir bisher alle (unendlich vielen) Interpretationen betrachten. Zu jeder Interpretation müsste man herausfinden, ob sie Modell von $\Phi$ ist und in diesem Fall überprüfen, ob sie auch Modell von $\varphi$ ist.

Um Folgerbarkeit stattdessen auf syntaktische (und automatisierbare) Weise zu untersuchen, verwendet man einen sogenannten *Kalkül*. Ein Kalkül besteht aus bestimmten (meist rein syntaktischen) Regeln, die es erlauben, aus einer Menge von Formeln $\Phi$ neue Formeln herzuleiten. Die Überprüfung, ob $\varphi$ aus $\Phi$ *herleitbar* ist, kann dann auf syntaktische Weise erfolgen.

Damit ein solcher Kalkül etwas über Folgerbarkeit aussagt, sollte er natürlich *korrekt* sein, d.h., falls man $\varphi$ aus $\Phi$ herleiten kann, dann sollte die Folgerbarkeit $\Phi \models \varphi$ gelten. Gilt auch der Umkehrschritt, d.h., folgt aus der Folgerbarkeit $\Phi \models \varphi$ auch die Herleitbarkeit von $\varphi$ aus $\Phi$, so heißt der Kalkül *vollständig*.

In diesem Kapitel stellen wir den sogenannten *Resolutionskalkül* vor, der in der Logikprogrammierung verwendet wird, um $\Phi \models \varphi$ zu untersuchen. Wir zeigen, dass dieser Kalkül sowohl korrekt als auch vollständig ist, d.h., hier entsprechen sich Folgerbarkeit und Herleitbarkeit.

Die Idee des Resolutionskalküls ist, das Folgerbarkeitsproblem $\Phi \models \varphi$ auf ein *Unerfüllbarkeitsproblem* zu reduzieren und anschließend dieses Unerfüllbarkeitsproblem durch Resolution zu untersuchen. Das folgende Lemma zeigt, wie jedes Folgerbarkeitsproblem in ein Unerfüllbarkeitsproblem überführt werden kann.

**Lemma 3.0.1 (Überführung von Folgerbarkeits- in Unerfüllbarkeitsprobleme)** *Seien $\varphi_1, \ldots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$. Dann gilt $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ gdw. die Formel $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$ unerfüllbar ist.*

**Beweis.**

| | |
|---|---|
| Es gilt | $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ |
| gdw. | für alle Interpretationen $I$ mit $I \models \{\varphi_1, \ldots, \varphi_k\}$ gilt $I \models \varphi$ |
| gdw. | es gibt keine Interpretation $I$ mit $I \models \{\varphi_1, \ldots, \varphi_k\}$ und $I \models \neg\varphi$ |
| gdw. | $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$ ist unerfüllbar |

□

**Beispiel 3.0.2** *Um zu zeigen, dass in dem Logikprogramm mit dem Faktum*

```
mutterVon(renate,susanne).
```

*die Anfrage*

```
?- mutterVon(X,susanne).
```

*beweisbar ist, muss man zeigen, dass*

$$\{\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne})\} \models \exists\, X\ \mathsf{mutterVon}(X, \mathsf{susanne})$$

*gilt. Stattdessen kann man also jetzt die Unerfüllbarkeit der folgenden Formel nachweisen:*

$$\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg\exists\, X\ \mathsf{mutterVon}(X, \mathsf{susanne})$$

Das Problem, die Unerfüllbarkeit einer Menge von Formeln nachzuweisen (und damit auch das Problem, die Folgerbarkeit zu untersuchen) ist im Allgemeinen *unentscheidbar*, vgl. z.B. [Gra11, Satz 4.30]. Es existiert also kein automatisches Verfahren, dass stets terminiert und zu jeder Formel die Unerfüllbarkeit überprüfen kann. Allerdings ist die Unerfüllbarkeit (und damit auch die Folgerbarkeit) *semi–entscheidbar*. Es existiert also ein Algorithmus, der für jede unerfüllbare Formel die Unerfüllbarkeit in endlicher Zeit herausfindet, bei erfüllbaren Formeln jedoch eventuell nicht terminiert. Für das Folgerbarkeitsproblem bedeutet das, dass man $\Phi \models \varphi$ mit dem Algorithmus stets in endlicher Zeit beweisen kann, falls es gilt. Falls aber $\Phi \not\models \varphi$ gilt, so terminiert der Algorithmus evtl. nicht. Da der Resolutionskalkül ein korrektes und vollständiges Verfahren zur Überprüfung der Unerfüllbarkeit einer Formel ist, handelt es sich damit um ein solches Semi-Entscheidungsverfahren. Falls die Formel unerfüllbar ist, so kann man dies durch eine Automatisierung des Resolutionskalküls auch nachweisen. Falls sie aber erfüllbar ist, so terminiert die Automatisierung des Resolutionskalküls eventuell nicht.

Wir führen das Resolutionsprinzip zur Überprüfung der Unerfüllbarkeit einer Formel $\varphi$ in vier Schritten ein. Die Grundidee hierbei ist, das Resolutionsprinzip für die Prädikatenlogik auf das Resolutionsprinzip der Aussagenlogik zu reduzieren. In der Aussagenlogik ist die Korrektheit und Vollständigkeit der Resolution leichter nachweisbar. (In der Aussagenlogik ist die Resolution außerdem ein *Entscheidungsverfahren* für die Unerfüllbarkeit.)

1. Zunächst zeigen wir, dass man $\varphi$ zur Überprüfung der Unerfüllbarkeit in *Skolem–Normalform* überführen kann (Abschnitt 3.1). Formeln in Skolem–Normalform sind geschlossen, sie besitzen keine Existenzquantoren mehr und sie enthalten Allquantoren nur ganz außen. Solche Formeln haben also die Gestalt $\forall X_1, \ldots, X_n\ \psi$, wobei $\psi$ quantorfrei ist und keine Variablen außer $X_1, \ldots, X_n$ enthält.

2. Dann zeigen wir, dass man bei Formeln in Skolem–Normalform zur Untersuchung der Unerfüllbarkeit nicht *alle* Interpretationen betrachten muss, sondern sich auf sogenannte *Herbrand-Interpretationen* beschränken kann (Abschnitt 3.2). Dies sind Interpretationen, bei denen Grundterme als "sich selbst" gedeutet werden. Hiermit ergibt sich bereits ein erstes automatisierbares Verfahren zur Untersuchung der Unerfüllbarkeit, das allerdings noch recht ineffizient ist.

3. Um ein effizienteres Verfahren zu erhalten, erweitern wir schließlich die aussagenlogische Resolution (Abschnitt 3.3) auf die Prädikatenlogik, indem wir hierzu das Verfahren der *Unifikation* benutzen (Abschnitt 3.4).

4. Um die Effizienz weiter zu erhöhen, zeigen wir in Abschnitt 3.5, wie sich die Resolution weiter einschränken lässt.

## 3.1 Skolem–Normalform

Wie erwähnt, besteht das Ziel zunächst darin, Formeln in eine Normalform der Gestalt $\forall X_1, \ldots, X_n\ \psi$ zu überführen, wobei $\psi$ quantorfrei ist und keine Variablen außer $X_1, \ldots, X_n$ enthält. Hierzu geht man in zwei Schritten vor. Zunächst überführt man die Formel in *Pränex–Normalform.*

**Definition 3.1.1 (Pränex–Normalform)** *Eine Formel $\varphi$ ist in* Pränex–Normalform *gdw. sie die Gestalt $Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$ mit $Q_i \in \{\forall, \exists\}$ hat, wobei $\psi$ quantorfrei ist.*

Der folgende Satz zeigt, dass (und wie) man jede Formel in eine äquivalente Formel in Pränex–Normalform überführen kann.

**Satz 3.1.2 (Überführung in Pränex–Normalform)** *Zu jeder Formel $\varphi$ lässt sich automatisch eine Formel $\varphi'$ in Pränex–Normalform konstruieren, so dass $\varphi$ und $\varphi'$ äquivalent sind.*

**Beweis.** Zunächst werden alle Teilformeln $\varphi_1 \leftrightarrow \varphi_2$ in $\varphi$ durch $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$ ersetzt. Anschließend werden alle Teilformeln $\varphi_1 \rightarrow \varphi_2$ in $\varphi$ durch $\neg\varphi_1 \vee \varphi_2$ ersetzt. Die Überführung der verbleibenden Formel geschieht mit folgendem Algorithmus $PRAENEX$, dessen Terminierung und Korrektheit offensichtlich ist. Der Algorithmus bekommt als Eingabe eine beliebige Formel $\varphi$ ohne die Junktoren "$\leftrightarrow$" und "$\rightarrow$" und liefert als Ausgabe eine dazu äquivalente Formel in Pränex–Normalform.

- Falls $\varphi$ quantorfrei ist, so gib $\varphi$ zurück.
- Falls $\varphi = \neg\varphi_1$ ist, so berechne $PRAENEX(\varphi_1) = Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$. Liefere $\overline{Q_1}\, X_1\ \ldots \overline{Q_n}\, X_n\ \neg\psi$ zurück, wobei $\overline{\forall} = \exists$ und $\overline{\exists} = \forall$.
- Falls $\varphi = \varphi_1 \cdot \varphi_2$ mit $\cdot \in \{\wedge, \vee\}$, so berechne $PRAENEX(\varphi_1) = Q_1\, X_1\ \ldots Q_n\, X_n\ \psi_1$ und $PRAENEX(\varphi_2) = R_1\, Y_1\ \ldots R_m\, Y_m\ \psi_2$. Durch Umbenennung gebundener Variablen erreichen wir, dass $X_1, \ldots, X_n$ nicht in $R_1\, Y_1\ \ldots R_m\, Y_m\ \psi_2$ auftreten und dass $Y_1, \ldots, Y_m$ nicht in $Q_1\, X_1\ \ldots Q_n\, X_n\ \psi_1$ auftreten. Gib nun die folgende Formel zurück:

$$Q_1\, X_1\ \ldots Q_n\, X_n\ R_1\, Y_1\ \ldots R_m\, Y_m\ \ (\psi_1 \cdot \psi_2)$$

- Falls $\varphi = Q\, X\ \varphi_1$ mit $Q \in \{\forall, \exists\}$ ist, so berechne die Formel $PRAENEX(\varphi_1) = Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$. Durch Umbenennung gebundener Variablen erreichen wir, dass $X_1, \ldots, X_n$ verschieden von $X$ sind. Dann gib $Q\, X\, Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$ zurück.

□

**Beispiel 3.1.3** *Wir betrachten die Überführung der folgenden Formel:*

$$\neg\exists X\,(\,\mathsf{verheiratet}(X,Y) \vee \neg\exists Y\,\mathsf{mutterVon}(X,Y)\,)$$

*Zunächst berechnen wir* $PRAENEX(\neg\exists Y\,\mathsf{mutterVon}(X,Y)) = \forall Y\,\neg\mathsf{mutterVon}(X,Y)$. *Damit die gebundene Variable* $Y$ *in dieser Teilformel von der freien Variablen* $Y$ *in der Teilformel* $\mathsf{verheiratet}(X,Y)$ *verschieden ist, benennen wir erstere in* $Z$ *um und erhalten somit* $\forall Z\,\neg\mathsf{mutterVon}(X,Z)$.

*Nun berechnen wir* $PRAENEX(\mathsf{verheiratet}(X,Y) \vee \neg\exists Y\,\mathsf{mutterVon}(X,Y))$, *was die folgende Formel ergibt:*

$$\forall Z\,(\mathsf{verheiratet}(X,Y) \vee \neg\mathsf{mutterVon}(X,Z))$$

*Schließlich ergibt sich* $PRAENEX(\neg\exists X\,(\mathsf{verheiratet}(X,Y) \vee \neg\exists Y\,\mathsf{mutterVon}(X,Y))) = \forall X\,\exists Z\,\neg(\mathsf{verheiratet}(X,Y) \vee \neg\mathsf{mutterVon}(X,Z))$.

**Beispiel 3.1.4** *Als ein weiteres Beispiel betrachten wir die Überführung der Formel* $\mathsf{mutterVon}(\mathsf{renate},\mathsf{susanne}) \wedge \neg\exists\, X\ \mathsf{mutterVon}(X,\mathsf{susanne})$ *aus Bsp. 3.0.2, deren Unerfüllbarkeit nachgewiesen werden muss, um zu zeigen, dass die Anfrage*

```
?- mutterVon(X,susanne).
```

*im Logikprogramm mit dem Faktum*

```
mutterVon(renate,susanne).
```

*nachweisbar ist. Hier ergibt sich bei der Überführung in Pränex–Normalform die folgende Formel:*

$$\forall\, X\ \mathsf{mutterVon}(\mathsf{renate},\mathsf{susanne}) \wedge \neg\mathsf{mutterVon}(X,\mathsf{susanne})$$

Nun definieren wir die Skolem–Normalform.

**Definition 3.1.5 (Skolem–Normalform)** *Eine Formel* $\varphi$ *ist in* Skolem–Normalform *gdw. sie geschlossen ist und sie die Gestalt* $\forall\, X_1,\ldots,X_n\ \psi$ *hat, wobei* $\psi$ *quantorfrei ist.*

Zur Überführung einer Formel in Skolem–Normalform wird die Formel zunächst in Pränex–Normalform überführt. Die verbleibende Überführung dient dazu, die freien Variablen und Existenzquantoren zu beseitigen. Anders als bei der Pränex–Normalform existiert nicht zu jeder Formel eine äquivalente Formel in Skolem–Normalform, denn offensichtlich gibt es z.B. keine Formel in Skolem–Normalform, die zu $\mathsf{weiblich}(X)$ oder zu $\exists X\ \mathsf{weiblich}(X)$ äquivalent ist. Es gibt aber zu jeder Formel eine dazu *erfüllbarkeitsäquivalente* Formel in Skolem–Normalform.

Die Idee der Überführung besteht darin, zuerst die Pränex–Normalform zu bilden, dann alle freien Variablen durch Existenzquantoren zu binden, und schließlich alle existenzquantifizierten Variablen mit Hilfe neuer Funktionssymbole zu beseitigen.

**Satz 3.1.6 (Überführung in Skolem–Normalform)** *Zu jeder Formel* $\varphi$ *lässt sich automatisch eine Formel* $\varphi'$ *in Skolem–Normalform konstruieren, so dass* $\varphi$ *genau dann erfüllbar ist, wenn* $\varphi'$ *erfüllbar ist.*

**Beweis.** Zunächst wird die Formel $\varphi$ mit dem Verfahren aus Satz 3.1.2 in Pränex–Normalform überführt. Seien $X_1, \ldots, X_n$ die freien Variablen in der entstehenden Formel $\varphi_1$, die zu $\varphi$ äquivalent ist. Dann wird $\varphi_1$ weiter überführt in die geschlossene Formel $\varphi_2$ der Gestalt $\exists X_1, \ldots, X_n\ \varphi_1$, die zu $\varphi_1$ erfüllbarkeitsäquivalent ist: Aus $I \models \varphi_1$ folgt $I[\![X_1/\beta(X_1), \ldots, X_n/\beta(X_n)]\!] \models \varphi_1$ und damit offensichtlich auch $I \models \exists X_1, \ldots, X_n\ \varphi_1$. Umgekehrt folgt aus $I \models \exists X_1, \ldots, X_n\ \varphi_1$, dass es $\mathbf{a_1}, \ldots, \mathbf{a_n}$ gibt, so dass $I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \varphi_1$ gilt.

Die Formel $\varphi_2$ ist damit geschlossen, in Pränex–Normalform und zu $\varphi$ erfüllbarkeitsäquivalent. Nun eliminieren wir die Existenzquantoren schrittweise von außen nach innen. Falls $\varphi_2$ die Formel $\forall X_1, \ldots, X_n \exists Y\ \psi$ ist, so ersetzen wir $\varphi_2$ durch die folgende Formel:

$$\forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$$

Hierbei werden also alle Vorkommen von $Y$ durch $f(X_1, \ldots, X_n)$ ersetzt, wobei $f$ ein neues $n$-stelliges Funktionssymbol ist. Dieses Vorgehen wird solange wiederholt, bis keine Existenzquantoren mehr vorhanden sind. Die entstehende Formel ist erfüllbarkeitsäquivalent zu $\varphi_2$ und damit auch zu $\varphi$. Es gilt nämlich:

$$
\begin{array}{ll}
 & I \models \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)] \\
\curvearrowright & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \psi[Y/f(X_1, \ldots, X_n)] \text{ für alle } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A} \\
\curvearrowright & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!][\![Y/I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!](f(X_1, \ldots, X_n))]\!] \models \psi \\
 & \qquad \text{für alle } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}, \text{ wegen Substitutionslemma 2.2.3} \\
\curvearrowright & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \exists Y\ \psi \text{ für alle } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A} \\
\curvearrowright & I \models \forall X_1, \ldots, X_n\ \exists Y\ \psi \text{ für alle } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}
\end{array}
$$

Aus $I \models \forall X_1, \ldots, X_n \exists Y\ \psi$ mit $I = (\mathcal{A}, \alpha, \beta)$ folgt, dass es für alle $\mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}$ ein $\mathbf{b} \in \mathcal{A}$ gibt, so dass $I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}, Y/\mathbf{b}]\!] \models \psi$. Sei $F$ die Funktion von $\mathcal{A}^n \to \mathcal{A}$, die jedem Tupel $(\mathbf{a_1}, \ldots, \mathbf{a_n})$ jeweils das entsprechende $\mathbf{b}$ zuordnet. Sei $I' = (\mathcal{A}, \alpha', \beta)$, wobei sich $\alpha'$ von $\alpha$ nur in der Deutung des neuen Funktionssymbols $f$ unterscheidet. Hier definieren wir $\alpha'_f = F$. Dann erhalten wir $I' \models \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$. Der Grund ist, dass für alle $\mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}$ gilt:

$$
\begin{array}{ll}
 & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}, Y/F(\mathbf{a_1}, \ldots, \mathbf{a_n})]\!] \models \psi \\
\text{gdw.} & I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}, Y/F(\mathbf{a_1}, \ldots, \mathbf{a_n})]\!] \models \psi \text{ da } f \text{ nicht in } \psi \text{ auftritt} \\
\text{gdw.} & I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!][\![Y/I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!](f(X_1, \ldots, X_n))]\!] \models \psi \\
\text{gdw.} & I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \psi[Y/f(X_1, \ldots, X_n)]
\end{array}
$$

nach dem Substitutionslemma 2.2.3. Da diese Aussage für alle $\mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}$ gilt, erhält man also $I' \models \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$. □

**Beispiel 3.1.7** *Wir betrachten die Überführung der Formel*

$$\neg\exists X\ (\ \mathsf{verheiratet}(X, Y) \vee \neg\exists Y\ \mathsf{mutterVon}(X, Y)\ )$$

*aus Bsp. 3.1.3 in Skolem–Normalform. Wie in Bsp. 3.1.3 wird die Formel zuerst in Pränex–Normalform transformiert, was zu*

$$\forall X\ \exists Z\ \neg(\mathsf{verheiratet}(X, Y) \vee \neg\mathsf{mutterVon}(X, Z))$$

*führt. Anschließend wird die freie Variable $Y$ existenzquantifiziert, was*

$$\exists Y \, \forall X \, \exists Z \, \neg(\mathsf{verheiratet}(X, Y) \vee \neg\mathsf{mutterVon}(X, Z))$$

*ergibt. Um die Existenzquantoren zu beseitigen, wird nun zuerst die äußerste existenzquantifizierte Variable $Y$ durch eine neue Konstante $\mathsf{a}$ ersetzt. Hierbei ist $\mathsf{a}$ nullstellig, weil es vor dem Existenzquantor von $Y$ keine Allquantoren mehr gibt. Dies ergibt*

$$\forall X \, \exists Z \, \neg(\mathsf{verheiratet}(X, \mathsf{a}) \vee \neg\mathsf{mutterVon}(X, Z)).$$

*Schließlich wird $Z$ durch $\mathsf{f}(X)$ ersetzt, wobei $\mathsf{f}$ ein neues einstelliges Funktionssymbol ist. Dies führt zu*

$$\forall X \, \neg(\mathsf{verheiratet}(X, \mathsf{a}) \vee \neg\mathsf{mutterVon}(X, \mathsf{f}(X))).$$

## 3.2 Herbrand–Strukturen

Das Problem bei der Untersuchung der Unerfüllbarkeit einer Formel ist, dass dazu alle Interpretationen bzw. Strukturen betrachtet werden müssen und der Träger kann hierbei eine vollkommen *beliebige* Menge sein. In diesem Abschnitt zeigen wir, dass man sich bei Formeln in Skolem–Normalform auf solche Strukturen beschränken kann, bei denen der Träger gerade aus den Grundtermen besteht und bei denen Funktionssymbole als "sich selbst" gedeutet werden. Solche Strukturen bezeichnet man als *Herbrand–Strukturen* (nach dem Logiker *Jacques Herbrand*).

**Definition 3.2.1 (Herbrand–Struktur)** *Sei $(\Sigma, \Delta)$ eine Signatur. Eine* Herbrand–Struktur *(oder "freie" Struktur) hat dann die Gestalt $(\mathcal{T}(\Sigma), \alpha)$, wobei für alle $f \in \Sigma_n$ mit $n \in \mathbb{N}$ gilt: $\alpha_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$. Falls eine Herbrand–Struktur Modell einer Formel $\varphi$ ist, so bezeichnen wir sie auch als* Herbrand–Modell *von $\varphi$.*

Man erkennt, dass bei einer Herbrand–Struktur $S$ alle Grundterme $t$ als "sich selbst" interpretiert werden, d.h., es gilt $S(t) = t$, wie man durch einen leichten Induktionsbeweis nachweist. Bei einer Herbrand–Struktur ist also der Träger und die Deutung der Funktionssymbole festgelegt. Lediglich die Deutung der Prädikatssymbole kann noch frei gewählt werden. Wenn man zur Untersuchung der Unerfüllbarkeit also nur noch Herbrand–Strukturen statt beliebiger Strukturen betrachten muss, ist der Suchraum bereits deutlich eingeschränkt.

**Beispiel 3.2.2** *Wir betrachten wieder die Signatur $(\Sigma, \Delta)$ aus Bsp. 2.1.2. Eine Herbrand–Struktur für diese Signatur ist beispielsweise $S = (\mathcal{T}(\Sigma), \alpha)$ mit*

$$\begin{aligned}
\alpha_n &= n \text{ für alle } n \in \mathbb{N} \\
\alpha_{\mathsf{monika}} &= \mathsf{monika} \\
\alpha_{\mathsf{karin}} &= \mathsf{karin} \\
&\vdots \\
\alpha_{\mathsf{datum}}(t_1, t_2, t_3) &= \mathsf{datum}(t_1, t_2, t_3) \text{ für alle } t_1, t_2, t_3 \in \mathcal{T}(\Sigma)
\end{aligned}$$

$$\begin{aligned}
\alpha_{\mathsf{weiblich}} &= \{\mathsf{monika}, \mathsf{karin}, \ldots\} \\
\alpha_{\mathsf{maennlich}} &= \{\mathsf{werner}, \mathsf{klaus}, \ldots\} \\
\alpha_{\mathsf{mensch}} &= \mathcal{T}(\Sigma) \\
\alpha_{\mathsf{geboren}} &= \{(\mathsf{monika}, \mathsf{datum}(25, 7, 1972)), (\mathsf{werner}, \mathsf{datum}(12, 7, 1969)), \ldots\} \\
&\vdots
\end{aligned}$$

*Diese Herbrand–Struktur kommt damit der intuitiv erwarteten Semantik des ursprünglichen Logikprogramms bereits wesentlich näher.*

Der folgende Satz zeigt, dass man sich zur Untersuchung der Erfüllbarkeit in der Tat auf Herbrand–Strukturen beschränken kann. Hierzu muss man die ursprüngliche Formel zunächst wie in Satz 3.1.6 in eine erfüllbarkeitsäquivalente Formel in Skolem–Normalform überführen.

**Satz 3.2.3 (Erfüllbarkeitstest durch Herbrand–Strukturen)** *Sei $\Phi \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ eine Menge von Formeln in Skolem–Normalform. Dann ist $\Phi$ erfüllbar gdw. sie ein Herbrand–Modell besitzt.*

**Beweis.** Die Richtung "$\Leftarrow$" ist offensichtlich, da jedes Herbrand–Modell auch ein Modell ist. Wir zeigen nun die Richtung "$\Rightarrow$". Sei hierzu $S = (\mathcal{A}, \alpha)$ ein Modell von $\Phi$. (Es genügt, hierbei Strukturen statt Interpretationen zu betrachten, da Formeln in Skolem–Normalform geschlossen sind.) Wir zeigen, dass dann die Herbrand–Struktur $S' = (\mathcal{T}(\Sigma), \alpha')$ ebenfalls Modell von $\Phi$ ist. Wie bei jeder Herbrand–Struktur gilt hierbei $\alpha'_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ für alle $f \in \Sigma_n$ und alle $n \in \mathbb{N}$. Die Deutung der Prädikatssymbole wird wie folgt definiert. Für $p \in \Delta_0$ sei $\alpha'_p = \alpha_p$ und für alle $p \in \Delta_n$ mit $n \geq 1$ sei

$$(t_1, \ldots, t_n) \in \alpha'_p \qquad \text{gdw.} \qquad (S(t_1), \ldots, S(t_n)) \in \alpha_p.$$

Wir zeigen, dass für jede Formel $\varphi$ in Skolem–Normalform aus $S \models \varphi$ jeweils $S' \models \varphi$ folgt. Da $\varphi$ in Skolem–Normalform ist, hat es die Gestalt $\forall X_1, \ldots, X_n\ \psi$, wobei $\psi$ quantorfrei ist. Um die obige Behauptung zu beweisen, verwenden wir Induktion über die Anzahl $n$ der allquantifizierten Variablen von $\varphi$.

Im Induktionsanfang ist $n = 0$, d.h., $\varphi = \psi$ ist eine quantorfreie Formel ohne Variablen. Hier gilt sogar "$S \models \varphi$ gdw. $S' \models \varphi$", wie sich durch Induktion über den Formelaufbau zeigen lässt. Bei atomaren Formeln $p(t_1, \ldots, t_n)$ (d.h. im Induktionsanfang) gilt nach Definition:

$$\begin{aligned}
S \models p(t_1, \ldots, t_n) \quad &\text{gdw.} \quad (S(t_1), \ldots, S(t_n)) \in \alpha_p \\
&\text{gdw.} \quad (t_1, \ldots, t_n) \in \alpha'_p && \text{(nach Definition)} \\
&\text{gdw.} \quad (S'(t_1), \ldots, S'(t_n)) \in \alpha'_p && \text{(da } S'(t_i) = t_i\text{)} \\
&\text{gdw.} \quad S' \models p(t_1, \ldots, t_n)
\end{aligned}$$

Der Beweis bei nicht-atomaren quantorfreien Formeln (d.h. der Induktionsschluss) ist offensichtlich.

Nun betrachten wir den Fall $n > 0$, d.h. den Induktionsschluss der äußeren Induktion. Man beachte, dass die Formel $\forall X_1, \ldots, X_{n-1}\ \psi$, die durch Weglassen des Quantors "$\forall X_n$"

entsteht, im Allgemeinen keine geschlossene Formel ist, da sie die Variable $X_n$ frei enthalten könnte. Im Folgenden bezeichne $S[\![X_n/\mathbf{a}]\!]$ eine Interpretation, die aus der Struktur $S$ durch Hinzunahme der Variablenbelegung $\beta[\![X_n/\mathbf{a}]\!]$ für eine beliebige Variablenbelegung $\beta$ entsteht. Falls $I = (\mathcal{A}, \alpha, \beta)$ ist, so ist also $S[\![X_n/\mathbf{a}]\!]$ eine Kurzschreibweise für $I[\![X_n/\mathbf{a}]\!]$. Es gilt:

$$
\begin{array}{lll}
 & S \models \forall X_1, \ldots, X_n\ \psi & \\
\text{gdw.} & S[\![X_n/\mathbf{a}]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ für alle } \mathbf{a} \in \mathcal{A} & \\
\curvearrowright & S[\![X_n/S(t)]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ für alle } t \in \mathcal{T}(\Sigma) & \\
\text{gdw.} & S \models \forall X_1, \ldots, X_{n-1}\ \psi[X_n/t] \text{ für alle } t \in \mathcal{T}(\Sigma) & \text{(Substitutionslemma 2.2.3)} \\
\curvearrowright & S' \models \forall X_1, \ldots, X_{n-1}\ \psi[X_n/t] \text{ für alle } t \in \mathcal{T}(\Sigma) & \text{(Induktionshypothese)} \\
\text{gdw.} & S'[\![X_n/S'(t)]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ für alle } t \in \mathcal{T}(\Sigma) & \text{(Substitutionslemma 2.2.3)} \\
\text{gdw.} & S'[\![X_n/t]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ für alle } t \in \mathcal{T}(\Sigma) & \text{(da } S'(t) = t \text{ für } t \in \mathcal{T}(\Sigma)) \\
\text{gdw.} & S' \models \forall X_1, \ldots, X_n\ \psi & 
\end{array}
$$

□

**Beispiel 3.2.4** *Das folgende Beispiel zeigt, dass der obige Satz tatsächlich nur für Formeln in Skolem–Normalform gilt. So ist die Formelmenge* $\{\mathsf{p}(\mathsf{a}), \exists X\ \neg\mathsf{p}(X)\}$ *über der Signatur mit* $\Sigma = \Sigma_0 = \{\mathsf{a}\}$ *und* $\Delta = \Delta_1 = \{\mathsf{p}\}$ *erfüllbar. Ein Modell ist z.B. die Struktur* $S = (\{0,1\}, \alpha)$ *mit* $\alpha_{\mathsf{a}} = 0$ *und* $\alpha_{\mathsf{p}} = \{0\}$*. Es ist hierbei aber entscheidend, dass es Objekte im Träger (wie hier die* 1*) gibt, die nicht als Interpretation eines Grundterms erreicht werden (d.h., es existiert kein Grundterm* $t$ *mit* $S(t) = 1$*). In der Tat hat diese Formelmenge kein Herbrand–Modell. Der Grund ist, dass der Träger jeder Herbrand–Struktur aus der Menge* $\{\mathsf{a}\}$ *besteht.*

Nachdem wir nun wissen, dass man zur Überprüfung der Unerfüllbarkeit einer Formel $\varphi$ in Skolem–Normalform nur Herbrand–Strukturen betrachten muss, reduzieren wir dies nun auf die Überprüfung der Unerfüllbarkeit einer (unendlichen) Menge von Formeln ohne Variablen. Wie wir sehen werden, entspricht das der Überprüfung der Unerfüllbarkeit für eine (unendliche) Menge *aussagenlogischer* Formeln. Die Idee hierzu besteht darin, alle in $\varphi$ allquantifizierten Variablen mit allen möglichen Grundtermen zu instantiieren. Auf diese Weise entsteht die *Herbrand–Expansion* von $\varphi$.

**Definition 3.2.5 (Herbrand–Expansion einer Formel)** *Sei* $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ *eine Formel in Skolem–Normalform, wobei* $\varphi = \forall X_1, \ldots, X_n\ \psi$ *für eine quantorfreie Formel* $\psi$*. Die folgende Formelmenge* $E(\varphi)$ *wird als* Herbrand–Expansion *von* $\varphi$ *bezeichnet:*

$$E(\varphi) = \{\psi[X_1/t_1, \ldots, X_n/t_n] \mid t_1, \ldots, t_n \in \mathcal{T}(\Sigma)\}$$

Die Herbrand–Expansion einer Formel $\forall X_1, \ldots, X_n\ \psi$ ist also die Menge aller *Grundinstanzen* von $\psi$.

**Beispiel 3.2.6** *Wir betrachten die Formel* $\varphi$

$$\forall X\ \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg\mathsf{mutterVon}(X, \mathsf{susanne})$$

*aus Bsp. 3.1.4, deren Unerfüllbarkeit gezeigt werden muss, um nachzuweisen, dass die Anfrage*

```
?- mutterVon(X,susanne).
```

*aus dem Logikprogramm mit dem Faktum*

```
mutterVon(renate,susanne).
```

*folgt. Wir erhalten hier*

$$E(\varphi) = \{ \begin{array}{l} \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{mutterVon}(\mathsf{karin}, \mathsf{susanne}), \\ \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}), \\ \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{mutterVon}(\mathsf{datum}(\mathsf{karin}, \mathsf{werner}, 1982), \mathsf{susanne}), \\ \ldots \quad \}. \end{array}$$

Im obigen Beispiel sieht man, dass $E(\varphi)$ eine offensichtlich unerfüllbare Formel enthält, nämlich $\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne})$. Der folgende Satz sagt, dass dies bei jeder unerfüllbaren Formel $\varphi$ in Skolem–Normalform der Fall ist. Man kann also den Unerfüllbarkeitstest auf die Untersuchung der Unerfüllbarkeit einer Menge von variablenfreien Formeln reduzieren. Dieser Satz beruht auf dem vorigen Satz 3.2.3, der die Einschränkung auf Herbrand–Modelle rechtfertigt.

**Satz 3.2.7 (Erfüllbarkeit der Herbrand–Expansion)** *Sei $\varphi$ eine Formel in Skolem–Normalform. Dann ist $\varphi$ erfüllbar gdw. $E(\varphi)$ erfüllbar ist.*

**Beweis.** Da $\varphi$ in Skolem–Normalform ist, hat es die Gestalt $\forall X_1, \ldots, X_n\ \psi$, wobei $\psi$ quantorfrei ist. Nun gilt:

| | | |
|---|---|---|
| | $\forall X_1, \ldots, X_n\ \psi$ ist erfüllbar | |
| gdw. | es existiert eine Herbrand–Struktur $S$ mit $S \models \forall X_1, \ldots, X_n\ \psi$ | (Satz 3.2.3) |
| gdw. | es existiert eine Herbrand–Struktur $S$ mit $S[\![X_1/t_1, \ldots, X_n/t_n]\!] \models \psi$ für alle $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ | |
| gdw. | es existiert eine Herbrand–Struktur $S$ mit $S \models \psi[X_1/t_1, \ldots, X_n/t_n]$ für alle $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ | (Substitutionslemma 2.2.3) |
| gdw. | es existiert eine Herbrand–Struktur $S$ mit $S \models E(\varphi)$ | |
| gdw. | $E(\varphi)$ ist erfüllbar | (Satz 3.2.3) |

□

Prädikatenlogische Formeln ohne Variablen entsprechen aussagenlogischen Formeln. Der Grund ist, dass man jede atomare Teilformel $p(t_1, \ldots, t_n)$ als aussagenlogische Variable ansehen kann, die entweder wahr oder falsch sein kann. Nach Satz 3.2.3 und 3.2.7 genügt es zu untersuchen, ob die Menge $E(\varphi)$ ein Herbrand–Modell hat. Herbrand–Strukturen unterscheiden sich aber ja nur in der Deutung der Prädikatssymbole und bei variablenfreien Formeln bedeutet die Suche nach einem Herbrand–Modell nichts anderes als die Suche nach einer aussagenlogischen Belegung der aussagenlogischen Variablen $p(t_1, \ldots, t_n)$ mit "wahr" oder "falsch". Die Frage nach der Unerfüllbarkeit einer Formel ist jetzt also reduziert worden auf die Suche nach einer erfüllenden Belegung für eine (im Allgemeinen unendliche) Menge aussagenlogischer Formeln.

**Beispiel 3.2.8** *Wir betrachten noch einmal die Herbrand–Expansion der Formel $\varphi$ aus Bsp. 3.2.6. Ersetzt man hier jede atomare Formel $p(t_1, \ldots, t_n)$ ohne Variablen durch eine aussagenlogische Variable $V_{p(t_1,\ldots,t_n)}$, so ergibt sich aus $E(\varphi)$ die folgende unendliche Menge aussagenlogischer Formeln:*

$$\begin{array}{ll} \{ & V_{\mathsf{mutterVon(renate,susanne)}} \wedge \neg V_{\mathsf{mutterVon(karin,susanne)}}, \\ & V_{\mathsf{mutterVon(renate,susanne)}} \wedge \neg V_{\mathsf{mutterVon(renate,susanne)}}, \\ & V_{\mathsf{mutterVon(renate,susanne)}} \wedge \neg V_{\mathsf{mutterVon(datum(karin,werner,1982),susanne)}}, \\ & \ldots \quad \}. \end{array}$$

*Die Formel $\varphi$ und damit $E(\varphi)$ ist nun genau dann erfüllbar, wenn es eine Belegung der aussagenlogischen Variablen $V_{p(t_1,\ldots,t_n)}$ mit "wahr" oder "falsch" gibt, die die obige aussagenlogische Formelmenge erfüllt. Natürlich gibt es solch eine Belegung nicht, denn die Formel*

$$V_{\mathsf{mutterVon(renate,susanne)}} \wedge \neg V_{\mathsf{mutterVon(renate,susanne)}}$$

*ist weder für die Belegung von $V_{\mathsf{mutterVon(renate,susanne)}}$ mit "wahr" noch mit "falsch" erfüllt.*

Nach dem Endlichkeits- oder Kompaktheitssatz der Aussagenlogik ist eine (unendliche) Menge aussagenlogischer Formeln genau dann unerfüllbar, wenn sie eine *endliche* unerfüllbare Teilmenge hat (vgl. z.B. [Gra11, Satz 1.15] und [Sch00, Kap. 1.4]). Falls $E(\varphi)$ unerfüllbar ist, ist also bereits eine endliche Teilmenge von $E(\varphi)$ unerfüllbar. Man kann damit nun das folgende erste Verfahren zur Überprüfung der Unerfüllbarkeit bzw. der Folgerbarkeit geben. Dieses Verfahren ist ein Semi–Entscheidungsverfahren, d.h., falls die Folgerbarkeitsbeziehung gilt (bzw. falls die entsprechende Formel unerfüllbar ist), so terminiert das Verfahren und findet dieses auch heraus.

Das Verfahren wird als *Algorithmus von Gilmore* bezeichnet. Die Eingabe sind Formeln $\varphi_1, \ldots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ und das Ziel ist, herauszufinden, ob $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ gilt.

***Algorithmus von Gilmore***

1. *Sei $\xi$ die Formel $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$. Nach Lemma 3.0.1 gilt $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ gdw. $\psi$ unerfüllbar ist.*

2. *Überführe $\xi$ in Skolem–Normalform wie in den Sätzen 3.1.2 und 3.1.6. Die entstehende Formel $\psi$ ist erfüllbarkeitsäquivalent zur ursprünglichen Formel.*

3. *Wähle eine Aufzählung $\{\psi_1, \psi_2, \ldots\} = E(\psi)$ der Herbrand–Expansion von $\psi$ und ersetze dabei alle atomaren Formeln durch aussagenlogische Variablen. Nach Satz 3.2.3 und 3.2.7 ist $E(\psi)$ genau dann (aussagenlogisch) erfüllbar, wenn $\psi$ erfüllbar ist. Nach dem Kompaktheitssatz der Aussagenlogik ist dies genau dann der Fall, wenn alle endlichen Teilmengen von $E(\psi)$ aussagenlogisch erfüllbar sind.*

4. *Prüfe, ob $\psi_1$, $\psi_1 \wedge \psi_2$, $\psi_1 \wedge \psi_2 \wedge \psi_3$, ... aussagenlogisch erfüllbar sind (indem man alle möglichen Wahrheitsbelegungen der vorkommenden aussagenlogischen Variablen durchtestet). Sofern eine dieser Formeln nicht erfüllbar ist, brich ab und gib "**true**" zurück.*

Dieser Algorithmus terminiert also und liefert genau dann das Ergebnis "**true**" zurück, wenn $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ gilt. Sofern dies nicht gilt, terminiert der Algorithmus nicht (falls $\psi$ Variablen enthält und es unendlich viele Grundterme gibt). Der Nachteil des Algorithmus ist allerdings, dass er noch nicht sehr zielgerichtet vorgeht und daher noch sehr ineffizient ist.

**Beispiel 3.2.9** *Zur Überprüfung von*

$$\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \exists\, X\ \mathsf{mutterVon}(X, \mathsf{susanne})$$

*musste man die aussagenlogische Erfüllbarkeit der Formelmenge aus Bsp. 3.2.8 untersuchen. Hierzu genügt es nun, nur die endlichen Teilmengen zu betrachten. Wählt man die Aufzählung, die der Reihenfolge in Bsp. 3.2.8 entspricht, so erkennt man bereits im zweiten Schritt die Unerfüllbarkeit (d.h., sobald die Teilformel*

$$V_{\mathsf{mutterVon(renate,susanne)}} \wedge \neg V_{\mathsf{mutterVon(renate,susanne)}}$$

*mit betrachtet wird).*

## 3.3 Grundresolution

Um die Überprüfung auf Unerfüllbarkeit effizienter durchführen zu können, führen wir in diesem Abschnitt das Beweisverfahren der *Resolution* ein. Dabei werden wir uns in diesem Abschnitt auf die aussagenlogische Resolution beschränken und diese dann in Abschnitt 3.4 auf die prädikatenlogische Resolution erweitern. Die aussagenlogische Resolution wird auch als *Grundresolution* bezeichnet, da wir hier nur variablenfreie Formeln (d.h., Formeln mit *Grundtermen*) betrachten.

Um eine Formel der Form $\forall\, X_1, \ldots, X_n\ \psi$ in Skolem–Normalform mit dem Resolutionskalkül auf Unerfüllbarkeit zu untersuchen, muss die Formel $\psi$ zunächst in *konjunktive Normalform (KNF)* überführt werden. Solche Formeln werden dann als *Klauselmengen* dargestellt.

**Definition 3.3.1 (Konjunktive Normalform, Literal, Klausel)** *Eine Formel $\psi$ ist in* konjunktiver Normalform (KNF) *gdw. sie quantorfrei ist und die folgende Gestalt hat.*

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$$

*Hierbei sind die $L_{i,j}$* Literale*, d.h., es sind atomare oder negierte atomare Formeln der Gestalt $p(t_1, \ldots, t_k)$ oder $\neg p(t_1, \ldots, t_k)$. Zu einem Literal $L$ definieren wir sein Negat $\overline{L}$ wie folgt:*

$$\overline{L} = \begin{cases} \neg A, & \text{falls } L = A \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \\ A, & \text{falls } L = \neg A \text{ mit } A \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \end{cases}$$

*Eine Menge von Literalen wird als* Klausel *bezeichnet. Jede Formel $\psi$ in KNF wie oben entspricht der zugehörigen* Klauselmenge

$$\mathcal{K}(\psi) = \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{m,1}, \ldots, L_{m,n_m}\}\}.$$

*Eine Klausel steht für die* allquantifizierte Disjunktion *ihrer Literale und eine Klauselmenge repräsentiert die* Konjunktion *ihrer Klauseln. Wir sprechen daher im Folgenden auch von* Erfüllbarkeit *und* Folgerbarkeit *von Klauselmengen. Falls nichts explizit anderes gesagt wird, betrachten wir im Folgenden stets nur endliche Klauselmengen. Die leere Klausel wird oft als* $\square$ *geschrieben und sie ist nach Definition unerfüllbar.*

Eine Überführung in KNF (und damit in eine Darstellung als Klauselmenge) ist für jede quantorfreie Formel leicht automatisch möglich.

**Satz 3.3.2 (Überführung in KNF)** *Zu jeder quantorfreien Formel $\psi$ lässt sich automatisch eine Formel $\psi'$ in konjunktiver Normalform konstruieren, so dass $\psi$ und $\psi'$ äquivalent sind.*

**Beweis.** Zunächst werden alle Teilformeln $\psi_1 \leftrightarrow \psi_2$ in $\psi$ durch $\psi_1 \rightarrow \psi_2 \wedge \psi_2 \rightarrow \psi_1$ ersetzt. Anschließend werden alle Teilformeln $\psi_1 \rightarrow \psi_2$ in $\psi$ durch $\neg\psi_1 \vee \psi_2$ ersetzt. Die Überführung der verbleibenden Formel geschieht mit folgendem Algorithmus $KNF$, dessen Terminierung und Korrektheit offensichtlich ist. Der Algorithmus bekommt als Eingabe eine beliebige quantorfreie Formel $\psi$ ohne die Junktoren "$\leftrightarrow$" und "$\rightarrow$" und liefert als Ausgabe eine dazu äquivalente Formel in KNF.

- Falls $\psi$ atomar ist, so gib $\psi$ zurück.
- Falls $\psi = \psi_1 \wedge \psi_2$ ist, so gib $KNF(\psi_1) \wedge KNF(\psi_2)$ zurück.
- Falls $\psi = \psi_1 \vee \psi_2$ ist, so berechne $KNF(\psi_1) = \bigwedge_{i\in\{1,\ldots,m_1\}} \psi'_i$ und $KNF(\psi_2) = \bigwedge_{j\in\{1,\ldots,m_2\}} \psi''_j$. Hierbei sind $\psi'_i$ und $\psi''_i$ Disjunktionen von Literalen. Durch Anwendung des Distributivgesetzes erhält man die Formel $\bigwedge_{i\in\{1,\ldots,m_1\}, j\in\{1,\ldots,m_2\}} \psi'_i \vee \psi''_j$ in KNF, die zurückgegeben wird.
- Falls $\psi = \neg\psi_1$ ist, so berechne $KNF(\psi_1) = \bigwedge_{i\in\{1,\ldots,m\}}(\bigvee_{j\in\{1,\ldots,n_i\}} L_{i,j})$. Mehrfache Anwendung der de Morganschen Regel auf $\neg \bigwedge_{i\in\{1,\ldots,m\}}(\bigvee_{j\in\{1,\ldots,n_i\}} L_{i,j})$ liefert $\bigvee_{i\in\{1,\ldots,m\}}(\bigwedge_{j\in\{1,\ldots,n_i\}} \overline{L_{i,j}})$. Anschließend geht man wie im vorigen Fall vor und wendet das Distributivgesetz an. Dies ergibt die Formel $\bigwedge_{j_1\in\{1,\ldots,n_1\},\ldots,j_m\in\{1,\ldots,n_m\}} \overline{L_{1,j_1}} \vee \ldots \vee \overline{L_{m,j_m}}$ in KNF, die zurückgegeben wird.

$\square$

**Beispiel 3.3.3** *Als Beispiel betrachten wir die folgende Formel, wobei* $\mathsf{p}, \mathsf{q}, \mathsf{r} \in \Delta_0$ *sind.*

$$\neg(\neg\mathsf{p} \wedge (\neg\mathsf{q} \vee \mathsf{r}))$$

*Durch Anwendung der de Morganschen Regel erhält man*

$$\mathsf{p} \vee (\mathsf{q} \wedge \neg\mathsf{r}).$$

*Schließlich ergibt das Distributivgesetz die gewünschte Formel in KNF:*

$$(\mathsf{p} \vee \mathsf{q}) \wedge (\mathsf{p} \vee \neg\mathsf{r})$$

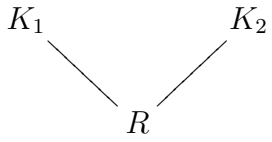
Die Grundidee im Algorithmus von Gilmore war, dass die Unerfüllbarkeit einer Formel $\psi$ der Prädikatenlogik auf die Unerfüllbarkeit einer Menge $E(\psi)$ von aussagenlogischen Formeln (bzw. von Formeln ohne Variablen) zurückgeführt werden konnte. Um die Unerfüllbarkeit von $E(\psi)$ zu überprüfen, wurde im Algorithmus von Gilmore nach einer endlichen unerfüllbaren Teilmenge gesucht und dann die Unerfüllbarkeit durch Testen aller Wahrheitsbelegungen nachgewiesen. Die Überprüfung der Unerfüllbarkeit in der Aussagenlogik lässt sich aber wesentlich zielgerichteter mit Hilfe des *Resolutionskalküls* durchführen, sofern der quantorfreie Teil der Formel in KNF vorliegt. Hierzu definieren wir den Begriff des *Resolventen* für aussagenlogische bzw. variablenfreie Klauseln.

**Definition 3.3.4 (Aussagenlogische Resolution)** *Seien $K_1$ und $K_2$ variablenfreie Klauseln. Dann ist die Klausel $R$* Resolvent *von $K_1$ und $K_2$ gdw. es ein Literal $L \in K_1$ gibt mit $\overline{L} \in K_2$ und $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$. Für eine Klauselmenge $\mathcal{K}$ definieren wir $Res(\mathcal{K}) = \mathcal{K} \cup \{R \mid R \text{ ist Resolvent zweier Klauseln aus } \mathcal{K}\}$. Weiterhin sei $Res^0(\mathcal{K}) = \mathcal{K}$ und $Res^{n+1}(\mathcal{K}) = Res(Res^n(\mathcal{K}))$ für alle $n \geq 0$. Schließlich definieren wir die Menge der aus $\mathcal{K}$ durch Resolution herleitbaren Klauseln als $Res^*(\mathcal{K}) = \bigcup_{n\geq 0} Res^n(\mathcal{K})$.*

Offensichtlich gilt $\Box \in Res^*(\mathcal{K})$ gdw. es eine Folge von Klauseln $K_1, \ldots, K_m$ gibt, so dass $K_m = \Box$ ist und so dass für alle $1 \leq i \leq m$ gilt:

- $K_i \in \mathcal{K}$ oder
- $K_i$ ist ein Resolvent von $K_j$ und $K_k$ für $j, k < i$

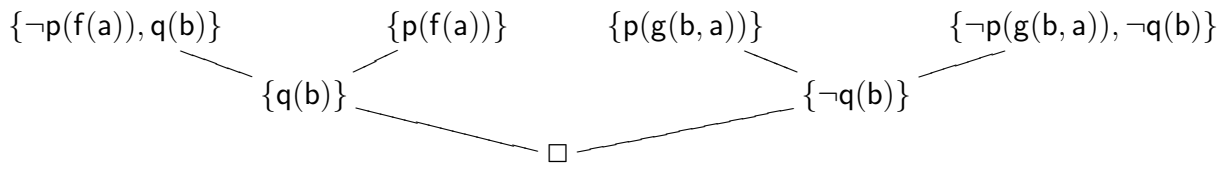
Zur Darstellung von Resolutionsbeweisen schreiben wir oftmals das folgende Diagramm, um deutlich zu machen, dass $R$ durch Resolution aus $K_1$ und $K_2$ entsteht.




**Beispiel 3.3.5** *Sei $\Delta_1 = \{\mathsf{p}, \mathsf{q}\}$, $\Sigma_0 = \{\mathsf{a}, \mathsf{b}\}$, $\Sigma_1 = \{\mathsf{f}\}$ und $\Sigma_2 = \{\mathsf{g}\}$. Betrachten wir die Klauselmenge $\{K_1, K_2, K_3, K_4\}$ über dieser Signatur mit*

$$\begin{array}{lcl} K_1 & = & \{\neg\mathsf{p}(\mathsf{f}(\mathsf{a})), \mathsf{q}(\mathsf{b})\} \\ K_2 & = & \{\mathsf{p}(\mathsf{f}(\mathsf{a}))\} \\ K_3 & = & \{\mathsf{p}(\mathsf{g}(\mathsf{b}, \mathsf{a}))\} \\ K_4 & = & \{\neg\mathsf{p}(\mathsf{g}(\mathsf{b}, \mathsf{a})), \neg\mathsf{q}(\mathsf{b})\} \end{array}$$

*Es gilt:*



Satz 3.3.7 zeigt die Korrektheit und Vollständigkeit des Resolutionskalküls in der Aussagenlogik. Eine Klauselmenge ohne Variablen ist also genau dann unerfüllbar, wenn man durch wiederholte Anwendung der Resolution die leere Klausel herleiten kann. Damit folgt, dass die Klauselmenge $\{K_1, K_2, K_3, K_4\}$ aus Bsp. 3.3.5 unerfüllbar ist. Um die Korrektheit und Vollständigkeit der aussagenlogischen Resolution zeigen zu können, beweisen wir zunächst das folgende Lemma, das besagt, dass das Hinzufügen von Resolventen eine äquivalenzerhaltende Operation ist.

**Lemma 3.3.6 (Aussagenlogisches Resolutionslemma)** *Sei $\mathcal{K}$ eine Menge von variablenfreien Klauseln. Falls $K_1, K_2 \in \mathcal{K}$ und $R$ Resolvent von $K_1$ und $K_2$ ist, dann sind $\mathcal{K}$ und $\mathcal{K} \cup \{R\}$ äquivalent.*

**Beweis.** Jede Struktur $S$, die $\mathcal{K} \cup \{R\}$ erfüllt, erfüllt auch $\mathcal{K}$ (denn eine Klauselmenge repräsentiert die *Konjunktion* ihrer Klauseln). Somit gilt $\mathcal{K} \cup \{R\} \models \mathcal{K}$.

Umgekehrt sei nun $S$ eine Struktur, die $\mathcal{K}$ erfüllt. Es existiert ein Literal $L \in K_1$ mit $\overline{L} \in K_2$ und $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$. Wir nehmen an, dass $S \not\models \mathcal{K} \cup \{R\}$ gilt. Aus $S \models \mathcal{K}$ folgt dann $S \not\models R$. Falls $S \models L$, so folgt aus $S \models K_2$ auch $S \models K_2 \setminus \{\overline{L}\}$ und damit $S \models R$. Falls $S \models \overline{L}$, so folgt aus $S \models K_1$ auch $S \models K_1 \setminus \{L\}$ und damit $S \models R$. Somit gilt auch $\mathcal{K} \models \mathcal{K} \cup \{R\}$. □

Nun können wir die Korrektheit und Vollständigkeit des aussagenlogischen Resolutionskalküls beweisen. Hierbei bedeutet *Korrektheit*, dass man mit der Resolution keine falschen Aussagen beweisen kann (d.h., dass die Klauselmenge tatsächlich unerfüllbar ist, wenn man die leere Klausel durch Resolution herleiten kann). *Vollständigkeit* bedeutet, dass alle wahren Aussagen durch Resolution beweisbar sind (d.h., dass man die leere Klausel aus jeder unerfüllbaren Klauselmenge durch Resolution herleiten kann).

**Satz 3.3.7 (Korrektheit und Vollständigkeit der aussagenlogischen Resolution)** *Sei $\mathcal{K}$ eine möglicherweise unendliche Menge von variablenfreien Klauseln. Dann ist $\mathcal{K}$ unerfüllbar gdw. $\square \in Res^*(\mathcal{K})$.*

**Beweis.** Wir zeigen zuerst die *Korrektheit* (d.h., die Richtung "$\Leftarrow$"). Aus dem Resolutionslemma 3.3.6 folgt, dass $\mathcal{K}$ und $Res(\mathcal{K})$ äquivalent sind. Durch Induktion über $n$ zeigt man sofort, dass dann auch $\mathcal{K}$ und $Res^n(\mathcal{K})$ für alle $n \in \mathbb{N}$ äquivalent sind. Aus $\square \in Res^*(\mathcal{K})$ folgt, dass es ein $n \in \mathbb{N}$ mit $\square \in Res^n(\mathcal{K})$ gibt. Somit ist $Res^n(\mathcal{K})$ unerfüllbar. Da $\mathcal{K}$ und $Res^n(\mathcal{K})$ äquivalent sind, ist dann auch $\mathcal{K}$ unerfüllbar.

Nun beweisen wir die *Vollständigkeit* (d.h., die Richtung "$\Rightarrow$"). Falls $\mathcal{K}$ unerfüllbar ist, so gibt es bereits eine endliche unerfüllbare Teilmenge $\mathcal{K}' \subseteq \mathcal{K}$ nach dem Endlichkeitssatz der Aussagenlogik. Sei $n$ die Anzahl der verschiedenen atomaren Formeln in der Klauselmenge $\mathcal{K}'$. Wir zeigen $\square \in Res^*(\mathcal{K}')$ durch Induktion über $n$.

Im Induktionsanfang betrachten wir den Fall $n = 0$. Es gibt nur zwei Klauselmengen ohne atomare Formeln, nämlich $\varnothing$ und $\{\square\}$. Da $\varnothing$ erfüllbar (sogar allgemeingültig) ist, muss somit $\mathcal{K}' = \{\square\}$ sein und daher $\square \in Res^0(\mathcal{K}')$.

Im Induktionsschluss $n > 0$ sei $A$ eine atomare Formel, die in $\mathcal{K}'$ vorkommt. Die Klauselmenge $\mathcal{K}^+$ entsteht aus $\mathcal{K}'$, indem wir alle Klauseln streichen, in denen $A$ vorkommt und indem wir $\neg A$ aus allen verbleibenden Klausel streichen. Analog dazu entsteht die Klauselmenge $\mathcal{K}^-$ aus $\mathcal{K}'$, indem wir alle Klauseln streichen, in denen $\neg A$ vorkommt und indem

wir $A$ aus allen verbleibenden Klausel streichen. Formal gilt also $\mathcal{K}^+ = \{K \setminus \{\neg A\} \mid K \in \mathcal{K}', A \notin K\}$ und $\mathcal{K}^- = \{K \setminus \{A\} \mid K \in \mathcal{K}', \neg A \notin K\}$.

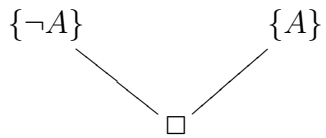
Aus der Unerfüllbarkeit von $\mathcal{K}'$ folgt auch die Unerfüllbarkeit der Klauselmenge $\mathcal{K}^+$. Gäbe es nämlich eine Struktur $S$ mit $S \models \mathcal{K}^+$, so könnten wir $S$ zu einer Struktur $S'$ mit $S' \models A$ erweitern und erhalten dann $S' \models \mathcal{K}'$. Analog dazu ist auch $\mathcal{K}^-$ unerfüllbar.

Aus der Induktionshypothese folgt somit $\square \in Res^*(\mathcal{K}^+)$ und $\square \in Res^*(\mathcal{K}^-)$. Aus $\square \in Res^*(\mathcal{K}^+)$ folgt, dass es eine Folge von Klauseln $K_1, \ldots, K_m$ gibt, so dass $K_m = \square$ ist und so dass für alle $1 \leq i \leq m$ gilt:

- $K_i \in \mathcal{K}^+$ oder
- $K_i$ ist ein Resolvent von $K_j$ und $K_k$ für $j, k < i$

Falls die benötigten $K_i \in \mathcal{K}^+$ auch bereits in $\mathcal{K}'$ enthalten sind (d.h., falls die entsprechenden Klauseln aus $\mathcal{K}'$ nicht das Literal $\neg A$ enthielten), so ist dies auch eine Herleitung aus $\mathcal{K}'$. Dann gilt also $K_1, \ldots, K_m \in Res^*(\mathcal{K}')$ und somit $\square \in Res^*(\mathcal{K}')$. Ansonsten erhalten wir durch Wiedereinfügen von $\neg A$ eine Folge von Klauseln $K'_1, \ldots, K'_m$, die beweist, dass $\{\neg A\} \in Res^*(\mathcal{K}')$ ist. Der Grund ist folgender: Wenn $C_i$ Resolvent von $C_j$ und $C_k$ ist, dann ist $C_i \cup \{\neg A\}$ Resolvent von $C_j \cup \{\neg A\}$ und $C_k$:


Analog dazu folgt aus $\square \in Res^*(\mathcal{K}^-)$, dass $\square \in Res^*(\mathcal{K}')$ oder $\{A\} \in Res^*(\mathcal{K}')$ ist. Aus $\{\neg A\}, \{A\} \in Res^*(\mathcal{K}')$ folgt aber wiederum $\square \in Res^*(\mathcal{K}')$:




□

Nun können wir den Algorithmus von Gilmore zum *Grundresolutionsverfahren* verbessern. Wie in Schritt 1 und 2 des Algorithmus von Gilmore überführen wir das Folgerbarkeitsproblem zunächst in das Problem, die Unerfüllbarkeit einer Formel $\forall X_1, \ldots, X_n\ \psi$ in Skolem–Normalform zu zeigen. Anschließend gehen wir wie folgt vor:

***Grundresolutionsalgorithmus***

3. *Überführe $\psi$ wie in Satz 3.3.2 in KNF bzw. in die entsprechende Klauselmenge $\mathcal{K}(\psi)$.*
4. *Wähle eine Aufzählung $\{K_1, K_2, \ldots\}$ aller Grundinstanzen der Klauseln aus $\mathcal{K}(\psi)$. Dies entspricht der Herbrand–Expansion von $\mathcal{K}(\psi)$, d.h., der Klauselmenge $\{\sigma(K) \mid K \in \mathcal{K}(\psi), \sigma$ ist Grundsubstitution, d.h., $\mathcal{V}(\sigma(K)) = \varnothing\}$. Hierbei bedeutet die Anwendung einer Substitution auf eine Klausel, dass die Substitution auf alle Literale der Klausel angewendet wird.*

5. *Berechne* $Res^*(\{K_1, K_2\})$, $Res^*(\{K_1, K_2, K_3\})$, ... *Sofern eine dieser Mengen die leere Klausel* $\square$ *enthält, brich ab und gib "**true**" zurück.*

**Beispiel 3.3.8** *Als Beispiel betrachten wir die Signatur aus Bsp. 3.3.5. Wir wollen die Unerfüllbarkeit der folgenden Formel in Skolem–Normalform zeigen, deren quantorfreier Teil* $\psi$ *bereits in KNF vorliegt:*

$$\forall X, Y\ (\neg \mathsf{p}(X) \vee \neg \mathsf{p}(\mathsf{f}(\mathsf{a})) \vee \mathsf{q}(Y)) \wedge \mathsf{p}(Y) \wedge (\neg \mathsf{p}(\mathsf{g}(\mathsf{b}, X)) \vee \neg \mathsf{q}(\mathsf{b}))$$

*Damit erhält man die folgende Klauselmenge* $\mathcal{K}(\psi)$*:*

$$\{\ \{\neg \mathsf{p}(X),\ \neg \mathsf{p}(\mathsf{f}(\mathsf{a})),\ \mathsf{q}(Y)\},\ \{\mathsf{p}(Y)\},\ \{\neg \mathsf{p}(\mathsf{g}(\mathsf{b}, X)),\ \neg \mathsf{q}(\mathsf{b})\}\ \}$$

*Wir wählen nun eine Aufzählung* $\{K_1, K_2, K_3, K_4, \ldots\}$ *der Grundinstanzen der drei Klauseln, wobei* $K_1, \ldots, K_4$ *die variablenfreien Klauseln aus Bsp. 3.3.5 sind. Wie in Bsp. 3.3.5 gezeigt wurde, kann man dann die leere Klausel herleiten und somit auch die Unerfüllbarkeit der ursprünglichen Formel zeigen. Man erkennt, dass hierbei sowohl* $K_2$ *als auch* $K_3$ *unterschiedliche Instanzen derselben Klausel* $\{\mathsf{p}(Y)\}$ *sind. Dies kann nötig sein, um die Unerfüllbarkeit beweisen zu können. Ebenso erkennt man auch, dass durch die Instanzenbildung mehrere Literale in einer Klausel gleich gemacht werden können. So ist* $K_1$ *eine Instanz der Klausel* $\{\neg \mathsf{p}(X),\ \neg \mathsf{p}(\mathsf{f}(\mathsf{a})),\ \mathsf{q}(Y)\}$*, die aber nur noch zwei Literale enthält, da durch die Substitution von* $X$ *durch* $\mathsf{f}(\mathsf{a})$ *die ersten beiden Literale der Klausel gleich gemacht werden.*

Der folgende Satz zeigt die Korrektheit und Vollständigkeit des Grundresolutionsalgorithmus. Dieser Algorithmus terminiert also und liefert genau dann das Ergebnis "**true**" zurück, wenn $\forall X_1, \ldots, X_n\ \psi$ unerfüllbar ist. Sofern dies nicht gilt, terminiert der Algorithmus nicht (sofern die Menge der Grundinstanzen unendlich ist).

**Satz 3.3.9 (Korrektheit und Vollständigkeit des Grundresolutionsalgorithmus)**

*(a) Falls eine Klauselmenge* $\mathcal{K}$ *unerfüllbar ist, dann existiert eine endliche Menge von Grundinstanzen von Klauseln aus* $\mathcal{K}$ *(d.h. eine endliche Teilmenge von* $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma$ *ist Grundsubstitution*$\}$*), die ebenfalls unerfüllbar ist.*

*(b) Sei* $\forall X_1, \ldots, X_n\ \psi$ *eine Formel in Skolem–Normalform, wobei* $\psi$ *in KNF ist. Dann ist* $\forall X_1, \ldots, X_n\ \psi$ *unerfüllbar gdw. es eine Folge von Klauseln* $K_1, \ldots, K_m$ *gibt, so dass* $K_m = \square$ *ist und so dass für alle* $1 \leq i \leq m$ *gilt:*

- $K_i$ *ist Grundinstanz einer Klausel aus* $\mathcal{K}(\psi)$ *oder*
- $K_i$ *ist ein Resolvent von* $K_j$ *und* $K_k$ *für* $j, k < i$

**Beweis.**

(a) Sei $\mathcal{K} = \{K_1, \ldots, K_r\}$, sei $\psi_i$ die Formel, die aus der nicht-allquantifizierten Disjunktion der Literale in $K_i$ entsteht (für alle $1 \leq i \leq r$), sei $\psi = \psi_1 \wedge \ldots \wedge \psi_r$ und

seien $X_1, \ldots, X_n$ die Variablen in $\psi$. Dann entspricht $\mathcal{K}$ der Formel $\forall X_1, \ldots, X_n\ \psi$ in Skolem–Normalform, wobei $\psi$ in KNF ist (und es gilt $\mathcal{K} = \mathcal{K}(\psi)$). Es gilt:

$\mathcal{K}$ ist unerfüllbar
gdw. $\forall X_1, \ldots, X_n\ \psi$ ist unerfüllbar
gdw. $E(\forall X_1, \ldots, X_n\ \psi) = \{\sigma(\psi) \mid \sigma \text{ ist Grundsubstitution}\}$ ist unerfüllbar (Satz 3.2.7)
gdw. $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ ist Grundsubstitution}\}$ ist unerfüllbar

Nach dem Endlichkeitssatz der Aussagenlogik ist dies gleichbedeutend damit, dass es eine endliche unerfüllbare Teilmenge von $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ Grundsubstitution}\}$ gibt.

(b) Wie in Teil (a) gilt, dass $\forall X_1, \ldots, X_n\ \psi$ genau dann unerfüllbar ist, wenn eine endliche Teilmenge von $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ ist Grundsubstitution}\}$ unerfüllbar ist. Dies ist nach Satz 3.3.7 äquivalent dazu, dass die leere Klausel $\square$ durch Resolution aus einer endlichen Teilmenge von $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ ist Grundsubstitution}\}$ herleitbar ist. $\square$

Der Nachteil des Algorithmus ist allerdings, dass die Suche nach geeigneten Grundinstanzen noch nicht zielgerichtet verläuft und man daher auf sehr ineffiziente Weise alle Grundinstanzen durchprobieren muss. Es müssen z.B. Entscheidungen für einige der Grundsubstitutionen in "vorausschauender Weise" getroffen werden, um spätere Resolutionen zu ermöglichen. Dies legt eine Modifikation nahe, bei der man die Substitutionen "zurückhaltend" wählt und immer nur insoweit Substitutionen durchführt, wie dies für den direkt nachfolgenden Resolutionsschritt benötigt wird. Dazu erlaubt man jetzt statt Grundsubstitutionen auch Substitutionen mit beliebigen Termen. Dies führt zur *prädikatenlogischen Resolution*, die im nächsten Abschnitt eingeführt wird.

## 3.4 Prädikatenlogische Resolution und Unifikation

Die angesprochene Idee der "zurückhaltenden" Substitutionen lässt sich am folgenden Beispiel verdeutlichen.

**Beispiel 3.4.1** *Wir betrachten die folgende Klauselmenge, wobei* $\mathsf{p}, \mathsf{q} \in \Delta_1$, $\mathsf{f} \in \Sigma_1$ *und* $\mathsf{a} \in \Sigma_0$ *sind.*

$$\{\ \{\mathsf{p}(X), \neg\mathsf{q}(X)\},\ \{\neg\mathsf{p}(\mathsf{f}(Y))\},\ \{\mathsf{q}(\mathsf{f}(\mathsf{a}))\}\ \}$$

*Um die ersten beiden Klauseln zu resolvieren, benutzen wir nun die Substitution* $\{X/\mathsf{f}(Y)\}$. *Nach Anwendung dieser Substitution ergeben sich die Literale* $\mathsf{p}(X)[X/\mathsf{f}(Y)] = \mathsf{p}(\mathsf{f}(Y))$ *und* $\neg\mathsf{p}(\mathsf{f}(Y))[X/\mathsf{f}(Y)] = \neg\mathsf{p}(\mathsf{f}(Y))$, *die zueinander komplementär sind. Die Substitution* $\{X/\mathsf{f}(Y)\}$ *ist daher ein* Unifikator *von* $\{\mathsf{p}(X), \mathsf{p}(\mathsf{f}(Y))\}$. *Der Resolvent ist dann die Klausel* $\{\neg\mathsf{q}(X)[X/\mathsf{f}(Y)]\} = \{\neg\mathsf{q}(\mathsf{f}(Y))\}$ *mit dem verbleibenden zweiten Literal der ersten Klausel.*

*Der Vorteil ist, dass durch die Substitution* $\{X/\mathsf{f}(Y)\}$ *noch nicht festgelegt wurde, wie man* $Y$ *anschließend instantiiert. Es zeigt sich, dass man anschließend die Substitution* $\{Y/\mathsf{a}\}$ *verwenden muss, um schließlich die leere Klausel herzuleiten. Dies muss man aber*

*nicht von vornherein erkennen (d.h., man muss nicht gleich die Substitution* $\{X/\mathsf{f}(\mathsf{a})\}$ *verwenden), sondern man benutzt jeweils nur Substitutionen wie* $\{X/\mathsf{f}(Y)\}$*, die so allgemein wie möglich sind. Mit anderen Worten, wir wählen im ersten Resolutionsschritt den* allgemeinsten *Unifikator von* $\{\mathsf{p}(X), \mathsf{p}(\mathsf{f}(Y))\}$ *und nicht den weniger allgemeinen Unifikator* $\{X/\mathsf{f}(\mathsf{a})\}$.

Die folgende Definition führt das Konzept der Unifikation formal ein.

**Definition 3.4.2 (Unifikation)** *Eine Klausel* $K = \{L_1, \ldots, L_n\}$ *ist* unifizierbar *gdw. es eine Substitution* $\sigma$ *mit* $\sigma(L_1) = \ldots = \sigma(L_n)$ *gibt (d.h.,* $|\sigma(K)| = 1$*). Solch eine Substitution heißt ein* Unifikator *von* $K$*. Ein Unifikator* $\sigma$ *heißt* allgemeinster *Unifikator (most general unifier, mgu), falls es für jeden Unifikator* $\sigma'$ *eine Substitution* $\delta$ *gibt mit* $\sigma'(X) = \delta(\sigma(X))$ *für alle* $X \in \mathcal{V}$*.*

Falls eine Klausel unifizierbar ist, so existiert auch ein allgemeinster Unifikator, der bis auf Variablenumbenennungen[1] eindeutig ist (vgl. z.B. [Gie11]).

Für eine Klausel ist es entscheidbar, ob sie unifizierbar ist. Der folgende erste Unifikationsalgorithmus wurde von *J. Robinson* [Rob65] entwickelt. Als Eingabe erhält er eine Klausel $K = \{L_1, \ldots, L_n\}$, die zu unifizieren ist.

***Unifikationsalgorithmus***

1. *Sei* $\sigma = \varnothing$ *die leere (oder "identische") Substitution.*

2. *Falls* $|\sigma(K)| = 1$ *ist, dann brich ab und gib* $\sigma$ *als mgu von* $K$ *aus.*

3. *Sonst durchsuche alle* $\sigma(L_i)$ *parallel von links nach rechts, bis in zwei Literalen die gelesenen Zeichen verschieden sind.*

4. *Falls keines der beiden Zeichen eine Variable ist, dann brich mit* Clash Failure *ab.*

5. *Sonst sei* $X$ *die Variable und* $t$ *der Teilterm im anderen Literal (hierbei kann* $t$ *auch eine Variable sein). Falls* $X$ *in* $t$ *vorkommt, dann brich mit* Occur Failure *ab. (Diese Überprüfung bezeichnet man als* Occur Check*.)*

6. *Sonst setze* $\sigma = \{X/t\} \circ \sigma$ *und gehe zurück zu Schritt 2.*

Hierbei bedeutet $\sigma_1 \circ \sigma_2$ die Komposition (oder "Hintereinanderausführung") von Substitutionen. Es gilt also $(\sigma_1 \circ \sigma_2)(X) = \sigma_1(\sigma_2(X))$.

[1] Eine Substitution $\sigma$ heißt *Variablenumbenennung* gdw. sie injektiv ist und $\sigma(X) \in \mathcal{V}$ für alle $X \in \mathcal{V}$ gilt.

**Beispiel 3.4.3** *Sei* $\mathsf{q} \in \Delta_1$, $\mathsf{p} \in \Delta_2$, $\mathsf{f}, \mathsf{g} \in \Sigma_2$, $\mathsf{h} \in \Sigma_1$ *und* $\mathsf{a} \in \Sigma_0$.

*Als Beispiel für einen Clash Failure betrachten wir die Klausel*

$$\{\ \mathsf{q}(\mathsf{f}(X,Y)),\ \mathsf{q}(\mathsf{g}(X,Y))\ \}.$$

*An der ersten Stelle, an der sich die beiden Literale unterscheiden, steht einmal das Funktionssymbol* $\mathsf{f}$ *und einmal das Funktionssymbol* $\mathsf{g}$*. Für keine Instantiierung der Variablen* $X$ *und* $Y$ *werden diese beiden Literale gleich, denn die Instantiierungen können diese verschiedenen Funktionssymbole nicht ändern. Diese Klausel ist daher nicht unifizierbar.*

*Als Beispiel für einen Occur Failure betrachten wir die folgende Klausel:*

$$\{\ \mathsf{q}(X),\ \mathsf{q}(\mathsf{h}(X))\ \}.$$

*An der ersten unterschiedlichen Stelle steht einmal die Variable* $X$ *und einmal der Term* $\mathsf{h}(X)$*, der die Variable* $X$ *enthält. Für keine Instantiierung von* $X$ *können diese Terme bzw. die ursprünglichen Literale gleich gemacht werden. Die Klausel ist daher ebenfalls nicht unifizierbar.*

*Schließlich wenden wir den Unifikationsalgorithmus auf die Klausel aus den folgenden zwei Literalen an:*

$$\begin{array}{l}\neg\mathsf{p}(\mathsf{f}(Z,\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(Z))\\ \neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),W),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))\\ \quad\ \uparrow\end{array}$$

*Der Pfeil deutet die erste Stelle an, an der sich die beiden Literale unterscheiden. Somit ergibt sich* $\sigma = \{Z/\mathsf{f}(U,V)\}$*. Nun wendet man den bereits gefundenen Teilunifikator* $\sigma$ *auf die Literale an:*

$$\begin{array}{l}\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(U,V)))\\ \neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),W),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))\\ \qquad\qquad\ \uparrow\end{array}$$

*Es ergibt sich* $\sigma = \{W/\mathsf{g}(\mathsf{a},Y)\} \circ \sigma = \{Z/\mathsf{f}(U,V), W/\mathsf{g}(\mathsf{a},Y)\}$*. Durch Anwendung von* $\sigma$ *erhält man:*

$$\begin{array}{l}\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(U,V)))\\ \neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))\\ \qquad\qquad\qquad\qquad\ \uparrow\end{array}$$

*Nun ergibt sich* $\sigma = \{U/\mathsf{a}\} \circ \sigma = \{Z/\mathsf{f}(\mathsf{a},V), W/\mathsf{g}(\mathsf{a},Y), U/\mathsf{a}\}$*. Die Anwendung von* $\sigma$ *führt zu:*

$$\begin{array}{l}\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(\mathsf{a},V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(\mathsf{a},V)))\\ \neg\mathsf{p}(\mathsf{f}(\mathsf{f}(\mathsf{a},V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))\\ \qquad\qquad\qquad\qquad\ \uparrow\end{array}$$

*Schließlich erhält man* $\sigma = \{Y/V\} \circ \sigma = \{Z/\mathsf{f}(\mathsf{a},V), W/\mathsf{g}(\mathsf{a},V), U/\mathsf{a}, Y/V\}$*. Danach sind die beiden instantiierten Literale identisch, so dass dies der gesuchte mgu ist.*

Der folgende Satz zeigt die Terminierung und Korrektheit des Unifikationsalgorithmus. (In [Gie11] findet sich eine formalere Fassung des Unifikationsalgorithmus als Sammlung von vier Transformationsregeln, die auch das "parallele Durchsuchen der Literale von links nach rechts" exakt formalisiert. Dort wird auch die Terminierung und Korrektheit dieses Algorithmus genau bewiesen.)

**Satz 3.4.4 (Terminierung und Korrektheit des Unifikationsalgorithmus)** *Der Unifikationsalgorithmus terminiert für jede Klausel $K$ und er ist korrekt, d.h., er liefert einen mgu für die Klausel $K$ gdw. $K$ unifizierbar ist.*

**Beweis.** Die Terminierung des Algorithmus folgt, da in jedem Durchlauf der Schleife von Schritt 2 – 6 die Zahl der Variablen in $\sigma(K)$ um 1 abnimmt.

Falls der Algorithmus mit Erfolg terminiert und eine Substitution $\sigma$ ausgibt, so ist $\sigma$ offensichtlich ein Unifikator von $K$, da $|\sigma(K)| = 1$ gilt. Sofern die Klausel $K$ also nicht-unifizierbar ist, muss der Algorithmus daher mit einem Clash oder Occur Failure abbrechen.

Es bleibt zu zeigen, dass bei jeder unifizierbaren Klausel auch tatsächlich ein Unifikator gefunden wird und dass dieser Unifikator dann auch ein *allgemeinster* Unifikator ist.

Sei $m \geq 0$ die Anzahl der Schleifendurchläufe, die bei Eingabe der Klausel $K$ stattfinden. Für alle $0 \leq i \leq m$ sei $\sigma_i$ der Wert von $\sigma$ nach dem $i$-ten Schleifendurchlauf. Wir zeigen die folgende Behauptung für alle $0 \leq i \leq m$:

$$\text{Für jeden Unifikator } \sigma' \text{ von } K \text{ gilt } \sigma' = \sigma' \circ \sigma_i. \tag{3.1}$$

Aus der Behauptung (3.1) folgt, dass zum Schluss nicht mit Clash oder Occur Failure abgebrochen werden kann. Denn würde im $(m+1)$-ten Schleifendurchlauf abgebrochen, so wäre $\sigma_m(K)$ nicht unifizierbar. Da aber $K$ unifizierbar ist, hat $K$ einen Unifikator $\sigma' = \sigma' \circ \sigma_m$. Somit ist $\sigma'$ auch Unifikator von $\sigma_m(K)$.

Da die Schleife nur $m$ mal durchlaufen wird, muss dann also $|\sigma_m(K)| = 1$ gelten, d.h., $\sigma_m$ ist Unifikator von $K$. Da außerdem für jeden Unifikator $\sigma'$ von $K$ eine Substitution $\delta = \sigma'$ mit $\sigma' = \delta \circ \sigma_m$ existiert, ist $\sigma_m$ dann auch mgu von $K$.

Wir beweisen nun die Behauptung (3.1) durch Induktion über $i$. Im Induktionsanfang $i = 0$ ist $\sigma_0 = \varnothing$ die Identität. Daher gilt dann auch $\sigma' = \sigma' \circ \sigma_0$ für alle $\sigma'$.

Im Induktionsschritt $i > 0$ wird im $i$-ten Schleifendurchlauf eine Variable $X$ im einen Literal und ein Term $t$ im anderen Literal gefunden und es ergibt sich $\sigma_i = \{X/t\} \circ \sigma_{i-1}$. Für jeden Unifikator $\sigma'$ von $K$ gilt nach der Induktionshypothese $\sigma' = \sigma' \circ \sigma_{i-1}$. Damit folgt:

$$\begin{array}{lll} & \sigma' \circ \sigma_i & \\ = & \sigma' \circ \{X/t\} \circ \sigma_{i-1} & \text{(Def. von } \sigma_i\text{)} \\ = & \sigma' \circ \sigma_{i-1} & \text{(da } \sigma' \circ \{X/t\} = \sigma'\text{)} \end{array}$$

Um $\sigma' \circ \{X/t\} = \sigma'$ zu zeigen, erkennt man zunächst, dass die beiden Substitutionen auf allen Variablen $Y \neq X$ offensichtlich identisch sind. Bei der Variablen $X$ ergibt sich $(\sigma' \circ \{X/t\})(X) = \sigma'(t) = \sigma'(X)$, da $\sigma'$ Unifikator von $\sigma_{i-1}(K)$ ist (denn $|\sigma'(K)| = |\sigma'(\sigma_{i-1}(K))| = 1$) und jeder Unifikator von $\sigma_{i-1}(K)$ auch die Terme $X$ und $t$ unifizieren muss. □

Mit Hilfe der Unifikation können wir nun die prädikatenlogische Resolution definieren.

**Definition 3.4.5 (Prädikatenlogische Resolution)** *Seien $K_1$ und $K_2$ Klauseln. Dann ist die Klausel $R$* Resolvent *von $K_1$ und $K_2$ gdw. die folgenden drei Bedingungen gelten:*

- *Es gibt Variablenumbenennungen $\nu_1$ und $\nu_2$, so dass $\nu_1(K_1)$ und $\nu_2(K_2)$ keine gemeinsamen Variablen enthalten.*
- *Es gibt Literale $L_1, \ldots, L_m \in \nu_1(K_1)$ und Literale $L'_1, \ldots, L'_n \in \nu_2(K_2)$ mit $n, m \geq 1$, so dass $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ mit einem mgu $\sigma$ unifizierbar ist.*
- $R = \sigma((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L'_1, \ldots, L'_n\}))$

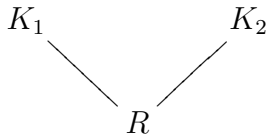
*Für eine Klauselmenge $\mathcal{K}$ definieren wir wie bisher (d.h. wie bei der aussagenlogischen Resolution in Def. 3.3.4):*

$$\begin{aligned} Res(\mathcal{K}) &= \mathcal{K} \cup \{R \mid R \text{ ist Resolvent zweier Klauseln aus } \mathcal{K}\} \\ Res^0(\mathcal{K}) &= \mathcal{K} \\ Res^{n+1}(\mathcal{K}) &= Res(Res^n(\mathcal{K})) \text{ für alle } n \geq 0 \\ Res^*(\mathcal{K}) &= \bigcup_{n \geq 0} Res^n(\mathcal{K}) \end{aligned}$$

Offensichtlich ist die aussagenlogische Resolution ein Spezialfall der prädikatenlogischen Resolution, denn bei variablenfreien Klauseln fällt diese Definition mit Def. 3.3.4 zusammen. Analog zur aussagenlogischen Resolution gilt $\square \in Res^*(\mathcal{K})$ gdw. es eine Folge von Klauseln $K_1, \ldots, K_m$ gibt, so dass $K_m = \square$ ist und so dass für alle $1 \leq i \leq m$ gilt:

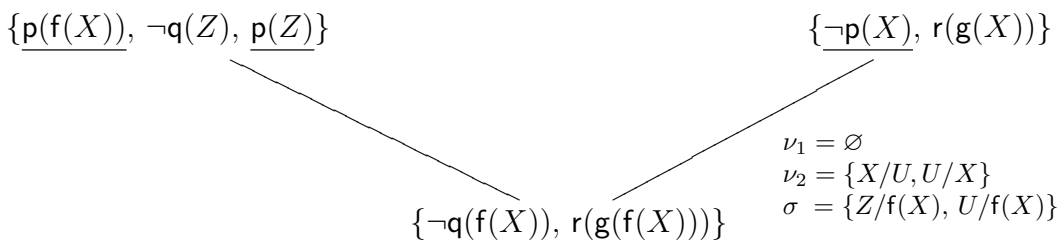
- $K_i \in \mathcal{K}$ oder
- $K_i$ ist ein Resolvent von $K_j$ und $K_k$ für $j, k < i$

Zur Darstellung von Resolutionsbeweisen schreiben wir wieder das folgende Diagramm, um deutlich zu machen, dass $R$ durch Resolution aus $K_1$ und $K_2$ entsteht.




Eine genauere Darstellung ist möglich, indem man die eliminierten Literale unterstreicht und die Umbenennungen und den Unifikator explizit angibt.

**Beispiel 3.4.6** *Als Beispiel hierzu betrachten wir den folgenden Resolutionsschritt, wobei $\mathsf{p}, \mathsf{q}, \mathsf{r} \in \Delta_1$ und $\mathsf{f}, \mathsf{g} \in \Sigma_1$ ist.*



Für den Korrektheitsbeweis der prädikatenlogischen Resolution müssen wir das Resolutionslemma 3.3.6 von der Aussagenlogik auf die Prädikatenlogik erweitern.

**Lemma 3.4.7 (Prädikatenlogisches Resolutionslemma)** *Sei $\mathcal{K}$ eine Menge von Klauseln. Falls $K_1, K_2 \in \mathcal{K}$ und $R$ Resolvent von $K_1$ und $K_2$ ist, dann sind $\mathcal{K}$ und $\mathcal{K} \cup \{R\}$ äquivalent.*

**Beweis.** Wie in der Aussagenlogik folgt aus $S \models \mathcal{K} \cup \{R\}$ trivialerweise $S \models \mathcal{K}$ für alle Strukturen $S$. Es genügt, hierbei *Strukturen* statt *Interpretationen* zu betrachten, da jede Klausel die *allquantifizierte* Disjunktion ihrer Literale repräsentiert und jede Klauselmenge die Konjunktion ihrer Klauseln. Somit gilt $\mathcal{K} \cup \{R\} \models \mathcal{K}$.

Umgekehrt sei nun $S$ eine Struktur, die $\mathcal{K}$ erfüllt. Es gilt

$$R = \sigma((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L'_1, \ldots, L'_n\})).$$

Hierbei sind $\nu_1$ und $\nu_2$ Variablenumbenennungen, so dass $\nu_1(K_1)$ und $\nu_2(K_2)$ keine gemeinsamen Variablen enthalten. Außerdem sind $L_1, \ldots, L_m \in \nu_1(K_1)$ und $L'_1, \ldots, L'_n \in \nu_2(K_2)$ mit $n, m \geq 1$, so dass $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ mit dem mgu $\sigma$ unifizierbar sind. Es gilt also $\sigma(L_1) = \ldots = \sigma(L_m) = L$ und $\sigma(L'_1) = \ldots = \sigma(L'_n) = \overline{L}$ für ein Literal $L$.

Wir nehmen an, dass $S \not\models \mathcal{K} \cup \{R\}$ gilt. Aus $S \models \mathcal{K}$ folgt dann $S \not\models R$. Sei

$$\nu_1(K_1) = \{L_1, \ldots, L_m, L_{m+1}, \ldots, L_p\} \qquad \text{und} \qquad \nu_2(K_2) = \{L'_1, \ldots, L'_n, L'_{n+1}, \ldots, L'_q\}$$

mit $p \geq m$ und $q \geq n$. Dann entsteht $R$ durch Allquantifizierung der Formel

$$\sigma(L_{m+1} \vee \ldots \vee L_p \vee L'_{n+1} \vee \ldots \vee L'_q).$$

Sei $S = (\mathcal{A}, \alpha)$. Es existiert also eine Interpretation $I = (\mathcal{A}, \alpha, \beta)$ mit

$$I \not\models \sigma(L_{m+1} \vee \ldots \vee L_p \vee L'_{n+1} \vee \ldots \vee L'_q).$$

Sei $\sigma = \{X_1/t_1, \ldots, X_k/t_k\}$ und sei $I'$ die Interpretation $I[\![X_1/I(t_1), \ldots, X_k/I(t_k)]\!]$. Nach dem Substitutionslemma 2.2.3 gilt dann

$$I' \not\models L_{m+1} \vee \ldots \vee L_p \vee L'_{n+1} \vee \ldots \vee L'_q. \tag{3.2}$$

Da aber $S \models K_1$ und $S \models K_2$ gilt, folgt auch $S \models \nu_1(K_1)$ und $S \models \nu_2(K_2)$ und somit

$$I' \models L_1 \vee \ldots \vee L_m \vee L_{m+1} \vee \ldots \vee L_p \qquad \text{und} \qquad I' \models L'_1 \vee \ldots \vee L'_n \vee L'_{n+1} \vee \ldots \vee L'_q.$$

Mit (3.2) folgt

$$I' \models L_1 \vee \ldots \vee L_m \qquad \text{und} \qquad I' \models L'_1 \vee \ldots \vee L'_n$$

und mit dem Substitutionslemma 2.2.3 ergibt sich

$$I \models \sigma(L_1 \vee \ldots \vee L_m) \qquad \text{und} \qquad I \models \sigma(L'_1 \vee \ldots \vee L'_n).$$

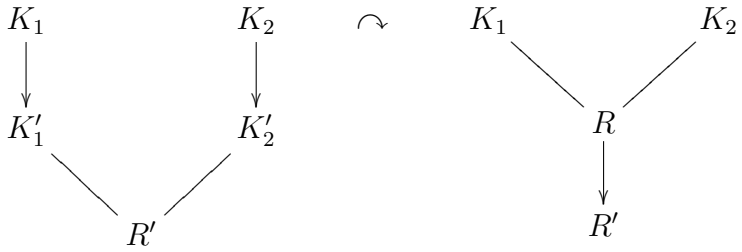
Wir erhalten also den folgenden Widerspruch:

$$I \models L \qquad \text{und} \qquad I \models \overline{L}.$$

□

Das folgende Lemma ist nötig, um die Vollständigkeit der prädikatenlogischen Resolution auf die Vollständigkeit der aussagenlogischen Resolution zurückführen zu können. Es zeigt, dass die (aussagenlogische) Resolution von Grundinstanzen zweier Klauseln in die prädikatenlogische Resolution der beiden ursprünglichen Klauseln "geliftet" werden kann.

**Lemma 3.4.8 (Lifting–Lemma)** *Seien $K_1$ und $K_2$ zwei Klauseln mit Grundinstanzen $K_1'$ und $K_2'$. Falls $R'$ (aussagenlogischer) Resolvent von $K_1'$ und $K_2'$ ist, so gibt es einen (prädikatenlogischen) Resolventen $R$ von $K_1$ und $K_2$, so dass $R'$ eine Grundinstanz von $R$ ist. Das folgende Diagramm veranschaulicht das Lifting–Lemma, wobei Pfeile Grundinstanzen andeuten.*




**Beweis.** Seien $\nu_1$ und $\nu_2$ Variablenumbenennungen, so dass $\nu_1(K_1)$ und $\nu_2(K_2)$ keine gemeinsamen Variablen besitzen. Da $K_i'$ jeweils Grundinstanz von $K_i$ ist, ist es dann auch Grundinstanz von $\nu_i(K_i)$. Da außerdem $\nu_1(K_1)$ und $\nu_2(K_2)$ keine gemeinsamen Variablen besitzen, kann man eine *gemeinsame* Grundsubstitution $\sigma$ für $\nu_1(K_1)$ und $\nu_2(K_2)$ wählen, so dass $\sigma(\nu_1(K_1)) = K_1'$ und $\sigma(\nu_2(K_2)) = K_2'$.

Da $R'$ Resolvent von $K_1'$ und $K_2'$ ist, gibt es ein Literal $L \in K_1'$ mit $\overline{L} \in K_2'$, so dass $R' = (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\})$ ist.

Wir wählen nun alle Urbilder von $L$ bzw. $\overline{L}$ unter $\sigma$ aus den Klauseln $\nu_1(K_1)$ und $\nu_2(K_2)$. Seien also $L_1, \ldots, L_m \in \nu_1(K_1)$ und $L_1', \ldots, L_n' \in \nu_2(K_2)$ *alle* Literale aus den jeweiligen Klauseln mit $L = \sigma(L_1) = \ldots = \sigma(L_m)$ und $\overline{L} = \sigma(L_1') = \ldots = \sigma(L_n')$. Hierbei gilt $m, n \geq 1$, da ja $L \in \sigma(\nu_1(K_1))$ und $\overline{L} \in \sigma(\nu_2(K_2))$ ist.

Da $\sigma$ ein Unifikator von $\{\overline{L_1}, \ldots, \overline{L_m}, L_1', \ldots, L_n'\}$ ist, existiert also auch ein mgu $\sigma'$. Daher haben $K_1$ und $K_2$ den Resolventen $R = \sigma'((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \ldots, L_n'\}))$.

Da $\sigma$ ein Unifikator von $\{\overline{L_1}, \ldots, \overline{L_m}, L_1', \ldots, L_n'\}$ ist und $\sigma'$ der mgu ist, gilt demnach $\sigma = \delta \circ \sigma'$ für eine Substitution $\delta$. Hierbei ist $\delta$ eine Grundsubstitution, da $\sigma$ ebenfalls eine Grundsubstitution ist. Nun kann man zeigen, dass auch der Resolvent $R'$ von $K_1'$ und $K_2'$ eine Grundinstanz des Resolventen $R$ von $K_1$ und $K_2$ ist:

$$\begin{aligned} R' &= (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}) \\ &= (\sigma(\nu_1(K_1)) \setminus \{L\}) \cup (\sigma(\nu_2(K_2)) \setminus \{\overline{L}\}) \\ &= \sigma((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \ldots, L_n'\})) \\ &= \delta(\sigma'((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \ldots, L_n'\}))) \\ &= \delta(R) \end{aligned}$$

Der Schritt von der zweiten zur dritten Zeile ist korrekt, da $L_1, \ldots, L_m$ bzw. $L_1', \ldots, L_n'$ *alle* Urbilder von $L$ bzw. $\overline{L}$ in $\nu_1(K_1)$ bzw. $\nu_2(K_2)$ sind. □

**Beispiel 3.4.9** *Das folgende Beispiel illustriert das Lifting–Lemma. Hierzu betrachten wir die Klauseln* $K_1 = \{\mathsf{p}(\mathsf{f}(X)), \neg\mathsf{q}(Z), \mathsf{p}(Z)\}$ *und* $K_2 = \{\neg\mathsf{p}(X), \mathsf{r}(\mathsf{g}(X))\}$ *aus Bsp. 3.4.6. Durch Instantiierung mit der Grundsubstitution* $\{X/\mathsf{a}, Z/\mathsf{f}(\mathsf{a})\}$ *erhält man die Grundinstanz* $K_1' = \{\mathsf{p}(\mathsf{f}(\mathsf{a})), \neg\mathsf{q}(\mathsf{f}(\mathsf{a}))\}$ *und durch Instantiierung mit* $\{X/\mathsf{f}(\mathsf{a})\}$ *entsteht* $K_2' = \{\neg\mathsf{p}(\mathsf{f}(\mathsf{a})), \mathsf{r}(\mathsf{g}(\mathsf{f}(\mathsf{a})))\}$. *Hierbei sei* $\mathsf{a} \in \Sigma_0$. *Die aussagenlogische Resolution von* $K_1'$ *und* $K_2'$ *ergibt* $R' = \{\neg\mathsf{q}(\mathsf{f}(\mathsf{a})), \mathsf{r}(\mathsf{g}(\mathsf{f}(\mathsf{a})))\}$. *Man erkennt, dass dieser Resolvent eine Instanz des Resolventen* $R = \{\neg\mathsf{q}(\mathsf{f}(X)), \mathsf{r}(\mathsf{g}(\mathsf{f}(X)))\}$ *von* $K_1$ *und* $K_2$ *aus Bsp. 3.4.6 ist.*

Nun können wir die Korrektheit und Vollständigkeit der prädikatenlogischen Resolution beweisen.

**Satz 3.4.10 (Korrektheit & Vollständigkeit der prädikatenlog. Resolution)** *Sei* $\mathcal{K}$ *eine (endliche) Menge von Klauseln. Dann ist* $\mathcal{K}$ *unerfüllbar gdw.* $\square \in Res^*(\mathcal{K})$.

**Beweis.** Die *Korrektheit* (d.h., die Richtung "$\Leftarrow$") folgt aus dem prädikatenlogischen Resolutionslemma 3.4.7 vollkommen analog zum Korrektheitsbeweis der aussagenlogischen Resolution (Satz 3.3.7).

Nun beweisen wir die *Vollständigkeit* (d.h. die Richtung "$\Rightarrow$"). Da $\mathcal{K}$ unerfüllbar ist, folgt aus Satz 3.3.9 (a), dass es eine endliche unerfüllbare Menge von Grundinstanzen von Klauseln aus $\mathcal{K}$ gibt. Aus der Vollständigkeit der aussagenlogischen Resolution (Satz 3.3.7) folgt also, dass es eine Folge von Klauseln $K_1', \ldots, K_m'$ gibt, so dass $K_m' = \square$ und so dass für alle $1 \leq i \leq m$ gilt:

- $K_i'$ ist Grundinstanz einer Klausel $K \in \mathcal{K}$ oder
- $K_i'$ ist ein Resolvent von $K_j'$ und $K_k'$ für $j, k < i$

Mit dem Lifting–Lemma 3.4.8 erzeugen wir nun eine Folge von Klauseln $K_1, \ldots, K_m$, wobei $K_i'$ jeweils Grundinstanz von $K_i$ ist und wobei $K_i \in Res^*(\mathcal{K})$ gilt. Hierbei gehen wir wie folgt vor:

- Falls $K_i'$ Grundinstanz einer Klausel $K \in \mathcal{K}$ ist, so wählen wir $K_i := K$.
- Sonst ist $K_i'$ ein Resolvent von $K_j'$ und $K_k'$ mit $j, k < i$. Es existieren bereits Klauseln $K_j$ und $K_k$ mit $K_j, K_k \in Res^*(\mathcal{K})$, so dass $K_j'$ und $K_k'$ Grundinstanzen von $K_j$ und $K_k$ sind. Nach dem Lifting–Lemma 3.4.8 existiert dann ein Resolvent von $K_j$ und $K_k$, so dass $K_i'$ eine Grundinstanz dieses Resolventen ist. Wir wählen diesen Resolventen als $K_i$. Da $K_j, K_k \in Res^*(\mathcal{K})$ gilt, folgt damit auch $K_i \in Res^*(\mathcal{K})$.

Demnach folgt, dass $K_m' = \square$ eine Grundinstanz von $K_m$ ist, d.h., $K_m = \square$. Außerdem folgt, dass $K_m = \square \in Res^*(\mathcal{K})$. $\square$

Wir erhalten also nun den folgenden Algorithmus, um das Folgerbarkeitsproblem zu lösen. Wie in Schritt 1 und 2 des Algorithmus von Gilmore überführen wir das Folgerbarkeitproblem wieder in das Problem, die Unerfüllbarkeit einer Formel $\forall X_1, \ldots, X_n\, \psi$ in Skolem–Normalform zu zeigen. Anschließend gehen wir wie folgt vor:

***Resolutionsalgorithmus***

3. *Überführe $\psi$ wie in Satz 3.3.2 in KNF bzw. in die entsprechende Klauselmenge $\mathcal{K}(\psi)$.*

4. *Berechne $Res^*(\mathcal{K}(\psi))$. Sofern die leere Klausel gefunden wurde, brich ab und gib "**true**" zurück. Sofern $Res^*(\mathcal{K}(\psi))$ komplett berechnet wurde, brich ab und gib "**false**" zurück.*

Dieser Semi–Entscheidungsalgorithmus kann auch mit "**false**" abbrechen, denn falls $Res^*(\mathcal{K}(\psi))$ endlich ist und die leere Klausel nicht enthält, folgt die Erfüllbarkeit der Klauselmenge $\mathcal{K}(\psi)$ aus der Vollständigkeit des Resolutionskalküls. Im Allgemeinen ist $Res^*(\mathcal{K}(\psi))$ allerdings unendlich und dann terminiert der obige Algorithmus bei erfüllbaren Klauselmengen $\mathcal{K}(\psi)$ nicht.

Die Effizienz dieses Algorithmus lässt sich allerdings noch weiter verbessern, denn bislang müssen noch *alle* möglichen Resolutionsschritte zwischen allen Klauseln betrachtet werden. Hierbei müssen natürlich auch die Klauseln betrachtet werden, die im Laufe der Resolution entstanden sind. Im folgenden Abschnitt zeigen wir, dass sich die Resolution noch weiter zur *linearen* Resolution einschränken lässt, ohne dabei die Vollständigkeit zu verlieren.

## 3.5 Einschränkungen der Resolution

Das Problem beim Unerfüllbarkeitsnachweis durch Resolution ist, dass es sehr viele Resolutionsmöglichkeiten gibt, was zu einer kombinatorischen Explosion führt. Unser Ziel ist daher, den Suchraum durch Verwendung spezieller Resolutionsstrategien zu verkleinern, ohne jedoch dabei die Vollständigkeit zu verlieren. In diesem Abschnitt werden wir drei Einschränkungen der Resolution vorstellen:

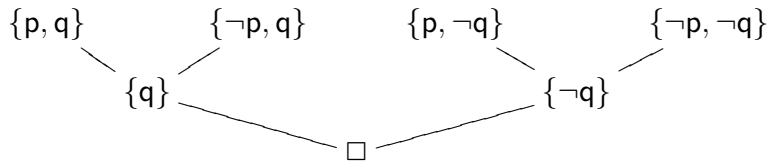
- In Abschnitt 3.5.1 stellen wir die *lineare* Resolution vor, die ebenfalls für beliebige Klauselmengen vollständig ist.
- In Abschnitt 3.5.2 schränken wir die lineare Resolution weiter zur Input–Resolution ein. Diese eingeschränkte Form der Resolution ist allerdings nicht mehr für beliebige Klauselmengen, sondern nur noch für sogenannte *Horn–Klauselmengen* vollständig. Aus diesem Grund sind die in Logikprogrammen verwendeten Klauseln nur *Horn–Klauseln.*
- Auf Horn–Klauselmengen kann man die Input–Resolution noch weiter zur *SLD–Resolution* einschränken. Dies ist das Resolutionsprinzip, das bei der Logikprogrammierung verwendet wird.

### 3.5.1 Lineare Resolution

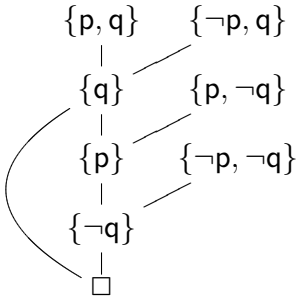
Die Idee der linearen Resolution besteht darin, dass bei jedem Resolutionsschritt eine der beiden Elternklauseln der zuletzt erzeugte Resolvent sein muss.

**Definition 3.5.1 (Lineare Resolution)** *Sei $\mathcal{K}$ eine Klauselmenge. Die leere Klausel $\square$ ist aus der Klausel $K$ in $\mathcal{K}$* linear resolvierbar *gdw. es eine Folge von Klauseln $K_1, \ldots, K_m$ gibt, so dass $K_1 = K \in \mathcal{K}$ und $K_m = \square$ ist und so dass für alle $2 \leq i \leq m$ gilt: $K_i$ ist ein Resolvent von $K_{i-1}$ und einer Klausel aus $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}$.*

**Beispiel 3.5.2** *Wir betrachten die folgende (aussagenlogische) Klauselmenge mit $\mathsf{p}, \mathsf{q} \in \Delta_0$, deren Unerfüllbarkeit hier mit einem Resolutionsbeweis nachgewiesen wird:*




*Diese Herleitung entspricht aber keinem* linearen *Resolutionsbeweis, denn wenn man im ersten Schritt den Resolvent $\{\mathsf{q}\}$ bildet, so müsste dieser Resolvent dann eine der Elternklauseln im nächsten Schritt sein. Eine Herleitung der leeren Klausel mit linearer Resolution wäre in diesem Beispiel wie folgt möglich:*




Der folgende Satz zeigt, dass auch die lineare Resolution korrekt und vollständig ist. Im Resolutionsalgorithmus lässt sich also nun die Berechnung von $Res^*(\mathcal{K}(\psi))$ dadurch einschränken, dass man nur noch lineare Resolutionsfolgen betrachtet, die mit einer Klausel aus $\mathcal{K}(\psi)$ beginnen.

**Satz 3.5.3 (Korrektheit und Vollständigkeit der linearen Resolution)** *Sei $\mathcal{K}$ eine Menge von Klauseln. Dann ist $\mathcal{K}$ unerfüllbar gdw. $\square$ aus einer Klausel $K$ in $\mathcal{K}$ linear resolvierbar ist. Falls $\mathcal{K}$ eine* minimale *unerfüllbare Menge ist (d.h., falls für jedes $K \in \mathcal{K}$ die Menge $\mathcal{K} \setminus \{K\}$ erfüllbar ist), dann ist $\square$ sogar aus* jeder *Klausel $K$ in $\mathcal{K}$ linear resolvierbar.*

**Beweis.** Die Korrektheit (d.h., die Richtung "$\Leftarrow$") folgt sofort aus Satz 3.4.10, da jeder lineare Resolutionsschritt auch ein Resolutionsschritt ist.

Wir zeigen daher jetzt die *Vollständigkeit* (d.h. die Richtung "$\Rightarrow$"). Hierzu beweisen wir zuerst die Vollständigkeit der linearen *Grundresolution*, d.h., wir betrachten den Fall, dass $\mathcal{K}$ eine Menge von variablenfreien Klauseln ist.

**Vollständigkeit der aussagenlogischen linearen Resolution**

Sei $\mathcal{K}_{min} \subseteq \mathcal{K}$ eine *minimale* unerfüllbare Teilmenge von $\mathcal{K}$, d.h., für jedes $K \in \mathcal{K}_{min}$ ist $\mathcal{K}_{min} \backslash \{K\}$ erfüllbar. Offensichtlich gilt $\mathcal{K}_{min} \neq \varnothing$, da die leere Klauselmenge erfüllbar (sogar allgemeingültig) ist. Wir zeigen nun, dass $\square$ aus *jeder* Klausel $K$ in $\mathcal{K}_{min}$ linear resolvierbar ist. Hierzu verwenden wir Induktion über die Anzahl $n$ der verschiedenen atomaren Formeln in der Klauselmenge $\mathcal{K}_{min}$.

*Induktionsanfang: $n = 0$*

Somit gilt $\mathcal{K}_{min} = \{\square\}$ und daher ist $\square$ linear aus der einzigen Klausel $\square$ in $\mathcal{K}_{min}$ resolvierbar.

*Induktionsschritt, 1. Fall: $n > 0$, $K \in \mathcal{K}_{min}$ mit $|K| = 1$*

Es gilt also $K = \{L\}$ für ein Literal $L$. Die Klauselmenge $\mathcal{K}^+$ entsteht aus $\mathcal{K}_{min}$, indem wir alle Klauseln streichen, in denen $L$ vorkommt und indem wir $\overline{L}$ aus allen verbleibenden Klauseln streichen. Aus der Unerfüllbarkeit von $\mathcal{K}_{min}$ folgt (wie im Beweis von Satz 3.3.7) auch die Unerfüllbarkeit der Klauselmenge $\mathcal{K}^+$. Außerdem enthält $\mathcal{K}^+$ höchstens $n - 1$ verschiedene atomare Formeln. Sei $\mathcal{K}^+_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}^+$. Dann folgt aus der Induktionshypothese, dass $\square$ aus *jeder* Klausel in $\mathcal{K}^+_{min}$ linear resolvierbar ist.

Da $\mathcal{K}_{min}$ eine *minimale* unerfüllbare Klauselmenge ist, gilt $\mathcal{K}^+_{min} \not\subseteq \mathcal{K}_{min}$. Demnach existiert also eine Klausel $K^+ \in \mathcal{K}^+_{min}$ mit $K^+ \cup \{\overline{L}\} \in \mathcal{K}_{min}$. Nach der Induktionshypothese existiert eine Herleitung von $\square$ aus $K^+$ in $\mathcal{K}^+_{min}$ mit linearer Resolution. Es gibt also eine Folge von Klauseln $K_1, \ldots, K_m$, so dass $K_1 = K^+$ und $K_m = \square$ ist und so dass für alle $2 \leq i \leq m$ gilt: $K_i$ ist ein Resolvent von $K_{i-1}$ und einer Klausel aus $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}^+$. Daraus konstruieren wir eine lineare Resolutionsfolge $K, ..., K_1, ..., K_2, \ldots, K_m = \square$ in $\mathcal{K}_{min}$:

Wir zeigen durch Induktion über $i$, dass es für alle $1 \leq i \leq m$ eine lineare Resolutionsfolge $K, \ldots, K_i$ in $\mathcal{K}_{min}$ gibt. Bei $i = 1$ gilt: $K, K_1$ ist eine Resolutionsfolge, denn die Resolution von $K = \{L\}$ und $K_1 \cup \{\overline{L}\}$ ergibt $K_1$. Bei $i > 1$ gibt es bereits eine lineare Resolutionsfolge $K, \ldots, K_{i-1}$ in $\mathcal{K}_{min}$. Falls $K_i$ Resolvent von $K_{i-1}$ und einem $K_j$ mit $j < i$ ist, so ist damit auch $K, \ldots, K_{i-1}, K_i$ eine lineare Resolutionsfolge in $\mathcal{K}_{min}$. Ansonsten ist $K_i$ Resolvent von $K_{i-1}$ und einer Klausel $K' \in \mathcal{K}^+$. Falls auch $K' \in \mathcal{K}_{min}$ gilt, so ist $K, \ldots, K_{i-1}, K_i$ wieder eine lineare Resolutionsfolge in $\mathcal{K}_{min}$. Sonst gilt $K' \cup \{\overline{L}\} \in \mathcal{K}_{min}$ nach der Definition von $\mathcal{K}^+$. Daher ist $K, \ldots, K_{i-1}, K_i \cup \{\overline{L}\}, K_i$ eine lineare Resolutionsfolge, wobei der letzte Resolutionsschritt durch Resolution von $K_i \cup \{\overline{L}\}$ mit $K = \{L\}$ geschieht.

*Induktionsschritt, 2. Fall: $n > 0$, $K \in \mathcal{K}_{min}$ mit $|K| > 1$*

Wir wählen ein beliebiges Literal $L \in K$ und setzen $K^- = K \setminus \{L\}$. Die Klauselmenge $\mathcal{K}^-$ entsteht aus $\mathcal{K}_{min}$, indem wir alle Klauseln streichen, in denen $\overline{L}$ vorkommt und indem wir $L$ aus allen verbleibenden Klausel streichen. Aus der Unerfüllbarkeit von $\mathcal{K}_{min}$ folgt wieder (wie im Beweis von Satz 3.3.7) die Unerfüllbarkeit der Klauselmenge $\mathcal{K}^-$. Außerdem enthält

$\mathcal{K}^-$ höchstens $n-1$ verschiedene atomare Formeln. Sei $\mathcal{K}^-_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}^-$. Dann folgt aus der Induktionshypothese, dass $\square$ aus *jeder* Klausel in $\mathcal{K}^-_{min}$ linear resolvierbar ist.

Es gilt $K^- \in \mathcal{K}^-_{min}$. Der Grund ist wie folgt: Zum einen haben wir $K^- \in \mathcal{K}^-$, da $K$ das Literal $\overline{L}$ nicht enthält. (Sonst gälte $L, \overline{L} \in K$ und dann wäre $K$ allgemeingültig und würde nicht in der *minimalen* unerfüllbaren Klauselmenge $\mathcal{K}_{min}$ vorkommen.) Es gilt auch o.B.d.A. $K^- \in \mathcal{K}^-_{min}$, denn $\mathcal{K}^- \setminus \{K^-\}$ ist erfüllbar. Der Grund hierfür ist, dass $\mathcal{K}_{min} \setminus \{K\}$ aufgrund der Minimalität von $\mathcal{K}_{min}$ erfüllbar sein muss. Es gibt also eine Interpretation $I \models \mathcal{K}_{min} \setminus \{K\}$. Da $\mathcal{K}_{min}$ unerfüllbar ist, folgt $I \not\models K$ und da $L \in K$ ist, gilt $I \not\models L$. Damit folgt aus $I \models \mathcal{K}_{min} \setminus \{K\}$ auch $I \models \mathcal{K}^-_{min} \setminus \{K^-\}$.

Nach der Induktionshypothese existiert also eine Herleitung von $\square$ aus $K^-$ in $\mathcal{K}^-_{min}$ mit linearer Resolution. Es gibt also eine Folge von Klauseln $K_1, \ldots, K_m$, so dass $K_1 = K^-$ und $K_m = \square$ ist und so dass für alle $2 \leq i \leq m$ gilt: $K_i$ ist ein Resolvent von $K_{i-1}$ und einer Klausel aus $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}^-_{min}$. Daraus konstruieren wir durch Wiedereinfügen von $L$ eine lineare Resolutionsfolge aus $K$ in $\mathcal{K}_{min}$. Für die Folge $K_1 \cup \{L\}, \ldots, K_m \cup \{L\}$ gilt: $K_1 \cup \{L\} = K \in \mathcal{K}_{min}$, $K_m \cup \{L\} = \{L\}$ und für alle $2 \leq i \leq m$ gilt: $K_i \cup \{L\}$ ist ein Resolvent von $K_{i-1} \cup \{L\}$ und einer Klausel aus $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}_{min}$. Man kann also aus $K$ die Klausel $\{L\}$ durch lineare Resolution herleiten.

Nun ist aber $(\mathcal{K}_{min} \setminus \{K\}) \cup \{\{L\}\}$ unerfüllbar. Der Grund ist, dass bereits $(\mathcal{K}_{min} \setminus \{K\}) \cup \{K\}$ unerfüllbar ist und dass $L \in K$ gilt. Jede minimale unerfüllbare Teilmenge von $(\mathcal{K}_{min} \setminus \{K\}) \cup \{\{L\}\}$ enthält die Klausel $\{L\}$, denn $\mathcal{K}_{min} \setminus \{K\}$ ist erfüllbar (aufgrund der Minimalität von $\mathcal{K}_{min}$). Also existiert aufgrund des 1. Falls eine lineare Herleitung von $\square$ aus der Klausel $\{L\}$ in $\mathcal{K}_{min} \setminus \{K\}$. Diese Herleitung fügen wir an die bisherige Herleitung von $\{L\}$ an und erhalten so insgesamt eine lineare Herleitung von $\square$ aus der Klausel $K$ in $\mathcal{K}_{min}$.

**Vollständigkeit der prädikatenlogischen linearen Resolution**

Nun beweisen wir auch die Vollständigkeit der linearen Resolution für die Prädikatenlogik. Hierzu gehen wir analog zum Vollständigkeitsbeweis der Resolution (Satz 3.4.10) vor, indem wir den Vollständigkeitsbeweis der aussagenlogischen linearen Resolution auf die Prädikatenlogik "liften".

Da $\mathcal{K}$ unerfüllbar ist, folgt aus Satz 3.3.9 (a), dass es eine endliche unerfüllbare Menge von Grundinstanzen von Klauseln aus $\mathcal{K}$ gibt. Aus der oben gezeigten Vollständigkeit der linearen aussagenlogischen Resolution folgt also, dass es eine Folge von Klauseln $K'_1, \ldots, K'_m$ gibt, so dass $K'_1$ Grundinstanz einer Klausel $K \in \mathcal{K}$ ist, $K'_m = \square$, und so dass für alle $2 \leq i \leq m$ gilt: $K'_i$ ist ein Resolvent von $K'_{i-1}$ und einer Klausel aus $\{K'_1, \ldots, K'_{i-1}\} \cup \{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ ist Grundsubstitution}\}$. Wie im Beweis von Satz 3.4.10 kann man nun durch Anwendung des Lifting–Lemmas 3.4.8 daraus eine lineare Herleitung der leeren Klausel aus einer Klausel $K$ in $\mathcal{K}$ erzeugen.

Falls $\mathcal{K}$ eine *minimale* unerfüllbare Klauselmenge ist, so muss die endliche unerfüllbare Menge ihrer Grundinstanzen auch Grundinstanzen von jeder Klausel aus $\mathcal{K}$ enthalten. Nun folgt aus dem Vollständigkeitsbeweis der aussagenlogischen linearen Resolution, dass es für jede Klausel $K$ aus $\mathcal{K}$ eine Herleitung der leeren Klausel aus einer Grundinstanz von $K$ mit aussagenlogischer linearer Resolution gibt. Durch das Lifting–Lemma kann man wieder

eine lineare Herleitung der leeren Klausel aus $K$ erzeugen. □

### 3.5.2 Input– und SLD–Resolution

Um die Möglichkeiten der Resolution noch weiter zu reduzieren, schränken wir die lineare Resolution nun noch weiter ein. Bei der linearen Resolution muss eine der Elternklauseln in jedem Resolutionsschritt der letzte Resolvent sein. Die andere Elternklausel konnte jedoch noch frei gewählt werden (d.h., es konnte eine Klausel aus der ursprünglichen Klauselmenge oder aber ein bereits früher gebildeter Resolvent sein). Wir verbieten nun die letzte Möglichkeit: nun muss in jedem Schritt zwischen dem zuletzt gebildeten Resolvent und einer der ursprünglichen "Eingabe"–Klauseln resolviert werden. Aus diesem Grund bezeichnet man diese Einschränkung als *Input–Resolution*.

**Definition 3.5.4 (Input–Resolution)** *Sei $\mathcal{K}$ eine Klauselmenge. Die leere Klausel $\square$ ist aus der Klausel $K$ in $\mathcal{K}$ durch* Input–Resolution *herleitbar gdw. es eine Folge von Klauseln $K_1, \ldots, K_m$ gibt, so dass $K_1 = K \in \mathcal{K}$ und $K_m = \square$ ist und so dass für alle $2 \leq i \leq m$ gilt: $K_i$ ist ein Resolvent von $K_{i-1}$ und einer Klausel aus $\mathcal{K}$.*

Aus der Definition folgt sofort, dass die Input–Resolution ein Spezialfall der linearen Resolution ist, d.h., jeder Input–Resolutionsbeweis ist auch ein linearer Resolutionsbeweis. Der Umkehrschluss gilt natürlich nicht. In der Tat ist die Input–Resolution im Unterschied zur linearen Resolution nicht mehr vollständig, wie das folgende Beispiel zeigt.

**Beispiel 3.5.5** *Wir betrachten wieder die unerfüllbare Klauselmenge aus Bsp. 3.5.2. Während wir dort die Unerfüllbarkeit mit linearer Resolution nachweisen konnten, gelingt dies mit Input–Resolution nicht. Durch Resolution zweier Klauseln aus der ursprünglichen Klauselmenge lassen sich nur die folgenden Klauseln herleiten:*

$$\{\mathsf{q}\}, \{\neg\mathsf{q}\}, \{\mathsf{p}\}, \{\neg\mathsf{p}\}, \{\mathsf{q}, \neg\mathsf{q}\}, \{\mathsf{p}, \neg\mathsf{p}\}$$

*Falls man eine der ersten vier entstandenen Klauseln wieder mit einer Klausel der ursprünglichen Menge resolviert, so entsteht wieder eine der ersten vier Klauseln. Wenn man eine der letzten beiden (allgemeingültigen) Klauseln mit einer Klausel der ursprünglichen Menge resolviert, so erhält man die Klausel der ursprünglichen Menge. Eine Herleitung der leeren Klausel ist also nur dann möglich, wenn man zwei der durch Resolution entstandenen Klauseln (wie z.B. $\{\mathsf{q}\}$ und $\{\neg\mathsf{q}\}$) miteinander resolviert. Dies ist jedoch bei Input–Resolution nicht zulässig.*

Während die Input–Resolution also auf beliebigen Klauselmengen nicht vollständig ist, ist sie dennoch auf der eingeschränkten Menge der sogenannten *Hornklauseln* vollständig. Aus diesem Grund verwendet man in der Logikprogrammierung keine beliebigen Klauseln, sondern nur Hornklauseln.

**Definition 3.5.6 (Hornklausel)** *Eine Klausel $K$ ist eine* Hornklausel *gdw. sie höchstens ein positives Literal enthält (d.h., höchstens eines ihrer Literale ist eine atomare Formel und die anderen Literale sind negierte atomare Formeln). Eine Hornklausel heißt* negativ,

*falls sie nur negative Literale enthält (d.h., falls sie die Gestalt* $\{\neg A_1, \ldots, \neg A_k\}$ *für atomare Formeln* $A_1, \ldots, A_k$ *hat). Eine Hornklausel heißt* definit*, falls sie ein positives Literal enthält (d.h., falls sie die Gestalt* $\{B, \neg C_1, \ldots, \neg C_n\}$ *für atomare Formeln* $B, C_1, \ldots, C_n$ *hat).*

Eine Menge definiter Hornklauseln entspricht also einer Konjunktion von Implikationen. Beispielsweise ist die Hornklausel

$$\{\ \{\mathsf{p}, \neg\mathsf{q}\},\ \{\neg\mathsf{r}, \neg\mathsf{p}, \mathsf{s}\},\ \{\mathsf{s}\}\ \}$$

äquivalent zur Formel

$$(\ (\mathsf{p} \vee \neg\mathsf{q})\ \wedge\ (\neg\mathsf{r} \vee \neg\mathsf{p} \vee \mathsf{s})\ \wedge\ \mathsf{s}\ )$$

und damit auch zur folgenden Formel:

$$(\ (\mathsf{q} \to \mathsf{p})\ \wedge\ (\mathsf{r} \wedge \mathsf{p} \to \mathsf{s})\ \wedge\ \mathsf{s}\ )$$

Man erkennt den Zusammenhang zur Logikprogrammierung:

- *Fakten* sind definite Hornklauseln ohne negative Literale (d.h., sie enthalten genau ein positives Literal). Ein Beispiel ist die Klausel $\{\mathsf{s}\}$. In der Logikprogrammierung schreibt man:

  ```
  s.
  ```

- *Regeln* sind definite Hornklauseln mit negativen Literalen. Ein Beispiel ist die Klausel $\{\neg\mathsf{r}, \neg\mathsf{p}, \mathsf{s}\}$. In der Logikprogrammierung schreibt man:

  ```
  s :- r,p.
  ```

- *Anfragen* sind negative Hornklauseln. Ein Beispiel ist die Klausel $\{\neg\mathsf{p}, \neg\mathsf{q}\}$. In der Logikprogrammierung schreibt man:

  ```
  ?- p,q.
  ```

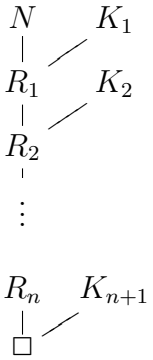
Die Betrachtung von Hornklauseln ist eine echte Einschränkung, denn es gibt nicht zu jeder Klauselmenge eine äquivalente Menge von Hornklauseln. Dies gilt bereits in der Aussagenlogik, denn bereits die Klauselmenge $\{\mathsf{p}, \mathsf{q}\}$ ist ein solches Beispiel. In der Praxis reicht die Einschränkung auf Hornklauseln aber schon oftmals aus. Der Vorteil dieser Einschränkung ist, dass sich der Erfüllbarkeitstest sehr viel effizienter automatisch durchführen lässt. In der Aussagenlogik ist die Erfüllbarkeit von Hornklauselmengen in polynomieller Zeit entscheidbar (vgl. z.B. [Gra11, Satz 1.12]), während das Erfüllbarkeitsproblem bei beliebigen aussagenlogischen Formeln bekanntlich NP–vollständig ist.

Bei prädikatenlogischen Hornklauseln ist die (Un)erfüllbarkeit nach wie vor unentscheidbar. Aber auch in der Prädikatenlogik bewirkt die Einschränkung auf Hornklauseln einen deutlichen Effizienzgewinn. Der Grund ist, dass wir uns nun auf Input–Resolution einschränken können, da diese auf Hornklauselmengen vollständig ist.

Auf Hornklauseln kann man die Input–Resolution sogar noch weiter zur *SLD–Resolution* einschränken, ohne die Vollständigkeit zu verlieren.

**Definition 3.5.7 (SLD–Resolution)** *Sei $\mathcal{K}$ eine Hornklauselmenge mit $\mathcal{K} = \mathcal{K}^d \uplus \mathcal{K}^n$, wobei $\mathcal{K}^d$ die definiten Klauseln und $\mathcal{K}^n$ die negativen Klauseln von $\mathcal{K}$ enthält. Die leere Klausel $\square$ ist aus der Klausel $K$ in $\mathcal{K}^n$ durch* SLD–Resolution *herleitbar gdw. es eine Folge von Klauseln $K_1, \ldots, K_m$ gibt, so dass $K_1 = K \in \mathcal{K}^n$ und $K_m = \square$ ist und so dass für alle $2 \leq i \leq m$ gilt: $K_i$ ist ein Resolvent von $K_{i-1}$ und einer Klausel aus $\mathcal{K}^d$.*

Man erkennt sofort, dass in einer SLD–Resolution alle Klauseln $K_1, \ldots, K_m$ negativ sind. SLD–Resolutionen haben also die folgende Gestalt:




Hierbei sind $K_1, \ldots, K_{n+1} \in \mathcal{K}^d$ definite Hornklauseln aus der Eingabemenge, $N \in \mathcal{K}^n$ ist eine negative Hornklausel aus der Eingabemenge und die Resolventen $R_1, \ldots, R_n$ sind ebenfalls negative Hornklauseln.

Die Abkürzung "SLD" steht für "**l**inear resolution with **s**election function for **d**efinite clauses". Die "selection function" soll jeweils im Resolutionsschritt von $N$ zu $R_1$ bzw. im Resolutionsschritt von $R_{i-1}$ zu $R_i$ die Literale von $N$ bzw. $R_{i-1}$ auswählen, mit denen resolviert wird. Wir ignorieren diese Selektionsfunktion momentan noch und erlauben in der obigen Definition die Resolution mit beliebigen Literalen. (Wir betrachten momentan also eigentlich die sogenannte "LUSH–Resolution", wobei "LUSH" für "**l**inear resolution with **u**nrestricted **s**election for **H**orn clauses" steht). Neben der Wahl des zu resolvierenden Literals ist die andere Wahlmöglichkeit bei der SLD–Resolution noch die Wahl der zu verwendenden Eingabeklausel aus $\mathcal{K}^d$. Wie diese beiden Wahlmöglichkeiten bei der Logikprogrammierung noch weiter eingeschränkt werden, wird in Abschnitt 4.3 genauer diskutiert.

Der folgende Satz zeigt die Vollständigkeit der SLD–Resolution auf Hornklauseln.

**Satz 3.5.8 (Korrektheit und Vollständigkeit der SLD–Resolution)** *Sei $\mathcal{K}$ eine Menge von Hornklauseln. Dann ist $\mathcal{K}$ unerfüllbar gdw. $\square$ aus einer negativen Klausel $N$ in $\mathcal{K}$ durch SLD–Resolution herleitbar ist.*

**Beweis.** Die Korrektheit (d.h., die Richtung "$\Leftarrow$") ist mit Satz 3.4.10 wieder offensichtlich, da jeder SLD–Resolutionsschritt auch ein Resolutionsschritt ist.

Wir zeigen daher jetzt die *Vollständigkeit* (d.h. die Richtung "$\Rightarrow$"). Sei $\mathcal{K}_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}$. Jede Menge definiter Hornklauseln ist erfüllbar, denn die Interpretation, die *alle* atomaren Formeln erfüllt, ist auch Modell jeder definiten Hornklausel. Daher muss $\mathcal{K}_{min}$ auch eine negative Hornklausel $N$ enthalten. Nach dem Vollständigkeitssatz der linearen Resolution (Satz 3.5.3) ist $\square$ aus jeder Klausel von $\mathcal{K}_{min}$

linear resolvierbar. Es existiert also auch ein linearer Resolutionsbeweis, der $\square$ aus der negativen Klausel $N$ herleitet.

Dieser lineare Resolutionsbeweis ist auch ein SLD–Resolutionsbeweis. Der Grund ist, dass alle Resolventen negative Klauseln sind und dass negative Klauseln nicht mit negativen Klauseln (sondern nur mit definiten Klauseln) resolviert werden können. $\square$

Der Semi–Entscheidungsalgorithmus zur Überprüfung der Folgerbarkeit bzw. der Unerfüllbarkeit aus Abschnitt 3.4 lässt sich also nun verbessern. Sofern wir eine Klauselmenge erhalten, die nur aus Hornklauseln besteht, so berechnen wir nur noch alle SLD–Resolutionsfolgen, wobei wir aber mit jeder beliebigen negativen Klausel starten müssen. Sofern die Klauselmenge auch Nicht–Hornklauseln enthält, so müssen wir alle linearen Resolutionsfolgen berechnen.

Die SLD–Resolution für Hornklauselmengen bildet die operationale Basis für die Logikprogrammierung, wie im nächsten Kapitel deutlich wird. Dabei schränkt man sich allerdings darauf ein, dass in jedem Resolutionsschritt jeweils nur zwischen *zwei* Literalen resolviert wird, nicht zwischen beliebig vielen. Bei der bisherigen Definition der prädikatenlogischen Resolution werden die Literale $L_1, \ldots, L_m$ aus der ersten Elternklausel und die Literale $L'_1, \ldots, L'_n$ aus der zweiten Elternklausel gelöscht, falls $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ unifizierbar ist. Wir verwenden stattdessen jetzt eine Einschränkung, bei der $m = n = 1$ ist. Man spricht hierbei von *binärer* Resolution. Wir zeigen in Satz 3.5.10, dass diese Einschränkung bei Hornklauseln die Vollständigkeit der Resolution erhält. Hingegen ist die binäre Resolution bei allgemeinen Klauseln nicht vollständig.

**Beispiel 3.5.9** *Ein Gegenbeispiel zur Vollständigkeit der binären Resolution ist folgende Klauselmenge. Hierbei ist* $\mathsf{p} \in \Delta_1$.

$$\{\ \{\mathsf{p}(X), \mathsf{p}(Y)\},\ \{\neg\mathsf{p}(U), \neg\mathsf{p}(V)\}\ \}$$

*Durch Verwendung des mgu* $\{X/V, Y/V, U/V\}$ *der Menge* $\{\mathsf{p}(X), \mathsf{p}(Y), \mathsf{p}(U), \mathsf{p}(V)\}$ *lässt sich die leere Klausel in einem Schritt mit normaler prädikatenlogischer Resolution herleiten. Die Klauselmenge ist also unerfüllbar. Mit binärer Resolution kann man stattdessen nur Klauseln wie* $\{\mathsf{p}(Y), \neg\mathsf{p}(V)\}$ *herleiten. Resolviert man zwei dieser Klauseln, so erhält man wieder eine solche Klausel. Resolviert man eine dieser Klauseln mit einer Eingabeklausel, so erhält man eine Klausel die zur Eingabeklausel äquivalent ist. Somit kann man also die leere Klausel nicht herleiten. Allerdings handelt es sich bei der Klausel* $\{\mathsf{p}(X), \mathsf{p}(Y)\}$ *auch nicht um eine Hornklausel.*

**Satz 3.5.10 (Korrektheit und Vollständigkeit der binären SLD–Resolution)** *Sei* $\mathcal{K}$ *eine Menge von Hornklauseln. Dann ist* $\mathcal{K}$ *unerfüllbar gdw.* $\square$ *aus einer negativen Klausel* $N$ *in* $\mathcal{K}$ *durch binäre SLD–Resolution herleitbar ist.*

**Beweis.** Die Korrektheit (d.h., die Richtung "$\Leftarrow$") ist mit Satz 3.4.10 wieder offensichtlich, da jeder binäre SLD–Resolutionsschritt auch ein Resolutionsschritt ist.

Wir zeigen daher jetzt die *Vollständigkeit* (d.h. die Richtung "$\Rightarrow$"). Hierzu gehen wir in zwei Schritten vor. Zunächst zeigen wir, dass jeder Resolutionsschritt mit allgemeiner SLD–Resolution durch eine Folge *unbeschränkter* binärer SLD–Resolutionsschritte ersetzt

werden kann. Hierbei bedeutet *unbeschränkte* SLD-Resolution, dass man beliebige Unifikatoren anstelle von mgu's verwenden darf. Anschließend beweisen wir, dass es zu jedem Unerfüllbarkeitsbeweis mit unbeschränkter binärer SLD–Resolution auch einen Unerfüllbarkeitsbeweis mit binärer SLD–Resolution gibt. Da die allgemeine SLD–Resolution auf Hornklauseln vollständig ist (Satz 3.5.8), folgt dann die Behauptung des Satzes.

Wir zeigen zuerst, dass jeder Resolutionsschritt mit allgemeiner SLD–Resolution durch eine Folge *unbeschränkter* binärer SLD–Resolutionsschritte ersetzt werden kann. Ein allgemeiner SLD–Resolutionsschritt resolviert eine negative Hornklausel $N = \{\neg A_1, \ldots, \neg A_m, \neg A_{m+1}, \ldots, \neg A_p\}$ mit einer definiten Klausel $K = \{B, \neg C_1, \ldots, \neg C_n\}$ unter Verwendung des (o.B.d.A. idempotenten) mgu $\sigma$ der Menge $\{A_1, \ldots, A_m, \nu_1(B)\}$ zum Resolventen $R = \sigma(\{\neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\})$. Hierbei ist $\nu_1$ eine geeignete Variablenumbenennung. Da $\sigma$ mgu von $\{A_1, \ldots, A_m, B\}$ ist, ist $\sigma$ insbesondere auch Unifikator von $A_1$ und $B$ (allerdings nicht unbedingt ihr mgu). Mit unbeschränkter binärer SLD–Resolution kann man aus $N$ und $K$ also den Resolventen

$$R_1 = \sigma(\{\neg A_2, \ldots, \neg A_m, \neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\})$$

herleiten. Nun verwenden wir eine weitere Variablenumbenennung $\nu_2$ und erweitern $\sigma$ so, dass $\sigma \circ \nu_2 = \sigma \circ \nu_1$ ist. Dies ist möglich, da $\nu_2$ o.B.d.A. nur neue Variablen einführt, die nicht im Domain von $\sigma$ waren. Dann ist $\sigma$ Unifikator von $\sigma(A_2)$ und $\nu_2(B)$ und durch erneute unbeschränkte binäre SLD–Resolution kann man aus $R_1$ und $K$ also den Resolventen

$$R_2 = \sigma(\{\neg A_3, \ldots, \neg A_m, \neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\})$$

herleiten (da $\sigma(\sigma(A_i)) = \sigma(A_i)$ und $\sigma(\sigma(\nu_1(C_i))) = \sigma(\nu_2(C_i))$ ist). Insgesamt erhält man also nach $m$ unbeschränkten binären Resolutionsschritten die Klausel

$$R = \sigma(\{\neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\}).$$

Jetzt beweisen wir, dass jeder unbeschränkte binäre SLD-Resolutionsbeweis der leeren Klausel mit $n$ Schritten auch in einen binären SLD-Resolutionsbeweis umgeformt werden kann. Hierzu verwenden wir Induktion über $n$. Im Fall $n = 0$ ist dies trivial. Ansonsten wird im ersten Schritt eine negative Hornklausel $N = \{\neg A_1, \ldots, \neg A_m\}$ mit einer definiten Klausel $K_1 = \{B, \neg C_1, \ldots, \neg C_p\}$ unter Verwendung des (nicht unbedingt allgemeinsten) Unifikators $\sigma_1$ der Menge $\{A_1, \nu_1(B)\}$ zum Resolventen

$$R_1 = \sigma_1(\{\neg A_2, \ldots, \neg A_m, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_p)\})$$

resolviert. Sofern $R_1 = \square$ ist, so hätte man anstelle des Unifikators $\sigma_1$ offensichtlich auch den entsprechenden mgu nehmen können. Ansonsten wird im nächsten Schritt mit einer definiten Klausel $K_2 = \{D, \neg E_1, \ldots, \neg E_q\}$ resolviert. Hierbei sei $\sigma_2$ ein Unifikator von $\sigma_1(A_2)$[2] und $\nu_2(D)$. Dann ergibt sich der Resolvent

$$R_2 = \sigma_2(\sigma_1(\{\neg A_3, \ldots, \neg A_m, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_p)\}) \cup \{\neg\nu_2(E_1), \ldots, \neg\nu_2(E_q)\}).$$

[2]Es könnte auch stattdessen mit $\nu_1(C_1)$ resolviert werden. Der Beweis wäre in diesem Fall analog.

Sei $\theta$ mgu von $\{A_1, \nu_1(B)\}$. Dann existiert eine Substitution $\delta$ mit $\sigma_1 = \delta \circ \theta$. Mit binärer SLD-Resolution hätte man aus $N$ und $K_1$ die Klausel

$$R_1' = \theta(\{\neg A_2, \ldots, \neg A_m, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_p)\})$$

erhalten. Mit einem weiteren unbeschränkten binären SLD-Resolutionsschritt könnten wir $R_1'$ und $K_2$ miteinander resolvieren. Um $\theta(A_2)$ und $\nu_2(D)$ zu unifizieren, kann man den Unifikator $\sigma_2 \circ \delta$ verwenden, denn $\sigma_2(\delta(\theta(A_2))) = \sigma_2(\sigma_1(A_2)) = \sigma_2(\nu_2(D)) = \sigma_2(\delta(\nu_2(D)))$ (da $\delta$'s Domain o.B.d.A. keine Variablen enthält, die durch $\nu_2$ eingeführt werden). So ergibt sich durch die unbeschränkte binäre SLD-Resolution von $R_1'$ und $K_2$ wieder der Resolvent $R_2$. Man kann also aus $R_1'$ durch unbeschränkte binäre SLD-Resolution die leere Klausel in $n-1$ Schritten herleiten. Nach der Induktionshypothese ist dies dann auch durch binäre SLD-Resolution möglich. Da auch der erste Schritt von $N$ zu $R_1'$ ein binärer SLD-Resolutionsschritt war, kann man also den gesamten Beweis mit binärer SLD-Resolution durchführen. □

# Kapitel 4

# Logikprogramme

In diesem Kapitel führen wir nun (reine) Logikprogramme formal ein und definieren in Abschnitt 4.1 ihre Syntax und Semantik. Dies beruht natürlich auf den Grundlagen der Prädikatenlogik und auf dem Resolutionskalkül aus den vorigen Kapiteln. Anschließend zeigen wir in Abschnitt 4.2, dass Logikprogrammierung *universell* (oder "Turing–vollständig") ist, d.h., dass man mit dieser Programmiersprache tatsächlich alle berechenbaren Programme berechnen kann. Schließlich gehen wir in Abschnitt 4.3 auf die Indeterminismen bei der Auswertung von Logikprogrammen ein.

## 4.1 Syntax und Semantik von Logikprogrammen

Die folgende Definition führt Logikprogramme formal ein. Hierbei verwenden wir wieder Hornklauseln und Hornklauselmengen. Im Unterschied zum vorigen Kapitel spielt nun aber die Reihenfolge der Literale in einer Klausel und die Reihenfolge der Klauseln in einer Klauselmenge eine Rolle. Daher betrachten wir von nun an *Folgen* statt Mengen. Wenn wir also von "Klauseln" sprechen, so sind *Folgen* von Literalen gemeint und wenn wir von "Klauselmengen" sprechen, so sind *Folgen* von Klauseln gemeint. Wir behalten hierbei die bisherige Schreibweise bei, d.h., wir schreiben Klauseln und Klauselmengen weiterhin mit Mengenklammern. Die Reihenfolge der Literale bzw. Klauseln ist allerdings jetzt nicht mehr unerheblich und darüber hinaus kann eine Klausel ein Literal auch mehrfach enthalten und eine Klauselmenge kann eine Klausel mehrfach enthalten.

**Definition 4.1.1 (Syntax von Logikprogrammen)** *Eine nicht–leere endliche Menge $\mathcal{P}$ von definiten Hornklauseln über einer Signatur $(\Sigma, \Delta)$ heißt* Logikprogramm *über $(\Sigma, \Delta)$. Die Klauseln aus $\mathcal{P}$ werden auch* Programmklauseln *genannt und man unterscheidet die folgenden Arten von Programmklauseln:*

- Fakten *(oder "Tatsachenklauseln") sind Klauseln der Form $\{B\}$, wobei $B$ eine atomare Formel ist*
- Regeln *(oder "Prozedurklauseln") sind Klauseln der Form $\{B, \neg C_1, \ldots, \neg C_n\}$ mit $n \geq 1$*

*Hierbei sind $B$ und $C_1, \ldots, C_n$ atomare Formeln. Der Aufruf eines Logikprogramms geschieht durch eine*

- Anfrage *(oder "Zielklausel") $G$ der Form* $\{\neg A_1, \ldots, \neg A_k\}$ *mit* $k \geq 1$

Wie bisher repräsentiert eine Klausel die allquantifizierte Disjunktion ihrer Literale und eine Klauselmenge (wie z.B. ein Logikprogramm) repräsentiert die Konjunktion ihrer Klauseln. Beim Aufruf eines Logikprogramms $\mathcal{P}$ mit der Anfrage $G = \{\neg A_1, \ldots, \neg A_k\}$ soll folgendes bewiesen werden:

$$\mathcal{P} \models \exists X_1, \ldots, X_p \; A_1 \wedge \ldots \wedge A_k \tag{4.1}$$

Hierbei sind $X_1, \ldots, X_p$ die Variablen in $G$. Wie erwähnt sind also die Variablen in den Programmklauseln implizit allquantifiziert und die Variablen in den Anfragen sind implizit existenzquantifiziert. Wie in Lemma 3.0.1 gezeigt wurde, ist die obige Folgerbarkeitsbeziehung äquivalent zur Unerfüllbarkeit der Klauselmenge $\mathcal{P} \cup \{G\}$, d.h., zur Unerfüllbarkeit von $\mathcal{P} \cup \{\forall X_1, \ldots, X_p \; \neg A_1 \vee \ldots \vee \neg A_k\}$.

Die Unerfüllbarkeit von $\mathcal{P} \cup \{G\}$ ist nach Satz 3.3.9 (a) äquivalent dazu, dass es eine endliche Teilmenge von Grundinstanzen der Klauseln aus $\mathcal{P} \cup \{G\}$ gibt, die ebenfalls unerfüllbar ist. Diese Menge kann nicht nur aus definiten Klauseln bestehen, d.h., sie enthält also auch mindestens eine Grundinstanz von $G$. Andererseits wissen wir aus der Vollständigkeit der SLD–Resolution für Hornklauselmengen (Satz 3.5.8), dass zur Herleitung der leeren Klausel aus einer unerfüllbaren Menge von Hornklauseln nur eine einzige negative Klausel benötigt wird. Somit bedeutet die Unerfüllbarkeit von $\mathcal{P} \cup \{G\}$ also, dass es eine endliche unerfüllbare Menge von Grundinstanzen der Klauseln aus $\mathcal{P} \cup \{G\}$ gibt, die genau eine Grundinstanz von $G$ enthält. Damit ist (4.1) also äquivalent dazu, dass eine Menge von Grundtermen $t_1, \ldots, t_p$, existiert, so dass

$$\mathcal{P} \cup \{(\neg A_1 \vee \ldots \vee \neg A_k)[X_1/t_1, \ldots, X_p/t_p]\} \text{ unerfüllbar ist bzw. dass}$$
$$\mathcal{P} \models A_1 \wedge \ldots \wedge A_k \, [X_1/t_1, \ldots, X_p/t_p].$$

Bei der Auswertung von Logikprogrammen ist das Ziel, nicht nur die Folgerbarkeitsbeziehung aus (4.1) zu untersuchen, sondern man möchte auch die Grundterme $t_1, \ldots, t_p$ berechnen, die die gültigen "Lösungen" für die Anfrage darstellen. Substitutionen, bei denen die gewünschte Anfrage aus den Programmklauseln folgt, bezeichnet man als *Antwortsubstitution*. Hierbei schränkt man die Substitution auf die Variablen der Anfrage ein. Falls die Antwortsubstitution die Variablen der Anfrage durch Terme mit Variablen ersetzt, so können die verbleibenden Variablen durch beliebige Terme ersetzt werden. Die Antwortsubstitutionen werden im Lauf des SLD–Resolutionsbeweises erzeugt, wenn die leere Klausel $\square$ hergeleitet wird.

**Beispiel 4.1.2** *Wir betrachten eine Teilmenge des Logikprogramms aus Kapitel 1, d.h., eine Teilmenge der Formelmenge aus Bsp. 2.1.7:*

```
mutterVon(renate,susanne).
verheiratet(gerd,renate).
vaterVon(V,K) :- verheiratet(V,F), mutterVon(F,K).
```

*Wenn wir dieses Logikprogramm $\mathcal{P}$ als Klauselmenge schreiben, ergibt sich*

$$\{ \quad \{\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne})\},$$
$$\{\mathsf{verheiratet}(\mathsf{gerd}, \mathsf{renate})\},$$
$$\{\mathsf{vaterVon}(V, F), \neg\mathsf{verheiratet}(V, F), \neg\mathsf{mutterVon}(F, K)\} \quad \}.$$

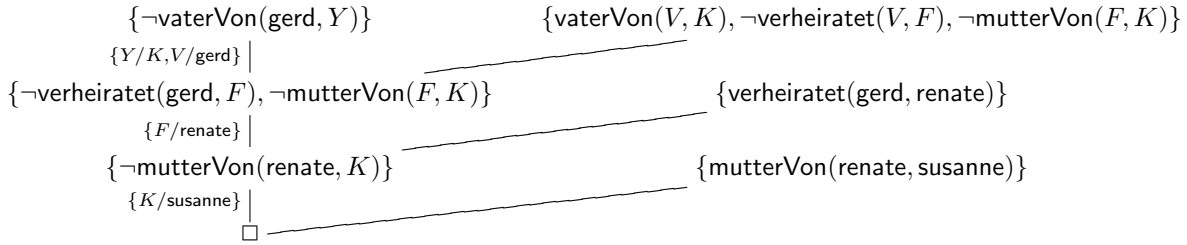
*Wir untersuchen die Anfrage*

```
?- vaterVon(gerd,Y).
```

*Dies bedeutet, dass wir zu der obigen Menge von definiten Hornklauseln noch die negative Hornklausel $G$*

$$\{\neg\mathsf{vaterVon}(\mathsf{gerd}, Y)\}$$

*hinzufügen. Wir erhalten nun den folgenden SLD–Resolutionsbeweis, um die leere Klausel herzuleiten. Hierbei wurde in jedem Resolutionsschritt der mgu angegeben, der auf die negative Elternklausel und die (ggf. variablenumbenannte) Programmklausel angewandt wurde.*




*Die Antwortsubstitution ergibt sich nun, indem man die einzelnen Substitutionen komponiert*

$$\{K/\mathsf{susanne}\} \circ \{F/\mathsf{renate}\} \circ \{Y/K, V/\mathsf{gerd}\} = \{K/\mathsf{susanne}, F/\mathsf{renate}, Y/\mathsf{susanne}, V/\mathsf{gerd}\}$$

*und dann auf die Variablen der Anfrage einschränkt. Da in unserem Beispiel die Anfrage nur die Variable $Y$ enthält, ergibt sich also die Antwortsubstitution $\{Y/\mathsf{susanne}\}$.*

Wir definieren nun die Semantik von Logikprogrammen. Hierbei werden wir drei verschiedene Möglichkeiten vorstellen, die Semantik festzulegen und wir werden beweisen, dass alle drei Möglichkeiten äquivalent sind. Diese Arten werden als *deklarative*, *prozedurale* und *Fixpunkt*–Semantik bezeichnet.

### 4.1.1 Deklarative Semantik der Logikprogrammierung

Die Idee der deklarativen (oder "modelltheoretischen") Semantik ist, das Logikprogramm als statische "Datenbank" aufzufassen und die wahren Aussagen über das Programm mit Hilfe der Folgerbarkeit der Prädikatenlogik zu definieren. Genauer definieren wir jeweils die Semantik eines Programms $\mathcal{P}$ *bezüglich* einer Anfrage $G$. Die deklarative Semantik besteht aus allen Grundinstanzen von $G$, die "wahre Aussagen" über das Programm $\mathcal{P}$ sind.

**Definition 4.1.3 (Deklarative Semantik eines Logikprogramms)** *Sei $\mathcal{P}$ ein Logikprogramm und $G = \{\neg A_1, \ldots, \neg A_k\}$ eine Anfrage. Hierbei sind $A_1, \ldots, A_k$ also atomare Formeln. Dann ist die* deklarative Semantik *von $\mathcal{P}$ bezüglich $G$ definiert als*

$$D[\![\mathcal{P}, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_k),\ \sigma \textit{ ist Grundsubstitution}\}.$$

Jede Grundinstanz $\sigma(A_1 \wedge \ldots \wedge A_k)$ in $D[\![\mathcal{P}, G]\!]$ enthält als "Lösung" die entsprechende Grundsubstitution der Variablen aus $A_1, \ldots, A_k$.

**Beispiel 4.1.4** *Wir betrachten wieder das Logikprogramm $\mathcal{P}$ und die Anfrage $G = \{\neg\mathsf{vaterVon}(\mathsf{gerd}, Y)\}$ aus Bsp. 4.1.2. Die einzige Grundinstanz von $\mathsf{vaterVon}(\mathsf{gerd}, Y)$, die aus $\mathcal{P}$ folgt, ist $\mathsf{vaterVon}(\mathsf{gerd}, \mathsf{susanne})$ (d.h., $\mathcal{P} \models \mathsf{vaterVon}(\mathsf{gerd}, \mathsf{susanne})$). Somit gilt*

$$D[\![\mathcal{P}, G]\!] = \{\mathsf{vaterVon}(\mathsf{gerd}, \mathsf{susanne})\}.$$

*Würde $\mathcal{P}$ noch das zusätzliche Faktum $\mathsf{mutterVon}(\mathsf{renate}, \mathsf{peter})$ enthalten, so ergäbe sich*

$$D[\![\mathcal{P}, G]\!] = \{\mathsf{vaterVon}(\mathsf{gerd}, \mathsf{susanne}), \mathsf{vaterVon}(\mathsf{gerd}, \mathsf{peter})\}.$$

### 4.1.2 Prozedurale Semantik der Logikprogrammierung

Die prozedurale (oder "operationelle") Semantik "operationalisiert" die deklarative Semantik, indem explizit angegeben wird, wie die entsprechenden Folgerungen aus $\mathcal{P}$ berechnet werden. Hierzu wird SLD–Resolution verwendet, wobei jeweils mitprotokolliert wird, welche Substitutionen auf die Variablen angewandt werden. Genauer geben wir einen abstrakten *Interpreter* für Logikprogramme an, der auf sogenannten *Konfigurationen* operiert. Eine Konfiguration ist ein Paar aus einer Anfrage (d.h., einer negativen Klausel) und einer Substitution. Hierbei starten wir mit der Konfiguration $(G, \varnothing)$ aus der ursprünglichen Anfrage und der identischen Substitution $\varnothing$. Das Ziel ist, schließlich eine Endkonfiguration der Gestalt $(\square, \sigma)$ zu erreichen. In diesem Fall ist $\sigma$ (bzw. die Einschränkung von $\sigma$ auf die Variablen der ursprünglichen Anfrage $G$) die gefundene *Antwortsubstitution*. Eine *Berechnung* ist eine Folge von Konfigurationen, wobei bei der Definition des Interpreters festgelegt wird, welche Konfigurationsübergänge erlaubt sind.

Die bei Logikprogrammen verwendete Form der SLD–Resolution unterscheidet sich in drei Punkten von der allgemeinen SLD–Resolution aus Def. 3.5.7.

- Anstatt Variablenumbenennungen jeweils auf beide Elternklauseln anzuwenden, schränken wir uns auf die sogenannte *standardisierte* SLD–Resolution ein. Hier dürfen Variablenumbenennungen nur auf die (definiten) Programmklauseln angewendet werden und nicht auf die jeweils andere (negative) Elternklausel. Eine solche Einschränkung ist natürlich generell ohne Beschränkung der Allgemeinheit möglich.

- Im Resolutionsschritt wird jeweils nur zwischen zwei Literalen resolviert, nicht zwischen beliebig vielen. Wir verwenden also nur *binäre* SLD–Resolution. Wie in Satz 3.5.10 gezeigt wurde, erhält diese Einschränkung aber auf Hornklauselmengen die Vollständigkeit.

- Schließlich betrachten wir Klauseln nicht mehr als Mengen, sondern als *Folgen* von Literalen. Dies bedeutet, dass in einer solchen Folge ein Literal also auch mehrfach auftreten kann. Beispielsweise ergibt dann die Resolution der Klauseln $\{\neg\mathsf{p}, \neg\mathsf{p}\}$ und $\{\mathsf{p}\}$ den Resolventen $\{\neg\mathsf{p}\}$. Hierbei gilt $\mathsf{p} \in \Delta_0$. Ähnlich wie im Vollständigkeitsbeweis der binären SLD–Resolution (Satz 3.5.10) folgt auch sofort die Vollständigkeit der binären SLD–Resolution auf diesen Folgen. Der Grund ist, dass man offensichtlich jeden bisherigen Resolutionsschritt auf Mengen durch eine Folge von Resolutionsschritten auf Folgen ersetzen kann.

**Definition 4.1.5 (Prozedurale Semantik eines Logikprogramms)** *Sei $\mathcal{P}$ ein Logikprogramm.*

- *Eine* Konfiguration *ist ein Paar $(G, \sigma)$, wobei $G$ eine Anfrage oder die leere Klausel $\square$ ist und wobei $\sigma$ eine Substitution ist.*
- *Es gibt einen* Rechenschritt $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ *gdw.*
  - $G_1 = \{\neg A_1, \ldots, \neg A_k\}$ *mit* $k \geq 1$
  - *es existiert eine Programmklausel $K \in \mathcal{P}$ und eine Variablenumbenennung $\nu$ mit $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ und $n \geq 0$, so dass*
    - *$\nu(K)$ keine gemeinsamen Variablen mit $G_1$ oder $RANGE(\sigma_1)$ hat und*
    - *es ein $1 \leq i \leq k$ gibt, so dass $A_i$ und $B$ mit einem mgu $\sigma$ unifizierbar sind*
  - $G_2 = \sigma(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\})$
  - $\sigma_2 = \sigma \circ \sigma_1$
- *Eine* Berechnung *von $\mathcal{P}$ bei Eingabe von $G = \{\neg A_1, \ldots, \neg A_k\}$ ist eine (endliche oder unendliche) Folge von Konfigurationen der Form*

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \ldots$$

- *Eine mit $(\square, \sigma)$ terminierende Berechnung, die mit $(G, \varnothing)$ startet (wobei $G = \{\neg A_1, \ldots, \neg A_k\}$), heißt* erfolgreich *mit dem Rechenergebnis $\sigma(A_1 \wedge \ldots \wedge A_k)$. Die berechnete* Antwortsubstitution *ist $\sigma$ eingeschränkt auf die Variablen aus $G$.*

*Damit ist die* prozedurale Semantik *von $\mathcal{P}$ bezüglich $G$ definiert als*

$$P[\![\mathcal{P}, G]\!] = \{\sigma'(A_1 \wedge \ldots \wedge A_k) \mid (G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma), \\ \sigma'(A_1 \wedge \ldots \wedge A_k) \text{ ist Grundinstanz von } \sigma(A_1 \wedge \ldots \wedge A_k)\}.$$

*Hierbei steht "$\vdash_{\mathcal{P}}^{+}$" für die transitive Hülle von "$\vdash_{\mathcal{P}}$" (d.h., es gilt $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma)$ gdw. $(G, \varnothing) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma)$). Analog dazu definieren wir auch für alle $l \in \mathbb{N}$ die Relation $\vdash_{\mathcal{P}}^{l}$ als $(G, \sigma) \vdash_{\mathcal{P}}^{l} (G_l, \sigma_l)$ gdw. es $G_i$ und $\sigma_i$ gibt mit $(G, \sigma) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (G_l, \sigma_l)$.*

**Beispiel 4.1.6** *Die prozedurale Semantik entspricht gerade dem Vorgehen in Bsp. 4.1.2. Hier gilt:*

$$\begin{array}{ll} & (\{\neg \mathsf{vaterVon}(\mathsf{gerd}, Y)\}, \varnothing) \\ \vdash_{\mathcal{P}} & (\{\neg \mathsf{verheiratet}(\mathsf{gerd}, F), \neg \mathsf{mutterVon}(F, K)\}, \{Y/K, V/\mathsf{gerd}\}) \\ \vdash_{\mathcal{P}} & (\{\neg \mathsf{mutterVon}(\mathsf{renate}, K)\}, \{F/\mathsf{renate}, Y/K, V/\mathsf{gerd}\}) \\ \vdash_{\mathcal{P}} & (\square, \{K/\mathsf{susanne}, F/\mathsf{renate}, Y/\mathsf{susanne}, V/\mathsf{gerd}\}) \end{array}$$

*und die Antwortsubstitution ist $\{Y/\mathsf{susanne}\}$. Wir erhalten*

$$P[\![\mathcal{P}, G]\!] = \{\mathsf{vaterVon}(\mathsf{gerd}, \mathsf{susanne})\}.$$

Das nächste Beispiel zeigt, dass es bei den Rechenschritten der prozeduralen Semantik noch zwei Indeterminismen gibt:

- Zum einen muss man die Programmklausel $K$ auswählen, mit der resolviert werden soll.
- Zum anderen muss man jeweils das nächste Literal $A_i$ aus der momentanen Anfrage auswählen, das zur Resolution verwendet werden soll.

**Beispiel 4.1.7** *Wir betrachten das folgende Logikprogramm* $\mathcal{P}$

$$\begin{array}{ll}\{ & \{\mathsf{p}(X,Z), \neg\mathsf{q}(X,Y), \neg\mathsf{p}(Y,Z)\},\\ & \{\mathsf{p}(U,U)\},\\ & \{\mathsf{q}(\mathsf{a},\mathsf{b})\}\ \}\end{array}$$

*mit* $\mathsf{p},\mathsf{q} \in \Delta_2$ *und* $\mathsf{a},\mathsf{b} \in \Sigma_0$. *Die zu untersuchende Anfrage ist*

$$G = \{\neg\mathsf{p}(V,\mathsf{b})\}.$$

*Die folgende Berechnung ist nicht erfolgreich (aber endlich und nicht weiter fortsetzbar). Hierbei ist jeweils das Literal* $A_i$ *unterstrichen, das zur Resolution verwendet wurde:*

$$\begin{array}{rl} & (\{\underline{\neg\mathsf{p}(V,\mathsf{b})}\}, \varnothing)\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{q}(V,Y)}, \neg\mathsf{p}(Y,\mathsf{b})\}, \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{p}(\mathsf{b},\mathsf{b})}\}, \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{q}(\mathsf{b},Y')}, \underline{\neg\mathsf{p}(Y',\mathsf{b})}\}, \{X'/\mathsf{b}, Z'/\mathsf{b}\} \circ \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\neg\mathsf{q}(\mathsf{b},\mathsf{b})\}, \underline{\{U/\mathsf{b}, Y'/\mathsf{b}\}} \circ \{X'/\mathsf{b}, Z'/\mathsf{b}\} \circ \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\}) \end{array}$$

*Eine erfolgreiche Berechnung wäre hingegen die folgende, die die gleichen drei ersten Schritte hat, aber dann mit der zweiten anstelle der ersten Programmklausel resolviert:*

$$\begin{array}{rl} & (\{\underline{\neg\mathsf{p}(V,\mathsf{b})}\}, \varnothing)\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{q}(V,Y)}, \neg\mathsf{p}(Y,\mathsf{b})\}, \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{p}(\mathsf{b},\mathsf{b})}\}, \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\square, \underbrace{\{U/\mathsf{b}\} \circ \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\}}_{\{U/\mathsf{b},\ V/\mathsf{a},\ Y/\mathsf{b},\ X/\mathsf{a},\ Z/\mathsf{b}\}}) \end{array}$$

*Die Antwortsubstitution ist also* $\{V/\mathsf{a}\}$ *und somit gilt* $\mathsf{p}(\mathsf{a},\mathsf{b}) \in P[\![\mathcal{P}, G]\!]$.

*In diesem Beispiel gibt es aber noch eine weitere erfolgreiche Berechnung, indem man bereits im ersten Schritt gleich mit der zweiten Programmklausel resolviert:*

$$\begin{array}{rl} & (\{\underline{\neg\mathsf{p}(V,\mathsf{b})}\}, \varnothing)\\ \vdash_{\mathcal{P}} & (\square, \{U/\mathsf{b}, V/\mathsf{b}\}) \end{array}$$

*Die Antwortsubstitution ist nun* $\{V/\mathsf{b}\}$ *und es gilt daher auch* $\mathsf{p}(\mathsf{b},\mathsf{b}) \in P[\![\mathcal{P}, G]\!]$.

Der folgende Satz von Clark zeigt, dass die deklarative und die prozedurale Semantik äquivalent sind. Dies bedeutet insbesondere, dass die Einschränkungen der SLD–Resolution bei der Logikprogrammierung die Vollständigkeit nicht zerstören.

**Satz 4.1.8 (Äquivalenz der deklarativen und der prozeduralen Semantik)** *Sei $\mathcal{P}$ ein Logikprogramm und $G = \{\neg A_1, \ldots, \neg A_k\}$ eine Anfrage. Dann gilt $D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$.*

**Beweis.** Wir zeigen zuerst die Korrektheit der prozeduralen Semantik bezüglich der deklarativen Semantik, d.h. $P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$. Sei $\sigma'(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, G]\!]$. Dann gibt es eine erfolgreiche Berechnung der Form

$$(G, \varnothing) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma),$$

wobei $\sigma'(A_1 \wedge \ldots \wedge A_k)$ eine Grundinstanz von $\sigma(A_1 \wedge \ldots \wedge A_k)$ ist. Zu zeigen ist, dass dann $\mathcal{P} \models \sigma'(A_1 \wedge \ldots \wedge A_k)$ gilt. Wir verwenden Induktion über die Länge $l$ der Berechnung. Genauer ist $l$ die Anzahl der $\vdash_{\mathcal{P}}$-Schritte in der Berechnung. Die Berechnung hat also die Form

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1)$$

mit $\sigma = \delta_l \circ \ldots \circ \delta_1$. Es existiert also ein $A_i$, ein $K \in \mathcal{P}$ und eine Variablenumbenennung $\nu$ mit $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ und $n \geq 0$, so dass $G$ und $\nu(K)$ keine gemeinsamen Variablen haben, so dass $\delta_1$ der mgu von $A_i$ und $B$ ist und so dass

$$G_1 = \delta_1(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

Im Induktionsanfang ist $l = 1$. Dann gilt $G_1 = \square$ und somit $i = k = 1$, $n = 0$, $\nu(K) = \{B\}$ (d.h., die Programmklausel $K$ ist ein Faktum) und $\sigma = \delta_1$. Da alle Grundinstanzen von $B$ aus $\mathcal{P}$ folgen und da $\delta_1(B) = \delta_1(A_1) = \sigma(A_1)$ gilt, folgen also auch alle Grundinstanzen von $\sigma(A_1)$ aus $\mathcal{P}$. Somit gilt insbesondere $\mathcal{P} \models \sigma'(A_1)$.

Im Induktionsschritt ist $l > 1$. Dann ist offensichtlich auch

$$(G_1, \varnothing) \vdash_{\mathcal{P}} (G_2, \delta_2) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_2)$$

eine Berechnung. Nach der Induktionshypothese ist jede Grundinstanz von

$$\delta_l(\ldots \delta_2(\delta_1(A_1 \wedge \ldots \wedge A_{i-1} \wedge C_1 \wedge \ldots \wedge C_n \wedge A_{i+1} \wedge \ldots \wedge A_k)) \ldots)$$

Folgerung von $\mathcal{P}$. Da alle Grundinstanzen von $C_1 \wedge \ldots \wedge C_n \rightarrow B$ auch Folgerungen von $\mathcal{P}$ sind, ist also auch jede Grundinstanz von

$$\delta_l(\ldots \delta_1(A_1 \wedge \ldots \wedge A_{i-1} \wedge B \wedge A_{i+1} \wedge \ldots \wedge A_k) \ldots)$$

Folgerung von $\mathcal{P}$. Da $\delta_1(B) = \delta_1(A_i)$ gilt, folgen also auch alle Grundinstanzen von

$$\delta_l(\ldots \delta_1(A_1 \wedge \ldots \wedge A_k) \ldots)$$

aus $\mathcal{P}$. Da $\sigma = \delta_l \circ \ldots \circ \delta_1$ ist, gilt also insbesondere $\mathcal{P} \models \sigma'(A_1 \wedge \ldots \wedge A_k)$.

Nun zeigen wir die Vollständigkeit der prozeduralen Semantik bezüglich der deklarativen Semantik, d.h. $D[\![\mathcal{P}, G]\!] \subseteq P[\![\mathcal{P}, G]\!]$. Sei $\sigma(A_1 \wedge \ldots \wedge A_k) \in D[\![\mathcal{P}, G]\!]$. Dann gilt $\mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_k)$, d.h., $\mathcal{P} \cup \{\sigma(\{\neg A_1, \ldots, \neg A_k\})\}$ ist nach Lemma 3.0.1 unerfüllbar.

Aufgrund der Vollständigkeit der binären SLD–Resolution (Satz 3.5.10) existiert eine Herleitung der leeren Klausel $\square$ aus der negativen Klausel $\sigma(\{\neg A_1, \ldots, \neg A_k\})$ in $\mathcal{P} \cup$

$\{\sigma(\{\neg A_1, \ldots, \neg A_k\})\}$ durch binäre SLD–Resolution. Somit existiert also eine erfolgreiche Berechnung

$$(\sigma(\{\neg A_1, \ldots, \neg A_k\}), \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \delta_l \circ \ldots \circ \delta_1).$$

Hierbei ist $l$ die Länge der Berechnung und $\delta_i$ ist der im $i$-ten Resolutionsschritt verwendete mgu (für alle $1 \leq i \leq l$).

Die obige Berechnung startet mit einer Anfrage $\sigma(\{\neg A_1, \ldots, \neg A_k\})$ ohne Variablen, da $\sigma$ eine Grundsubstitution ist. Wir zeigen nun, dass sich diese Berechnung zu einer Berechnung der Form

$$(\{\neg A_1, \ldots, \neg A_k\}, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \delta'_l \circ \ldots \circ \delta'_1)$$

"liften" lässt, so dass $\sigma(A_1 \wedge \ldots \wedge A_k)$ Grundinstanz von $\delta'_l(\ldots \delta'_1(A_1 \wedge \ldots \wedge A_k)\ldots)$ ist. Damit folgt dann wie gewünscht $\sigma(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, G]\!]$.

Um die obige Behauptung zu zeigen, beweisen wir eine etwas allgemeinere Aussage für beliebige Substitutionen $\sigma$, wobei $DOM(\sigma)$ nur Variablen aus $G$ enthält:[1]

$$\begin{array}{l}\text{Falls } (\sigma(G), \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \delta_l \circ \ldots \circ \delta_1), \text{ dann existiert eine Berechnung } (G, \varnothing) \vdash_{\mathcal{P}}^{+} \\ (\square, \delta'_l \circ \ldots \circ \delta'_1) \text{ und eine Substitution } \sigma' \text{ mit } \delta_l \circ \ldots \circ \delta_1 \circ \sigma = \sigma' \circ \delta'_l \circ \ldots \circ \delta'_1.\end{array} \tag{4.2}$$

Daraus folgt die obige Behauptung, da dort $\sigma$ eine Grundsubstitution auf allen Variablen von $G$ ist und somit $\delta_l(\ldots \delta_1(\sigma(G))\ldots) = \sigma(G)$ gilt.

Wir verwenden Induktion über die Länge $l$ der Berechnung. Im ersten Schritt $(\sigma(G), \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1)$ existiert also ein $A_i \in G$, das zur Resolution mit einer Klausel $K \in \mathcal{P}$ verwendet wird. Wir benutzen hier zur Resolution o.B.d.A. eine Variablenumbenennung $\nu$, so dass $\nu(K)$ keine gemeinsamen Variablen mit $G$ hat. Dann gilt $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ mit $n \geq 0$, so dass $\delta_1$ der mgu von $\sigma(A_i)$ und $B$ ist und es gilt

$$G_1 = \delta_1(\{\neg\sigma(A_1), \ldots, \neg\sigma(A_{i-1}), \neg C_1, \ldots, \neg C_n, \neg\sigma(A_{i+1}), \ldots, \neg\sigma(A_k)\}).$$

Wir zeigen nun, dass man einen entsprechenden Resolutionsschritt auch mit $G$ und $\nu(K)$ statt mit $\sigma(G)$ und $\nu(K)$ durchführen kann. Anstelle des Literals $\sigma(A_i) \in \sigma(G)$ wird nun das Literal $A_i \in G$ zur Resolution verwendet. Da $\sigma$ nur Variablen aus $A_i$ und nicht aus $B$ instantiiert, ist $\delta_1 \circ \sigma$ ein Unifikator von $A_i$ und $B$. Daher haben $A_i$ und $B$ auch einen mgu $\delta'_1$. Es existiert also eine Substitution $\tau$ mit

$$\delta_1 \circ \sigma = \tau \circ \delta'_1. \tag{4.3}$$

Der Resolutionsschritt mit $G$ und $\nu(K)$ ergibt

$$G'_1 = \delta'_1(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

Es gilt also $(G, \varnothing) \vdash_{\mathcal{P}} (G'_1, \delta'_1)$. Man erkennt, dass der Resolvent $G_1$, der aus $\sigma(G)$ und $\nu(K)$ entsteht, tatsächlich eine Instanz des Resolventen $G'_1$ von $G$ und $\nu(K)$ ist, denn es gilt $\tau(G'_1) = G_1$.

[1] Diese Aussage ist keine direkte Folgerung des Lifting–Lemmas 3.4.8, da hier die jeweiligen Elternklauseln aus dem Programm Variablen enthalten und da die Aussage auch etwas über die verwendeten Substitutionen aussagt, was beim Lifting–Lemma nicht der Fall ist.

Im Induktionsanfang ist $l = 1$. Dann gilt $G_1 = G'_1 = \square$. Wir haben $(G, \varnothing) \vdash_{\mathcal{P}} (\square, \delta'_1)$ und aus (4.3) folgt wie gewünscht $\delta_1 \circ \sigma = \sigma' \circ \delta'_1$, wenn man $\sigma' := \tau$ setzt. Somit ist (4.2) bewiesen.

Im Induktionsschluss ist $l > 1$. Da $(\sigma(G), \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash^+_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1)$ eine Berechnung der Länge $l$ ist, ist $(G_1, \varnothing) \vdash^+_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_2)$ eine Berechnung der Länge $l-1$. Wir hatten bereits gezeigt, dass sich der erste Berechnungsschritt $(\sigma(G), \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1)$ zu $(G, \varnothing) \vdash_{\mathcal{P}} (G'_1, \delta'_1)$ "liften" lässt. Da $G_1 = \tau(G'_1)$ ist, gibt es nach der Induktionshypothese also auch eine Berechnung $(G'_1, \varnothing) \vdash^+_{\mathcal{P}} (\square, \delta'_l \circ \ldots \circ \delta'_2)$ und eine Substitution $\tau'$ mit

$$\delta_l \circ \ldots \circ \delta_2 \circ \tau = \tau' \circ \delta'_l \circ \ldots \circ \delta'_2. \tag{4.4}$$

Damit folgt der erste Teil der Aussage (4.2): $(G, \varnothing) \vdash_{\mathcal{P}} (G'_1, \delta'_1) \vdash^+_{\mathcal{P}} (\square, \delta'_l \circ \ldots \circ \delta'_2 \circ \delta'_1)$. Für den zweiten Teil der Aussage (4.2) gilt:

$$\begin{array}{lll} & \delta_l \circ \ldots \circ \delta_2 \circ \delta_1 \circ \sigma & \\ = & \delta_l \circ \ldots \circ \delta_2 \circ \tau \circ \delta'_1 & \text{nach (4.3) (beim 1–Schritt–Liften entsteht aus } \sigma \text{ also } \tau) \\ = & \tau' \circ \delta'_l \circ \ldots \circ \delta'_2 \circ \delta'_1 & \text{nach (4.4) (beim } (l-1)\text{–Schritt–Liften entsteht aus } \tau \text{ also } \tau'). \end{array}$$

Somit folgt auch der zweite Teil der Aussage (4.2), wenn man $\sigma' := \tau'$ setzt. □

### 4.1.3 Fixpunkt–Semantik der Logikprogrammierung

Bei der folgenden dritten Art der Semantik–Definition wird die Semantik eines Logikprogramms als Fixpunkt einer Transformation $\underline{\text{trans}}_{\mathcal{P}}$ definiert. Diese Transformation leitet nach und nach alle wahren Aussagen über ein Programm her. Im Unterschied zur deklarativen Semantik ist die Semantik hier nicht modelltheoretisch definiert, sondern es wird (ähnlich wie bei der prozeduralen Semantik) die Einschränkung auf Hornklauseln ausgenutzt, um durch resolutionsähnliche Schritte wahre Aussagen herzuleiten. Im Unterschied zur prozeduralen Semantik wird bei der Herleitung aber nicht von der Anfrage ausgegangen, sondern man geht nur vom Programm aus und leitet alle wahren Aussagen (ohne Variablen) her.

Zu einem Logikprogramm $\mathcal{P}$ ist $\underline{\text{trans}}_{\mathcal{P}}$ eine Funktion, die Mengen von Aussagen in Mengen von Aussagen überführt. Hierbei enthält $\underline{\text{trans}}_{\mathcal{P}}(M)$ zusätzlich zu $M$ alle weiteren Aussagen, die aus $M$ durch Anwendung einer Regel aus $\mathcal{P}$ in einem Schritt herleitbar sind. Im Folgenden bezeichnet $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$ die Menge aller Mengen von atomaren Formeln ohne Variablen.

**Definition 4.1.9 (Die Transformation $\underline{\text{trans}}_{\mathcal{P}}$)** *Sei $\mathcal{P}$ ein Logikprogramm über einer Signatur $(\Sigma, \Delta)$. Die dazugehörige Transformation $\underline{\text{trans}}_{\mathcal{P}}$ ist eine Funktion*

$$\underline{\text{trans}}_{\mathcal{P}} : Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing)) \to Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$$

*mit*

$$\underline{\text{trans}}_{\mathcal{P}}(M) = M \cup \{A' \mid \{A', \neg B'_1, \ldots, \neg B'_n\} \text{ ist Grundinstanz einer Klausel } \{A, \neg B_1, \ldots, \neg B_n\} \in \mathcal{P} \text{ und } B'_1, \ldots, B'_n \in M\}$$

Die Idee zur Definition der Semantik besteht nun darin, zuerst von der leeren Menge von Aussagen $\varnothing$ auszugehen. Dann beschreibt $\varnothing$ alle Aussagen, die in null Schritten aus $\mathcal{P}$ herleitbar sind und $\underline{\text{trans}}_{\mathcal{P}}(\varnothing)$ sind alle Aussagen (ohne Variablen), die mit höchstens einer Klausel aus $\mathcal{P}$ herleitbar sind. Es handelt sich also um alle Grundinstanzen der Fakten von $\mathcal{P}$. Analog dazu sind $\underline{\text{trans}}_{\mathcal{P}}(\underline{\text{trans}}_{\mathcal{P}}(\varnothing)) = \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing)$ alle Aussagen, die in höchstens zwei Schritten aus $\mathcal{P}$ herleitbar sind, etc. Die Menge

$$M_{\mathcal{P}} = \varnothing \cup \underline{\text{trans}}_{\mathcal{P}}(\varnothing) \cup \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing) \cup \underline{\text{trans}}^3_{\mathcal{P}}(\varnothing) \cup \ldots = \bigcup_{i \in \mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$$

enthält also alle Aussagen ohne Variablen, die aus $\mathcal{P}$ herleitbar sind. Man kann die Semantik von $\mathcal{P}$ bezüglich einer Anfrage $G$ dadurch definieren, dass man alle Grundinstanzen von $G$ betrachtet, die in $\bigcup_{i \in \mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$ enthalten sind.

**Beispiel 4.1.10** *Wir betrachten wieder das Logikprogramm $\mathcal{P}$ aus Bsp. 4.1.2.*

$$\begin{array}{ll} \{ & \{\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne})\}, \\ & \{\mathsf{verheiratet}(\mathsf{gerd}, \mathsf{renate})\}, \\ & \{\mathsf{vaterVon}(V, K), \neg\mathsf{verheiratet}(V, F), \neg\mathsf{mutterVon}(F, K)\} \ \}. \end{array}$$

*Es ergibt sich*

$$\begin{array}{lcl} \underline{\text{trans}}_{\mathcal{P}}(\varnothing) & = & \{\mathsf{mutterVon}(\mathsf{renate}, \mathsf{susanne}), \mathsf{verheiratet}(\mathsf{gerd}, \mathsf{renate})\} \\ \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing) & = & \underline{\text{trans}}_{\mathcal{P}}(\varnothing) \cup \{\mathsf{vaterVon}(\mathsf{gerd}, \mathsf{susanne})\} \end{array}$$

*und $\underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) = \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing)$ für alle $i \geq 2$. In diesem Beispiel ist $M_{\mathcal{P}}$ also nach nur 2 Iterationsschritten erreicht.*

**Beispiel 4.1.11** *Im Allgemeinen kann die Iteration zur Berechnung von $M_{\mathcal{P}}$ aber unendlich sein. Hierzu betrachten wir das folgende Logikprogramm:*

```
p(a).
p(f(X)) :- p(X).
```

*Es entspricht der folgenden Klauselmenge:*

$$\begin{array}{ll} \{ & \{\mathsf{p}(\mathsf{a})\}, \\ & \{\mathsf{p}(\mathsf{f}(X)), \neg\mathsf{p}(X)\} \ \} \end{array}$$

*Nun gilt*

$$\begin{array}{lcl} \underline{\text{trans}}_{\mathcal{P}}(\varnothing) & = & \{\mathsf{p}(\mathsf{a})\}, \\ \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing) & = & \{\mathsf{p}(\mathsf{a}), \mathsf{p}(\mathsf{f}(\mathsf{a}))\}, \\ \underline{\text{trans}}^3_{\mathcal{P}}(\varnothing) & = & \{\mathsf{p}(\mathsf{a}), \mathsf{p}(\mathsf{f}(\mathsf{a})), \mathsf{p}(\mathsf{f}^2(\mathsf{a}))\}, \textit{etc.} \end{array}$$

*Somit ergibt sich $\underline{\text{trans}}^{i+1}_{\mathcal{P}}(\varnothing) = \{\mathsf{p}(\mathsf{a}), \mathsf{p}(\mathsf{f}(\mathsf{a})), \ldots, \mathsf{p}(\mathsf{f}^i(\mathsf{a}))\}$ und $M_{\mathcal{P}} = \bigcup_{i \in \mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) = \{\mathsf{p}(\mathsf{f}^i(\mathsf{a})) \mid i \in \mathbb{N}\}$.*

Die obige Konstruktion berechnet einen *Fixpunkt* der Transformation $\underline{\text{trans}}_{\mathcal{P}}$, denn es gilt $\underline{\text{trans}}_{\mathcal{P}}(M_{\mathcal{P}}) = M_{\mathcal{P}}$. In der Tat handelt es sich hierbei sogar um einen speziellen Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$. $M_{\mathcal{P}}$ ist nämlich der *kleinste* Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$, wenn man Mengen bezüglich der Teilmengenrelation $\subseteq$ vergleicht. Dies bedeutet, dass $M_{\mathcal{P}}$ nur die Aussagen enthält, die wirklich unbedingt aufgenommen werden müssen, wenn man alle aus $\mathcal{P}$ herleitbaren Aussagen bekommen will.

Wir zeigen nun, dass die oben berechnete Menge $M_{\mathcal{P}}$ wirklich stets der kleinste Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$ ist. Dies beweist auch, dass $\underline{\text{trans}}_{\mathcal{P}}$ stets einen (eindeutigen) kleinsten Fixpunkt hat.

Wie üblich bezeichnet man jede transitive und antisymmetrische Relation als *Ordnung*. Die Relation $\subseteq$ auf $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$ ist offensichtlich eine reflexive Ordnung, d.h., es gilt für alle $M_1, M_2, M_3 \subseteq \mathcal{A}t(\Sigma, \Delta, \varnothing)$:

- $M_1 \subseteq M_1$ (Reflexivität)
- Aus $M_1 \subseteq M_2$ und $M_2 \subseteq M_3$ folgt $M_1 \subseteq M_3$ (Transitivität)
- Aus $M_1 \subseteq M_2$ und $M_2 \subseteq M_1$ folgt $M_1 = M_2$. (Antisymmetrie)

Wenn man die Folge von Mengen

$$\varnothing,\ \underline{\text{trans}}_{\mathcal{P}}(\varnothing),\ \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing),\ \underline{\text{trans}}^3_{\mathcal{P}}(\varnothing), \ldots$$

betrachtet, sieht man, dass diese Mengen offensichtlich immer "größer" bezüglich der Ordnung $\subseteq$ werden. Solche Folgen bezeichnet man als *Ketten*. Die gesuchte Menge $M_{\mathcal{P}}$ ist dann gerade der "Grenzwert" dieser Kette, d.h., es ist ihre *kleinste obere Schranke*. Auf Englisch wird dies als "*least upper bound*" oder "*lub*" bezeichnet. Eine reflexive Ordnung ist *vollständig*, falls sie ein kleinstes Element hat und falls jede Kette eine kleinste obere Schranke besitzt. Solch eine Ordnung wird auch als "*complete partial order*" oder "*cpo*" bezeichnet. Das folgende Lemma zeigt, dass $\subseteq$ auf $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$ in der Tat vollständig ist. Zu jedem Programm $\mathcal{P}$ existiert also die kleinste obere Schranke $M_{\mathcal{P}}$ der obigen Kette.[2]

**Lemma 4.1.12 (Vollständigkeit von $\subseteq$)** *Die Relation $\subseteq$ ist auf $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$ vollständig: Das kleinste Element von $\mathcal{A}t(\Sigma, \Delta, \varnothing)$ bezüglich $\subseteq$ ist die leere Menge $\varnothing$ und jede Kette*

$$M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$$

*mit $M_i \subseteq \mathcal{A}t(\Sigma, \Delta, \varnothing)$ hat die kleinste obere Schranke $M := \bigcup_{i \in \mathbb{N}} M_i$. Es gilt also $M_i \subseteq M$ für alle $M_i$ und für jede weitere obere Schranke $M'$ gilt $M \subseteq M'$.*

**Beweis.** Dass $\varnothing$ das kleinste Element ist, ist offensichtlich, da $\varnothing \subseteq N$ für alle $N \subseteq \mathcal{A}t(\Sigma, \Delta, \varnothing)$. Ebenso ist auch offensichtlich, dass $M := \bigcup_{i \in \mathbb{N}} M_i$ obere Schranke der Kette ist, da trivialerweise $M_i \subseteq M$ gilt. Sei nun $M'$ eine weitere obere Schranke, d.h., $M_i \subseteq M'$ für alle $M_i$. Dann gilt offensichtlich auch $M = \bigcup_{i \in \mathbb{N}} M_i \subseteq M'$. □

Die Funktion $\underline{\text{trans}}_{\mathcal{P}}$ besitzt zwei wichtige Eigenschaften, die entscheidend dafür sind, dass sie tatsächlich immer einen kleinsten Fixpunkt besitzt. Sie ist *monoton* und *stetig*.

[2] Formale allgemeine Definitionen für diese Begriffe sowie für die Begriffe "Monotonie" und "Stetigkeit" finden sich z.B. in [Gie14].

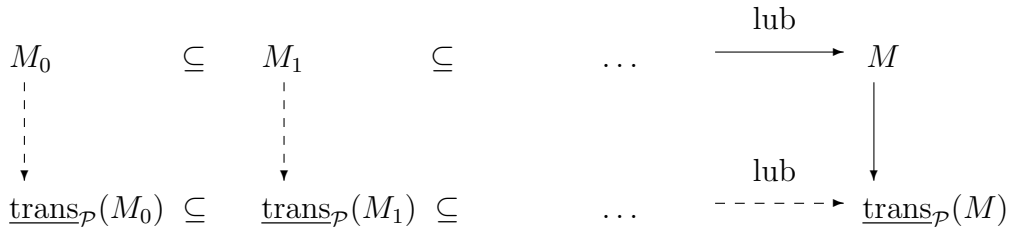



Abbildung 4.1: Stetigkeit von $\underline{\text{trans}}_{\mathcal{P}}$

Monotonie bedeutet in diesem Zusammenhang, dass sich aus größeren Mengen von Aussagen auch mehr Aussagen herleiten lassen. Stetigkeit bedeutet, dass $\underline{\text{trans}}_{\mathcal{P}}$ den Grenzwert jeder Kette auf das gleiche Ergebnis abbildet, das man erhalten würde, wenn man das Ergebnis für jedes individuelle Element der Kette berechnen würde und dann den Grenzwert dieser Ergebnisse bildet. Dies wird in Abb. 4.1 verdeutlicht. Wenn man zunächst die Kette $M_0 \subseteq M_1 \subseteq \ldots$ betrachtet, ihren Grenzwert $M = \bigcup_{i\in\mathbb{N}} M_i$ bildet und dann $\underline{\text{trans}}_{\mathcal{P}}$ darauf anwendet (durchgezogene Pfeile), so muss sich dasselbe ergeben, wie wenn man erst $\underline{\text{trans}}_{\mathcal{P}}$ auf die einzelnen Elemente der Kette anwendet und dann den Grenzwert bildet (gestrichelte Pfeile).

**Lemma 4.1.13 (Monotonie und Stetigkeit)**

*(a) Die Funktion* $\underline{\text{trans}}_{\mathcal{P}}$ *ist* monoton*, d.h., falls* $M_1 \subseteq M_2$ *ist, so gilt auch* $\underline{\text{trans}}_{\mathcal{P}}(M_1) \subseteq \underline{\text{trans}}_{\mathcal{P}}(M_2)$.

*(b) Die Funktion* $\underline{\text{trans}}_{\mathcal{P}}$ *ist* stetig*, d.h., für jede Kette*

$$M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$$

*gilt* $\underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) = \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$.

**Beweis.** Teil (a) ist offensichtlich. Wir zeigen nun Teil (b). Hierbei folgt $\underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) \supseteq \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$ aus der Monotonie. Der Grund ist, dass wegen der Monotonie $\underline{\text{trans}}_{\mathcal{P}}(M_i) \subseteq \underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i)$ für alle $i$ gilt.

Um auch $\underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) \subseteq \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$ zu zeigen, sei $A' \in \underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i)$. Dann ist $\{A', \neg B'_1, \ldots, \neg B'_n\}$ Grundinstanz einer Klausel $\{A, \neg B_1, \ldots, \neg B_n\} \in \mathcal{P}$ und $B'_1, \ldots, B'_n \in \bigcup_{i\in\mathbb{N}} M_i$. Da $M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$ eine Kette ist, gibt es daher ein $j \in \mathbb{N}$ mit $B'_1, \ldots, B'_n \in M_j$. Damit folgt auch $A' \in \underline{\text{trans}}_{\mathcal{P}}(M_j) \subseteq \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$. □

Nun können wir zeigen, dass $\underline{\text{trans}}_{\mathcal{P}}$ in der Tat immer einen kleinsten Fixpunkt ("*least fixpoint*" bzw. "*lfp*") hat und dass dieser gerade der kleinsten oberen Schranke der Kette $\varnothing$, $\underline{\text{trans}}_{\mathcal{P}}(\varnothing)$, $\underline{\text{trans}}^2_{\mathcal{P}}(\varnothing), \ldots$ entspricht. Der folgende Satz entspricht dem allgemeinen Fixpunktsatz von Tarski bzw. Kleene [Kle52]: Jede stetige Funktion bezüglich einer vollständigen Ordnung besitzt einen kleinsten Fixpunkt. Eine allgemeine Formulierung dieses Satzes findet sich z.B. in [Gie14].

**Satz 4.1.14 (Fixpunktsatz)** *Für jedes Logikprogramm* $\mathcal{P}$ *besitzt* $\underline{\text{trans}}_{\mathcal{P}}$ *einen kleinsten Fixpunkt* $\text{lfp}(\underline{\text{trans}}_{\mathcal{P}})$*. Hierbei gilt* $\text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) = \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$.

**Beweis.** Wir zeigen zunächst, dass $\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$ ein Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$ ist. Es gilt

$$\begin{array}{lcll} \underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)) & = & \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^{i+1}_{\mathcal{P}}(\varnothing) & \text{(wegen Stetigkeit, Lemma 4.1.13 (b))} \\ & = & \varnothing \cup \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^{i+1}_{\mathcal{P}}(\varnothing) & \\ & = & \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^{i}_{\mathcal{P}}(\varnothing). & \end{array}$$

Nun zeigen wir, dass $\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$ der kleinste Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$ ist. Sei hierzu $M$ ein weiterer Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$. Um $\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) \subseteq M$ zu beweisen, reicht es, zu zeigen, dass $\underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) \subseteq M$ für alle $i \in \mathbb{N}$ gilt. Dies zeigen wir durch Induktion über $i$. Im Induktionsanfang ($i = 0$) gilt offensichtlich $\underline{\text{trans}}^0_{\mathcal{P}}(\varnothing) = \varnothing \subseteq M$.

Im Induktionsschritt setzen wir als Induktionshypothese $\underline{\text{trans}}^{i-1}_{\mathcal{P}}(\varnothing) \subseteq M$ voraus. Aufgrund der Monotonie von $\underline{\text{trans}}_{\mathcal{P}}$ (Lemma 4.1.13 (a)) folgt daraus $\underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) \subseteq \underline{\text{trans}}_{\mathcal{P}}(M) = M$, denn $M$ ist ein Fixpunkt von $\underline{\text{trans}}_{\mathcal{P}}$. □

Nun können wir die Fixpunkt–Semantik definieren.

**Definition 4.1.15 (Fixpunkt–Semantik eines Logikprogramms)** *Sei $\mathcal{P}$ ein Logikprogramm und $G = \{\neg A_1, \ldots, \neg A_k\}$ eine Anfrage. Hierbei sind $A_1$, …, $A_k$ also atomare Formeln. Dann ist die* Fixpunkt–Semantik *von $\mathcal{P}$ bezüglich $G$ definiert als*

$$F[\![\mathcal{P}, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \sigma(A_i) \in \text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) \text{ für alle } i\}.$$

Der folgende Satz zeigt, dass auch die Fixpunkt–Semantik äquivalent zu den beiden bisher eingeführten Semantik–Definitionen ist.

**Satz 4.1.16 (Äquivalenz der Fixpunkt–Semantik zu den anderen Semantiken)** *Sei $\mathcal{P}$ ein Logikprogramm und $G = \{\neg A_1, \ldots, \neg A_k\}$ eine Anfrage. Dann gilt $F[\![\mathcal{P}, G]\!] = D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$.*

**Beweis.** Da nach Satz 4.1.8 $D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$ gilt, genügt es, $P[\![\mathcal{P}, G]\!] \subseteq F[\![\mathcal{P}, G]\!]$ und $F[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$ zu zeigen.

**Teil 1:** $P[\![\mathcal{P}, G]\!] \subseteq F[\![\mathcal{P}, G]\!]$

Der Beweis ist analog zum Beweis von $P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$ in Satz 4.1.8. Sei $\sigma'(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, G]\!]$. Dann gibt es eine erfolgreiche Berechnung der Form

$$(G, \varnothing) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma),$$

wobei $\sigma'(A_1 \wedge \ldots \wedge A_k)$ eine Grundinstanz von $\sigma(A_1 \wedge \ldots \wedge A_k)$ ist. Zu zeigen ist, dass dann $\sigma'(A_i) \in \text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) = \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$ für alle $i$ gilt. Genauer zeigen wir, dass es ein $j \in \mathbb{N}$ gibt, so dass $\underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$ alle Grundinstanzen von $\sigma(A_i)$ für alle $i$ enthält.

Wir verwenden Induktion über die Länge $l$ der Berechnung. Die Berechnung hat also die Form

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1)$$

mit $\sigma = \delta_l \circ \ldots \circ \delta_1$. Es existiert also ein $A_i$, ein $K \in \mathcal{P}$ und eine Variablenumbenennung $\nu$ mit $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ und $n \geq 0$, so dass $G$ und $\nu(K)$ keine gemeinsamen Variablen haben, so dass $\delta_1$ der mgu von $A_i$ und $B$ ist und so dass

$$G_1 = \delta_1(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

Im Induktionsanfang ist $l = 1$. Dann gilt $G_1 = \square$ und somit $i = k = 1$, $n = 0$, $\nu(K) = \{B\}$ (d.h., die Programmklausel $K$ ist ein Faktum) und $\sigma = \delta_1$. Da $\underline{\text{trans}}_{\mathcal{P}}(\varnothing)$ alle Grundinstanzen aller Fakten wie $B$ enthält, enthält also $\underline{\text{trans}}_{\mathcal{P}}(\varnothing)$ auch alle Grundinstanzen von $\delta_1(B) = \delta_1(A_1) = \sigma(A_1)$.

Im Induktionsschritt ist $l > 1$. Dann ist offensichtlich auch

$$(G_1, \varnothing) \vdash_{\mathcal{P}} (G_2, \delta_2) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_2)$$

eine Berechnung. Nach der Induktionshypothese existiert ein $j$, so dass $\underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$ alle Grundinstanzen von $\delta_l(\ldots\delta_2(\delta_1(A_p))\ldots)$ mit $p \in \{1, \ldots, k\} \setminus \{i\}$ und alle Grundinstanzen von $\delta_l(\ldots\delta_2(\delta_1(C_p))\ldots)$ mit $p \in \{1, \ldots, n\}$ enthält. Da $\{B, \neg C_1, \ldots, \neg C_n\}$ durch Umbenennung aus einer Klausel von $\mathcal{P}$ entstanden ist, ist dann jede Grundinstanz von $\delta_l(\ldots\delta_2(\delta_1(B))\ldots)$ in $\underline{\text{trans}}^{j+1}_{\mathcal{P}}(\varnothing)$ enthalten. Da $\delta_1(B) = \delta_1(A_i)$ gilt, enthält $\underline{\text{trans}}^{j+1}_{\mathcal{P}}(\varnothing)$ also auch alle Grundinstanzen von $\delta_l(\ldots\delta_2(\delta_1(A_i))\ldots)$.

**Teil 2:** $F[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$

Sei $\sigma(A_1 \wedge \ldots \wedge A_k) \in F[\![\mathcal{P}, G]\!]$. Dann gilt $\sigma(A_i) \in \text{lfp}(\underline{\text{trans}}_{\mathcal{P}})$ für alle $i$. Zu zeigen ist, dass dann auch $\mathcal{P} \models \sigma(A_i)$ gilt. Wir zeigen durch Induktion über $j$, dass für alle $A' \in \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$ jeweils $\mathcal{P} \models A'$ gilt. Daraus folgt die Behauptung, da aus $\text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) = \bigcup_{j \in \mathbb{N}} \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$ folgt, dass es ein $j \in \mathbb{N}$ mit $\sigma(A_i) \in \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$ gibt.

Der Induktionsanfang $j = 0$ ist trivial, da $\underline{\text{trans}}^0_{\mathcal{P}}(\varnothing) = \varnothing$ gilt. Im Induktionsschluss sei $j > 0$ und $A' \in \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$. Falls bereits $A' \in \underline{\text{trans}}^{j-1}_{\mathcal{P}}(\varnothing)$ gilt, so folgt die Behauptung aus der Induktionshypothese. Ansonsten existiert eine Klausel $\{A, \neg B_1, \ldots, \neg B_n\} \in \mathcal{P}$ mit einer Grundinstanz $\{A', \neg B'_1, \ldots, \neg B'_n\}$, so dass $B'_p \in \underline{\text{trans}}^{j-1}_{\mathcal{P}}(\varnothing)$ für alle $p \in \{1, \ldots, n\}$ gilt. Nach der Induktionshypothese gilt also $\mathcal{P} \models B'_p$ für alle $p$ und somit also auch $\mathcal{P} \models A'$. $\square$

## 4.2 Universalität der Logikprogrammierung

Nachdem wir nun die Syntax und Semantik der Logikprogrammierung festgelegt haben, wollen wir zeigen, dass die Logikprogrammierung in der Tat eine vollwertige Programmiersprache darstellt. Man kann also durch Logikprogramme *jede berechenbare* Funktion berechnen. Solche Programmiersprachen bezeichnet man auch als "*Turing–vollständig*".

Wie üblich beschränken wir uns bei der Betrachtung der berechenbaren Funktionen auf arithmetische Funktionen $f : \mathbb{N}^n \to \mathbb{N}$ über den natürlichen Zahlen. Der Grund ist, dass man alle anderen Datenstrukturen durch eine entsprechende Abbildung in die natürlichen Zahlen codieren kann.

Es existieren verschiedene Ansätze, um die Menge der berechenbaren Funktionen zu charakterisieren. Eine Möglichkeit ist es, die Menge der "berechenbaren" Funktionen als die Menge der Funktionen zu definieren, die durch Turing–Maschinen berechenbar sind. Zeitlich parallel zu Turings Ansatz entstand der folgende Ansatz von Kleene, der die Menge der berechenbaren Funktionen als die Klasse der $\mu$–rekursiven Funktionen definiert. Alle diese Ansätze führen zu derselben Menge von Funktionen, d.h., die Menge der berechenbaren Funktionen ist immer dieselbe, unabhängig davon, ob man sie mit Hilfe von

Turing–Maschinen oder mit Hilfe von $\mu$–rekursiven Funktionen definiert. Dies führte zur *Churchschen These*, die besagt, dass diese Menge tatsächlich die Menge aller im intuitiven Sinne berechenbaren Funktionen ist.

Die Klasse der $\mu$–rekursiven Funktionen ist induktiv definiert, indem man von bestimmten Basisfunktionen ausgeht und dann verschiedene Prinzipien angibt, wie aus $\mu$–rekursiven Funktionen neue $\mu$–rekursive Funktionen gewonnen werden können.

**Definition 4.2.1 ($\mu$–rekursive Funktionen)** *Die Klasse der* $\mu$–rekursiven Funktionen *ist die kleinste Klasse arithmetischer Funktionen mit:*

1. *Für jedes* $n \in \mathbb{N}$ *ist die Funktion* $\mathsf{null}_n : \mathbb{N}^n \to \mathbb{N}$ *mit* $\mathsf{null}_n(k_1, \ldots, k_n) = 0$ *$\mu$–rekursiv.*

2. *Die Nachfolgerfunktion* $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$ *mit* $\mathsf{succ}(k) = k + 1$ *ist $\mu$–rekursiv.*

3. *Für jedes* $n \geq 1$ *und jedes* $1 \leq i \leq n$ *ist die Projektionsfunktion* $\mathsf{proj}_{n,i}(k_1, \ldots, k_n) = k_i$ *$\mu$–rekursiv.*

4. *Die $\mu$–rekursiven Funktionen sind unter* Komposition *abgeschlossen. Für alle* $m \geq 1$ *und alle* $n \geq 0$ *gilt also: Falls* $f : \mathbb{N}^m \to \mathbb{N}$ *und* $f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N}$ *$\mu$–rekursiv sind, dann ist auch die folgende Funktion* $g : \mathbb{N}^n \to \mathbb{N}$ *$\mu$–rekursiv:*

$$g(k_1, \ldots, k_n) = f(f_1(k_1, \ldots, k_n), \ldots, f_m(k_1, \ldots, k_n))$$

5. *Die $\mu$–rekursiven Funktionen sind unter* primitiver Rekursion *abgeschlossen. Für alle* $n \geq 0$ *gilt also: Falls* $f : \mathbb{N}^n \to \mathbb{N}$ *und* $g : \mathbb{N}^{n+2} \to \mathbb{N}$ *$\mu$–rekursiv sind, dann ist auch die folgende Funktion* $h : \mathbb{N}^{n+1} \to \mathbb{N}$ *$\mu$–rekursiv:*

$$\begin{array}{rcl} h(k_1, \ldots, k_n, 0) & = & f(k_1, \ldots, k_n) \\ h(k_1, \ldots, k_n, k+1) & = & g(k_1, \ldots, k_n, k, h(k_1, \ldots, k_n, k)) \end{array}$$

   *Primitive Rekursion bedeutet also, dass die Definition von* $h(\ldots, k+1)$ *auf die Definition von* $h(\ldots, k)$ *zurückgeführt wird.*

6. *Die $\mu$–rekursiven Funktionen sind unter (unbeschränkter)* Minimalisierung *abgeschlossen. Für alle* $n \geq 0$ *gilt also: Falls* $f : \mathbb{N}^{n+1} \to \mathbb{N}$ *$\mu$–rekursiv ist, dann ist auch die folgende Funktion* $g : \mathbb{N}^n \to \mathbb{N}$ *$\mu$–rekursiv:*

$$g(k_1, \ldots, k_n) = k \quad \textit{gdw.} \quad \begin{array}{l} f(k_1, \ldots, k_n, k) = 0 \textit{ und für alle } 0 \leq k' < k \textit{ ist} \\ f(k_1, \ldots, k_n, k') \textit{ definiert und } f(k_1, \ldots, k_n, k') > 0 \end{array}$$

   *Falls es kein solches* $k$ *gibt, dann ist* $g(k_1, \ldots, k_n)$ *undefiniert, d.h., durch die Minimalisierung können auch partielle Funktionen entstehen.*

*Die Klasse der Funktionen, die nur mit Hilfe der Punkte 1 – 5 konstruiert werden, ist die Klasse der* primitiv rekursiven *Funktionen.*

Die Klasse der primitiv rekursiven Funktionen kann natürlich nicht alle berechenbaren Funktionen enthalten, da alle primitiv rekursiven Funktionen total sind und es aber (durch die Verwendung von nicht–terminierenden Programmen) offensichtlich auch berechenbare partielle Funktionen gibt. (Es existieren aber auch totale berechenbare und nicht primitiv rekursive Funktionen, wie z.B. die sogenannte *Ackermann*–Funktion.) Die folgenden Beispiele verdeutlichen die Definition der primitiv rekursiven und der $\mu$–rekursiven Funktionen.

**Beispiel 4.2.2** *Die Additionsfunktion* $\mathsf{plus} : \mathbb{N}^2 \to \mathbb{N}$ *ist primitiv rekursiv, denn sie kann wie folgt mit Hilfe von primitiver Rekursion dargestellt werden. Hierbei ist* $\mathsf{f}$ *die dreistellige Funktion mit* $\mathsf{f}(x, y, z) = z + 1$*:*

$$\begin{array}{rcl} \mathsf{f}(x,y,z) & = & \mathsf{succ}(\mathsf{proj}_{3,3}(x,y,z)) \\ \mathsf{plus}(x,0) & = & \mathsf{proj}_{1,1}(x) \\ \mathsf{plus}(x,y+1) & = & \mathsf{f}(x,y,\mathsf{plus}(x,y)) \end{array}$$

*Ebenso ist auch die Multiplikationsfunktion* $\mathsf{times} : \mathbb{N}^2 \to \mathbb{N}$ *primitiv rekursiv. Hierbei ist* $\mathsf{g}$ *die dreistellige Funktion mit* $\mathsf{g}(x, y, z) = x + z$*:*

$$\begin{array}{rcl} \mathsf{g}(x,y,z) & = & \mathsf{plus}(\mathsf{proj}_{3,1}(x,y,z),\mathsf{proj}_{3,3}(x,y,z)) \\ \mathsf{times}(x,0) & = & \mathsf{null}_1(x) \\ \mathsf{times}(x,y+1) & = & \mathsf{g}(x,y,\mathsf{times}(x,y)) \end{array}$$

*Eine weitere primitiv rekursive Funktion ist die Vorgängerfunktion* $\mathsf{p} : \mathbb{N} \to \mathbb{N}$ *mit* $\mathsf{p}(0) = 0$ *und* $\mathsf{p}(x + 1) = x$*:*

$$\begin{array}{rcl} \mathsf{p}(0) & = & \mathsf{null}_0 \\ \mathsf{p}(x+1) & = & \mathsf{proj}_{2,1}(x,\mathsf{p}(x)) \end{array}$$

*Auch die folgende Funktion* $\mathsf{minus} : \mathbb{N}^2 \to \mathbb{N}$ *ist primitiv rekursiv. Es gilt* $\mathsf{minus}(x, y) = 0$ *falls* $x \leq y$ *und* $\mathsf{minus}(x, y) = x - y$ *sonst. (Genauer erhält man* $\mathsf{minus}(x, y) = \mathsf{p}^y(x)$*.)*

$$\begin{array}{rcl} \mathsf{h}(x,y,z) & = & \mathsf{p}(\mathsf{proj}_{3,3}(x,y,z)) \\ \mathsf{minus}(x,0) & = & \mathsf{proj}_{1,1}(x) \\ \mathsf{minus}(x,y+1) & = & \mathsf{h}(x,y,\mathsf{minus}(x,y)) \end{array}$$

*Schließlich zeigen wir noch, dass die Funktion* $\mathsf{div} : \mathbb{N}^2 \to \mathbb{N}$ *$\mu$-rekursiv ist. Hierbei gilt* $\mathsf{div}(x, y) = \lceil \frac{x}{y} \rceil$ *und* $\mathsf{div}(0, 0) = 0$*. Hingegen ist* $\mathsf{div}(x + 1, 0)$ *undefiniert.*

$$\mathsf{div}(x,y) = z \quad \textit{gdw.} \quad \begin{array}{l} \mathsf{i}(x,y,z) = 0 \textit{ und für alle } 0 \leq z' < z \textit{ ist } \mathsf{i}(x,y,z') \textit{ definiert} \\ \textit{und } \mathsf{i}(x,y,z') > 0 \end{array}$$

*Hierbei berechnet* $\mathsf{i}(x, y, z) = x - (y \cdot z)$*, d.h.*

$$\begin{array}{rcl} \mathsf{i}(x,y,z) & = & \mathsf{minus}(\mathsf{proj}_{3,1}(x,y,z),\mathsf{j}(x,y,z)) \\ \mathsf{j}(x,y,z) & = & \mathsf{times}(\mathsf{proj}_{3,2}(x,y,z),\mathsf{proj}_{3,3}(x,y,z)) \end{array}$$

Nun wollen wir zeigen, dass man jede berechenbare (d.h. jede $\mu$–rekursive Funktion) auch durch ein Logikprogramm berechnen kann. Hierzu müssen wir zunächst festlegen, wie überhaupt Funktionen mit Logikprogrammen berechnet werden können. Der Grund ist, dass man in Logikprogrammen ja nur Relationen und keine Funktionen definiert. Die Lösung besteht darin, eine $n$-stellige Funktion $f : \mathbb{N}^n \to \mathbb{N}$ stattdessen durch eine $(n + 1)$-stellige Relation $\underline{\mathsf{f}} \subseteq \mathbb{N}^{n+1}$ darzustellen, die gerade dem Graphen von $f$ entspricht.

Ebenso müssen wir erklären, wie man natürliche Zahlen durch Terme darstellen kann, da ja Logikprogramme nur auf Termen arbeiten. Hierzu verwenden wir eine Darstellung mit Hilfe der Funktionssymbole $\mathtt{0} \in \Sigma_0$ und $\mathtt{s} \in \Sigma_1$, wobei $\mathtt{0}$ die Zahl 0 darstellt, $\mathtt{s}(\mathtt{0})$ die Zahl 1 darstellt, $\mathtt{s}(\mathtt{s}(\mathtt{0}))$ die Zahl 2 darstellt, etc. Das Funktionssymbol $\mathtt{s}$ repräsentiert also die Nachfolgerfunktion.

**Definition 4.2.3 (Berechnung arith. Funktionen durch Logikprogramme)**

- *Jede Zahl $k \in \mathbb{N}$ wird durch den Term $\underline{k} \in \mathcal{T}(\Sigma, \mathcal{V})$ mit $\underline{k} = \mathtt{s}^k(\mathtt{0})$ dargestellt, wobei $\mathtt{0} \in \Sigma_0$ und $\mathtt{s} \in \Sigma_1$ ist.*
- *Ein Logikprogramm $\mathcal{P}$ über $(\Sigma, \Delta)$ berechnet eine arithmetische Funktion $f : \mathbb{N}^n \to \mathbb{N}$ gdw. es ein Prädikatssymbol $\underline{\mathtt{f}} \in \Delta_{n+1}$ gibt, so dass*

$$f(k_1, \ldots, k_n) = k \quad \text{gdw.} \quad \mathcal{P} \models \underline{\mathtt{f}}(\underline{k_1}, \ldots, \underline{k_n}, \underline{k}).$$

Die Motivation für die obige Definition ist, dass man dann die Funktion $f$ durch Anfragen an das Logikprogramm berechnen lassen kann. Um $f(k_1, \ldots, k_n)$ zu berechnen, stellt man dem Logikprogramm die Anfrage `?-` $\underline{\mathtt{f}}(\underline{k_1}, \ldots, \underline{k_n}, \mathtt{X})$.

**Beispiel 4.2.4** *Das folgende Logikprogramm berechnet die Funktionen aus Bsp. 4.2.2. Die angegebenen Klauseln entstehen allerdings nicht direkt aus der Überführung der Definitionen von Bsp. 4.2.2, sondern wir haben hier Definitionen gewählt, die möglichst lesbar sind. (Dies betrifft insbesondere die Klauseln des Divisionsprädikats* `div`*.)*

```
plus(X,0,X).
plus(X,s(Y),s(Z)) :- plus(X,Y,Z).

times(X,0,0).
times(X,s(Y),Z) :- times(X,Y,U), plus(X,U,Z).

p(0,0).
p(s(X),X).

minus(X,0,X).
minus(X,s(Y),Z) :- minus(X,Y,U), p(U,Z).

div(0,Y,0).
div(s(X),s(Y),s(Z)) :- minus(X,Y,U), div(U,s(Y),Z).
```

Der folgende Satz beweist schließlich die Universalität der Logikprogrammierung.

**Satz 4.2.5 (Universalität der Logikprogrammierung)** *Jede $\mu$–rekursive Funktion ist durch ein Logikprogramm berechenbar.*

**Beweis.** Der Beweis wird durch Induktion über den Aufbau der Klasse der $\mu$–rekursiven Funktionen geführt.

1. Die Funktion $\mathsf{null}_n$ wird durch das folgende Logikprogramm berechnet:
   $\underline{\mathtt{null_n}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{0})$.
2. Die Nachfolgerfunktion succ wird durch das folgende Logikprogramm berechnet:
   $\underline{\mathtt{succ}}(\mathtt{X}, \mathtt{s}(\mathtt{X}))$.

3. Die Projektionsfunktion $\mathsf{proj}_{n,i}$ wird durch das folgende Logikprogramm berechnet:

   $\underline{\mathtt{proj_{n,i}}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{X_i})$.

4. Nun zeigen wir, wie die Komposition durch Logikprogramme realisiert werden kann. Seien $f : \mathbb{N}^m \to \mathbb{N}$ und $f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N}$ $\mu$–rekursiv. Dann existiert nach der Induktionshypothese ein Logikprogramm mit Prädikaten $\underline{\mathtt{f}} \in \Delta_{m+1}$ und $\underline{\mathtt{f_1}}, \ldots, \underline{\mathtt{f_m}} \in \Delta_{n+1}$, das diese Funktionen berechnet. Sei $g$ durch Komposition von $f$ mit $f_1, \ldots, f_m$ definiert, d.h.

   $$g(k_1, \ldots, k_n) = f(f_1(k_1, \ldots, k_n), \ldots, f_m(k_1, \ldots, k_n)).$$

   Zur Berechnung von $g$ wird das Logikprogramm um die folgende Klausel ergänzt:

   $\underline{\mathtt{g}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Z})$ :- $\underline{\mathtt{f_1}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y_1}), \ldots, \underline{\mathtt{f_m}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y_m}), \underline{\mathtt{f}}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z})$.

5. Als nächstes zeigen wir, wie die primitive Rekursion in der Logikprogrammierung realisiert werden kann. Seien $f : \mathbb{N}^n \to \mathbb{N}$ und $g : \mathbb{N}^{n+2} \to \mathbb{N}$ $\mu$–rekursiv. Dann existiert nach der Induktionshypothese ein Logikprogramm mit Prädikaten $\underline{\mathtt{f}} \in \Delta_{n+1}$ und $\underline{\mathtt{g}} \in \Delta_{n+3}$, das diese Funktionen berechnet. Sei $h$ durch primitive Rekursion mit $f$ und $g$ definiert, d.h.

   $$\begin{aligned} h(k_1, \ldots, k_n, 0) &= f(k_1, \ldots, k_n) \\ h(k_1, \ldots, k_n, k+1) &= g(k_1, \ldots, k_n, k, h(k_1, \ldots, k_n, k)) \end{aligned}$$

   Zur Berechnung von $h$ wird das Logikprogramm um die folgenden Klauseln ergänzt:

   $\underline{\mathtt{h}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{0}, \mathtt{Z})$ :- $\underline{\mathtt{f}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Z})$.

   $\underline{\mathtt{h}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{s(X)}, \mathtt{Z})$ :- $\underline{\mathtt{h}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{X}, \mathtt{Y}), \underline{\mathtt{g}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{X}, \mathtt{Y}, \mathtt{Z})$.

6. Schließlich zeigen wir, wie die Minimalisierung durch Logikprogramme realisiert werden kann. Sei $f : \mathbb{N}^{n+1} \to \mathbb{N}$ $\mu$–rekursiv. Dann existiert nach der Induktionshypothese ein Logikprogramm mit einem Prädikat $\underline{\mathtt{f}} \in \Delta_{n+2}$, das diese Funktion berechnet. Sei $g$ durch Minimalisierung mit $f$ definiert, d.h.

   $$g(k_1, \ldots, k_n) = k \quad \text{gdw.} \quad \begin{array}{l} f(k_1, \ldots, k_n, k) = 0 \text{ und für alle } 0 \leq k' < k \text{ ist} \\ f(k_1, \ldots, k_n, k') \text{ definiert und } f(k_1, \ldots, k_n, k') > 0 \end{array}$$

   Zur Berechnung von $g$ wird das Logikprogramm um die folgenden Klauseln ergänzt. Hierbei gilt $\underline{\mathtt{f}}'(X_1, \ldots, X_n, Y, Z)$ gdw. $f(X_1, \ldots, X_n, Z) = 0$ und für alle $X$ mit $Y \leq X < Z$ ist $f(X_1, \ldots, X_n, X) > 0$.

   $\underline{\mathtt{g}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Z})$ :- $\underline{\mathtt{f}}'(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{0}, \mathtt{Z})$.

   $\underline{\mathtt{f}}'(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y}, \mathtt{Y})$ :- $\underline{\mathtt{f}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y}, \mathtt{0})$.
   $\underline{\mathtt{f}}'(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y}, \mathtt{Z})$ :- $\underline{\mathtt{f}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y}, \mathtt{s(U)}), \underline{\mathtt{f}}'(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{s(Y)}, \mathtt{Z})$.

□

**Beispiel 4.2.6** *Das folgende Logikprogramm entsteht, indem man das Vorgehen aus dem Beweis von Satz 4.2.5 auf die Definition der* plus*–Funktion aus Bsp. 4.2.2 anwendet.*

$\underline{\texttt{proj}_{3,3}}(\texttt{X},\texttt{Y},\texttt{Z},\texttt{Z})$.

$\underline{\texttt{succ}}(\texttt{X},\texttt{s}(\texttt{X}))$.

$\underline{\texttt{f}}(\texttt{X},\texttt{Y},\texttt{Z},\texttt{V})$ :- $\underline{\texttt{proj}_{3,3}}(\texttt{X},\texttt{Y},\texttt{Z},\texttt{U}), \underline{\texttt{succ}}(\texttt{U},\texttt{V})$.

$\underline{\texttt{proj}_{1,1}}(\texttt{X},\texttt{X})$.

$\underline{\texttt{plus}}(\texttt{X},\texttt{0},\texttt{U})$ :- $\underline{\texttt{proj}_{1,1}}(\texttt{X},\texttt{U})$.
$\underline{\texttt{plus}}(\texttt{X},\texttt{s}(\texttt{Y}),\texttt{U})$ :- $\underline{\texttt{plus}}(\texttt{X},\texttt{Y},\texttt{Z}), \underline{\texttt{f}}(\texttt{X},\texttt{Y},\texttt{Z},\texttt{U})$.

*Wenn man das Vorgehen aus dem Beweis auf die Definition der* div*–Funktion aus Bsp. 4.2.2 anwendet, erhält man die folgenden Klauseln:*

$\underline{\texttt{div}}(\texttt{X1},\texttt{X2},\texttt{Z})$ :- $\underline{\texttt{i}'}(\texttt{X1},\texttt{X2},\texttt{0},\texttt{Z})$.

$\underline{\texttt{i}'}(\texttt{X1},\texttt{X2},\texttt{Y},\texttt{Y})$ :- $\underline{\texttt{i}}(\texttt{X1},\texttt{X2},\texttt{Y},\texttt{0})$.
$\underline{\texttt{i}'}(\texttt{X1},\texttt{X2},\texttt{Y},\texttt{Z})$ :- $\underline{\texttt{i}}(\texttt{X1},\texttt{X2},\texttt{Y},\texttt{s}(\texttt{U})), \underline{\texttt{i}'}(\texttt{X1},\texttt{X2},\texttt{s}(\texttt{Y}),\texttt{Z})$.

*Hierbei gilt* $\underline{\texttt{i}}(\texttt{X},\texttt{Y},\texttt{Z},\texttt{U})$ *gdw.* $\texttt{X} - (\texttt{Y} \cdot \texttt{Z}) = \texttt{U}$ *ist.*

## 4.3 Indeterminismus und Auswertungsstrategien

Um Logikprogramme *auszuführen*, geht man analog zur Definition der prozeduralen Semantik vor. Falls an das Logikprogramm $\mathcal{P}$ die Anfrage $G$ gestellt wird, versucht man also, eine Berechnung $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma)$ zu finden und gibt dann als Ergebnis die Antwortsubstitution $\sigma$ eingeschränkt auf die Variablen von $G$ aus.

Man erkennt aber, dass die Definition von $\vdash_{\mathcal{P}}$ (Def. 4.1.5) bei jedem Schritt $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ noch zwei Indeterminismen aufweist:

- **Indeterminismus 1. Art**: Dieser Indeterminismus ist die Wahl der Programmklausel $K \in \mathcal{P}$, mit der $G_1$ resolviert werden soll.

- **Indeterminismus 2. Art**: Dieser Indeterminismus ist die Wahl des Literals $A_i$ in $G_1$, das zur Resolution verwendet werden soll.

Insofern kann es zu einem $(G_1, \sigma_1)$ *mehrere* $(G_2, \sigma_2)$ mit $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ geben. Dies wird durch folgendes Beispiel illustriert.

**Beispiel 4.3.1** *Wir betrachten die folgende Variante des Logikprogramms aus Kapitel 1.*

```
mutterVon(renate, susanne).
mutterVon(susanne, aline).

vorfahre(V,X) :- mutterVon(V,X).
vorfahre(V,X) :- mutterVon(V,Y), vorfahre(Y,X).
```

*und die Anfrage*

```
?- vorfahre(Z,aline).
```

*Im ersten Berechnungsschritt gibt es nun zwei Möglichkeiten, da man mit zwei verschiedenen Programmklauseln resolvieren kann. Dies entspricht also dem Indeterminismus 1. Art:*

$$(\{\neg\mathsf{vorfahre}(Z,\mathsf{aline})\},\varnothing) \vdash_{\mathcal{P}} (\{\neg\mathsf{mutterVon}(Z,\mathsf{aline})\},\{V/Z,X/\mathsf{aline}\})$$
$$(\{\neg\mathsf{vorfahre}(Z,\mathsf{aline})\},\varnothing) \vdash_{\mathcal{P}} (\{\neg\mathsf{mutterVon}(Z,Y),\neg\mathsf{vorfahre}(Y,\mathsf{aline})\},\{V/Z,X/\mathsf{aline}\})$$

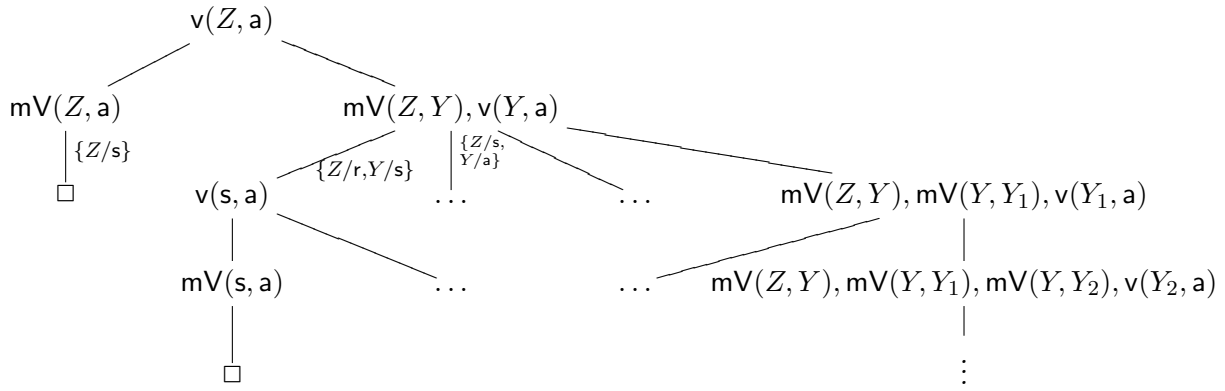
*Falls man sich für den zweiten Berechnungsschritt entscheidet, so muss man nun die Anfrage*

```
?- mutterVon(Z,Y), vorfahre(Y,aline).
```

*bearbeiten. Nun muss man auch den Indeterminismus 2. Art behandeln. Falls man sich zur Resolution mit Hilfe des ersten Literals* $\neg\mathsf{mutterVon}(Z,Y)$ *entscheidet, so gibt es (wegen des Indeterminismus 1. Art und der zwei* mutterVon*-Fakten) zwei Möglichkeiten fortzufahren. Resolviert man mit der Klausel* $\{\mathsf{mutterVon}(\mathsf{susanne},\mathsf{aline})\}$*, so kann man keine erfolgreiche Berechnung mehr erhalten. Resolviert man hingegen mit der Klausel* $\{\mathsf{mutterVon}(\mathsf{renate},\mathsf{susanne})\}$*, so kann man eine erfolgreiche Berechnung erzeugen, die zur Antwortsubstitution* $\{Z/\mathsf{renate}\}$ *führt.*

*Falls man sich zur Resolution mit Hilfe des zweiten Literals* $\neg\mathsf{vorfahre}(Y,\mathsf{aline})$ *entscheidet, so gibt es ebenfalls (wegen des Indeterminismus 1. Art und der zwei* vorfahre*-Fakten) zwei Möglichkeiten fortzufahren etc. Insbesondere gibt es auch die Möglichkeit einer unendlichen Berechnung, indem man beim Indeterminismus 1. Art stets die zweite (rekursive)* vorfahre*–Klausel nimmt und beim Indeterminismus 2. Art stets das letzte* vorfahre*–Literal.*

*Der folgende Baum zeigt die Möglichkeiten für Berechnungen auf. Hierbei wurde in den Knoten anstelle eines Paars* $(\{\neg A_1,\ldots,\neg A_k\},\sigma)$ *jeweils nur die Atome* $A_1,\ldots,A_k$ *in der Anfrage angegeben. Ein Berechnungsschritt der Form* $(G_1,\sigma)\vdash_{\mathcal{P}}(G_2,\sigma'\circ\sigma)$ *wird durch eine Kante dargestellt, die mit* $\sigma'$ *(eingeschränkt auf die Variablen in* $G_1$*) markiert wird. Auf diese Weise kann man bei erfolgreichen Pfaden direkt die Antwortsubstitution ablesen. Zur Abkürzung steht "*$\mathsf{v}$*" für "*vorfahre*","*$\mathsf{mV}$*" für "*mutterVon*", "*$\mathsf{s}$*" für "*susanne*", "*$\mathsf{a}$*" für "*aline*" und "*$\mathsf{r}$*" für "*renate*".*



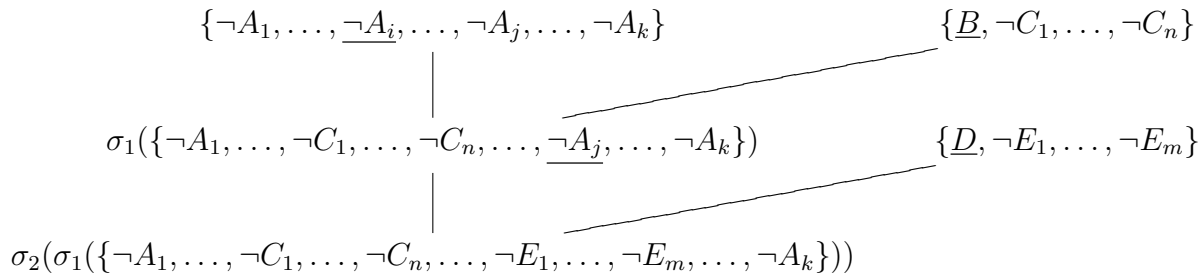
*Man erkennt, dass der Indeterminismus 1. Art die Lösung beeinflusst: bei der linkesten erfolgreichen Berechnung erhält man die Antwortsubstitution* $\{Z/\mathsf{susanne}\}$ *und bei der zweiten erfolgreichen Berechnung erhält man* $\{Z/\mathsf{renate}\}$*. Ebenso erkennt man, dass der Indeterminismus 2. Art beeinflusst, ob man eine unendliche oder eine endliche Berechnung erhält, da der rechteste Pfad in diesem Baum unendlich ist.*

Um Logikprogramme auf einem (deterministischen) Rechner ausführen zu können, müssen wir diese beiden Indeterminismen auflösen. Wir müssen also eindeutig festlegen, in welches Paar $(G_2, \sigma_2)$ ein Paar $(G_1, \sigma_1)$ überführt werden soll.

Hierzu betrachten wir zunächst den Indeterminismus 2. Art, d.h., die Auswahl des Literals in der Anfrage, das zur Resolution verwendet werden soll. Hier zeigen wir, dass dieser Indeterminismus "unerheblich" ist, da er das Ergebnis (bei erfolgreichen Berechnungen) nicht beeinflusst. Man braucht also nicht mehr alle Möglichkeiten zur Auswahl des Literals betrachten, sondern man kann diesen Indeterminismus beliebig auflösen und behält trotzdem ein vollständiges Verfahren.

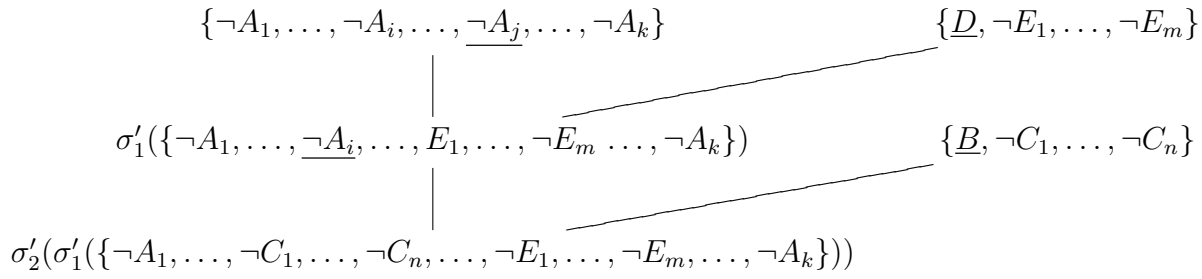
Das folgende Vertauschungslemma wird für diesen Satz benötigt und zeigt, dass die Reihenfolge der Literale in der Anfrage, mit denen resolviert wird, vertauscht werden kann.

**Lemma 4.3.2 (Vertauschungslemma)** *Seien* $A_1, \ldots, A_k, B, C_1, \ldots, C_n, D, E_1, \ldots, E_m \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V})$*. Hierbei seien* $\{\neg A_1, \ldots, \neg A_k\}$*,* $\{B, \neg C_1, \ldots, \neg C_n\}$ *und* $\{D, \neg E_1, \ldots, \neg E_m\}$ *jeweils paarweise variablendisjunkt. Sei* $\sigma_1$ *der mgu von* $A_i$ *und* $B$ *und* $\sigma_2$ *der mgu von* $\sigma_1(A_j)$ *und* $D$ *mit* $j \neq i$*. Dann sind daher offensichtlich die folgenden (binären) SLD–Resolutionsschritte möglich:*




*Wie üblich wurden in dem Diagramm Resolutionsschritte wieder durch Kanten von den Elternklauseln zu den Resolventen gekennzeichnet und die jeweils zur Resolution verwendeten Literale wurden durch Unterstreichen gekennzeichnet.*

*Dann existieren auch ein mgu* $\sigma_1'$ *von* $A_j$ *und* $D$ *und ein mgu* $\sigma_2'$ *von* $\sigma_1'(A_i)$ *und* $B$*. Daher sind dann die folgenden (binären) SLD–Resolutionsschritte möglich:*



*Darüber hinaus unterscheiden sich die Substitutionen $\sigma_2 \circ \sigma_1$ und $\sigma_2' \circ \sigma_1'$ nur durch eine Variablenumbenennung (d.h., es existiert eine Variablenumbenennung $\nu$ mit $\sigma_2' \circ \sigma_1' = \nu \circ \sigma_2 \circ \sigma_1$).*

**Beweis.** Aufgrund der Variablendisjunktheit der Klauseln können wir o.B.d.A. davon ausgehen, dass der mgu $\sigma_1$ von $A_i$ und $B$ die Variablen von $D$ nicht verändert, d.h., $\sigma_1(D) = D$. Dann ist $\sigma_2$ nicht nur Unifikator von $\sigma_1(A_j)$ und $D$, sondern auch Unifikator von $\sigma_1(A_j)$ und $\sigma_1(D)$. Aus $\sigma_2(\sigma_1(A_j)) = \sigma_2(\sigma_1(D))$ folgt also, dass $A_j$ und $D$ mit dem Unifikator $\sigma_2 \circ \sigma_1$ unifizierbar sind und daher einen mgu $\sigma_1'$ haben. Es existiert also eine Substitution $\sigma$ mit

$$\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'. \tag{4.5}$$

Damit lässt sich also wie gewünscht auch zuerst ein Resolutionsschritt mit Hilfe des Literals $\neg A_j$ durchführen. Um den zweiten gewünschten Resolutionsschritt durchführen zu können, muss man zeigen, dass $\sigma_1'(A_i)$ und $B$ unifizierbar ist. Dies gilt in der Tat, denn $\sigma$ ist ein Unifikator:

$$\begin{array}{rcll} \sigma(\sigma_1'(A_i)) & = & \sigma_2(\sigma_1(A_i)) & \text{nach (4.5)} \\ & = & \sigma_2(\sigma_1(B)) & \text{da } \sigma_1 \text{ Unifikator von } A_i \text{ und } B \text{ ist} \\ & = & \sigma(\sigma_1'(B)) & \text{nach (4.5)} \\ & = & \sigma(B) & \text{da o.B.d.A. der mgu } \sigma_1' \text{ von } A_j \text{ und } D \\ & & & \text{die Variablen von } B \text{ nicht verändert} \end{array}$$

Da $\sigma$ Unifikator von $\sigma_1'(A_i)$ und $B$ ist, gibt es also auch einen mgu $\sigma_2'$ und es existiert eine Substitution $\delta$ mit

$$\sigma = \delta \circ \sigma_2'. \tag{4.6}$$

Damit lässt sich also wie gewünscht auch der zweite Resolutionsschritt mit Hilfe des Literals $\neg A_i$ durchführen.

Es bleibt zu zeigen, dass $\sigma_2 \circ \sigma_1$ und $\sigma_2' \circ \sigma_1'$ bis auf eine Variablenumbenennung gleich sind. Hierzu zeigen wir, dass $\sigma_2 \circ \sigma_1$ eine Instanz von $\sigma_2' \circ \sigma_1'$ und dass $\sigma_2' \circ \sigma_1'$ eine Instanz von $\sigma_2 \circ \sigma_1$ ist.

Dass $\sigma_2 \circ \sigma_1$ eine Instanz von $\sigma_2' \circ \sigma_1'$ ist, folgt sofort aus (4.5) und (4.6), denn

$$\begin{array}{rcll} \sigma_2 \circ \sigma_1 & = & \sigma \circ \sigma_1' & \text{nach (4.5)} \\ & = & \delta \circ \sigma_2' \circ \sigma_1' & \text{nach (4.6)} \end{array}$$

Für die andere Richtung beweisen wir analoge Zusammenhänge zu (4.5) und (4.6), indem wir $\sigma_1$ und $\sigma_2$ mit $\sigma_1'$ und $\sigma_2'$ vertauschen.

Für die Aussage (4.5) hatten wir verwendet, dass $\sigma_2 \circ \sigma_1$ Unifikator von $A_j$ und $D$ ist. Analog dazu verwenden wir jetzt, dass $\sigma_2' \circ \sigma_1'$ auch Unifikator von $A_i$ und $B$ ist. Der Grund ist folgender:

$$\begin{array}{rcll} \sigma_2'(\sigma_1'(A_i)) & = & \sigma_2'(B) & \text{da } \sigma_2' \text{ Unifikator von } \sigma_1'(A_i) \text{ und } B \text{ ist} \\ & = & \sigma_2'(\sigma_1'(B)) & \text{da o.B.d.A. der mgu } \sigma_1' \text{ von } A_j \text{ und } D \\ & & & \text{die Variablen von } B \text{ nicht verändert} \end{array}$$

Da $\sigma_2' \circ \sigma_1'$ also Unifikator von $A_i$ und $B$ ist und da $\sigma_1$ ihr mgu ist, existiert also eine Substitution $\sigma'$ mit

$$\sigma_2' \circ \sigma_1' = \sigma' \circ \sigma_1 \tag{4.7}$$

Für die Aussage (4.6) hatten wir verwendet, dass $\sigma$ Unifikator von $\sigma_1'(A_i)$ und $B$ ist. Analog dazu verwenden wir jetzt, dass $\sigma'$ Unifikator von $\sigma_1(A_j)$ und $D$ ist:

$$\begin{array}{rcll} \sigma'(\sigma_1(A_j)) & = & \sigma_2'(\sigma_1'(A_j)) & \text{nach (4.7)} \\ & = & \sigma_2'(\sigma_1'(D)) & \text{da } \sigma_1' \text{ Unifikator von } A_j \text{ und } D \text{ ist} \\ & = & \sigma'(\sigma_1(D)) & \text{nach (4.7)} \\ & = & \sigma'(D) & \text{da o.B.d.A. der mgu } \sigma_1 \text{ von } A_i \text{ und } B \\ & & & \text{die Variablen von } D \text{ nicht verändert} \end{array}$$

Da $\sigma'$ Unifikator von $\sigma_1(A_j)$ und $D$ ist und da $\sigma_2$ ihr mgu ist, existiert also eine Substitution $\delta'$ mit

$$\sigma' = \delta' \circ \sigma_2. \tag{4.8}$$

Dass $\sigma_2'(\sigma_1'(K))$ eine Instanz von $\sigma_2(\sigma_1(K))$ ist, folgt nun sofort aus (4.7) und (4.8), denn

$$\begin{array}{rcll} \sigma_2' \circ \sigma_1' & = & \sigma' \circ \sigma_1 & \text{nach (4.7)} \\ & = & \delta' \circ \sigma_2 \circ \sigma_1 & \text{nach (4.8)} \end{array}$$

□

**Beispiel 4.3.3** *Um das Vertauschungslemma zu illustrieren, betrachten wir das Programm*

```
p(Z,Z) :- r(Z).
q(W).
```

*und die Anfrage*

```
?- p(X,Y), q(X).
```

*Wenn man erst mit dem* p*-Literal und dann mit dem* q*-Literal resolviert, erhält man z.B.*

$$\begin{array}{rcl} (\{\neg \mathsf{p}(X,Y), \neg \mathsf{q}(X)\}, \varnothing) & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(Z), \neg \mathsf{q}(Z)\}, \{X/Z, Y/Z\}) \\ & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(Z)\}, \{W/Z\} \circ \{X/Z, Y/Z\}) \end{array}$$

*Wenn man hingegen erst mit dem* q*-Literal und dann mit dem* p*-Literal resolviert, ergibt sich z.B.*

$$\begin{array}{rcl} (\{\neg \mathsf{p}(X,Y), \neg \mathsf{q}(X)\}, \varnothing) & \vdash_{\mathcal{P}} & (\{\neg \mathsf{p}(W,Y)\}, \{X/W\}) \\ & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(Y)\}, \{W/Y, Z/Y\} \circ \{X/W\}) \end{array}$$

*Die beiden Resolventen und berechneten Substitutionen sind also bis auf die Variablenumbenennung $\nu = \{Y/Z, Z/Y\}$ gleich.*

Aus dem Vertauschungslemma folgt also, dass man eine beliebige Ordnung der Literale in den Anfragen wählen kann und sich darauf einschränken kann, immer nur das "erste" Literal nach dieser Ordnung zur Resolution zu verwenden. Sofern es überhaupt eine erfolgreiche Berechnung gibt, gibt es dann auch eine Berechnung, die dieser Einschränkung genügt. Wie am Anfang von Abschnitt 4.1 erwähnt, betrachten wir daher Klauseln in der Logikprogrammierung als *Folgen* statt als *Mengen* von Literalen. Man kann nun also eine

beliebige *Selektionsfunktion* wählen, die jeweils aus einer Anfrageklausel das zur Resolution verwendete Literal aussucht. Nun wird damit also auch die Bedeutung der "selection function" in der Abkürzung "SLD" deutlich.

In der Programmiersprache Prolog wird die Selektionsfunktion verwendet, die stets das erste (bzw. das linkeste) Literal in der Anfrageklausel auswählt. Diese Einschränkung der Berechnungsfolgen (mit der Relation $\vdash_{\mathcal{P}}$) bezeichnen wir als *kanonische* Berechnungen.

**Definition 4.3.4 (Kanonische Berechnung)** *Eine Berechnung* $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \ldots$ *eines Logikprogramms* $\mathcal{P}$ *heißt* kanonisch *gdw. bei jedem Resolutionsschritt mit dem ersten Literal der jeweiligen Klausel* $G_i$ *resolviert wird.*

Um zu zeigen, dass man sich in der Tat auf kanonische Berechnungen einschränken kann, benötigen wir noch das folgende Lemma. Es besagt, dass Variablenumbenennungen bei Berechnungen unerheblich sind. Wenn sich also zwei Anfragen nur durch eine Variablenumbenennung unterscheiden, dann kann man mit ihnen dieselben Berechnungen durchführen und erhält Antwortsubstitutionen, die ebenfalls bis auf Variablenumbenennungen gleich sind.

**Lemma 4.3.5 (Variablenumbenennungen bei Berechnungen)** *Sei* $l \in \mathbb{N}$. *Falls* $(G, \sigma) \vdash_{\mathcal{P}}^{l} (G', \sigma')$ *gilt und* $\nu$ *eine Variablenumbenennung ist, so gilt auch* $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}}^{l} (\nu(G'), \nu \circ \sigma')$.

**Beweis.** Wir beweisen das Lemma durch Induktion über $l$. Der Induktionsanfang $l = 0$ ist trivial, da dann $G' = G$ und $\sigma' = \sigma$ gilt.

Im Induktionsschritt $l \geq 1$ sei die Berechnung wie folgt:

$$(G, \sigma) \vdash_{\mathcal{P}} (H, \delta \circ \sigma) \vdash_{\mathcal{P}}^{l-1} (G', \sigma')$$

Nach der Induktionshypothese gilt $(\nu(H), \nu \circ \delta \circ \sigma) \vdash_{\mathcal{P}}^{l-1} (\nu(G'), \nu \circ \sigma')$. Es genügt also, zu zeigen, dass auch $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}} (\nu(H), \nu \circ \delta \circ \sigma)$ gilt.

Wir haben $G = \{\neg A_1, \ldots, \neg A_k\}$ und es existiert eine Programmklausel $K \in \mathcal{P}$ und eine Variablenumbenennung $\omega$ mit $\omega(K) = \{B, \neg C_1, \ldots, \neg C_n\}$, so dass $G$ und $\omega(K)$ keine gemeinsamen Variablen haben. O.B.d.A. können wir davon ausgehen, dass die Variablen in $\omega(K)$ alle frisch sind und somit auch $\nu(G)$ und $\omega(K)$ keine gemeinsamen Variablen haben. Dann ist $\delta$ der mgu von einem $A_i$ und $B$ und es gilt

$$H = \delta(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

Es genügt zu zeigen, dass $\nu \circ \delta \circ \nu^{-1}$ mgu von $\nu(A_i)$ und $B$ ist. Dann ergibt die Resolution von $\nu(G)$ und $\omega(K)$ nämlich die Klausel

$$\begin{array}{ll} & \nu(\delta(\nu^{-1}(\{\nu(\neg A_1), \ldots, \nu(\neg A_{i-1}), \neg C_1, \ldots, \neg C_n, \nu(\neg A_{i+1}), \ldots, \nu(\neg A_k)\}))) \\ = & \nu(\delta(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\})) \\ = & \nu(H). \end{array}$$

Der zweite Schritt gilt, da $\nu^{-1}$ nicht auf den frischen Variablen in $\omega(K)$ operiert und somit $\nu^{-1}(C_j) = C_j$ ist. Man erhält also $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}} (\nu(H), \nu \circ \delta \circ \nu^{-1} \circ \nu \circ \sigma) = (\nu(H), \nu \circ \delta \circ \sigma)$.

Wir zeigen zuerst, dass $\nu \circ \delta \circ \nu^{-1}$ Unifikator von $\nu(A_i)$ und $B$ ist. Es gilt nämlich

$$\begin{array}{lll} & \nu(\delta(\nu^{-1}(\nu(A_i)))) & \\ = & \nu(\delta(A_i)) & \\ = & \nu(\delta(B)) & \text{da } \delta \text{ Unifikator von } A_i \text{ und } B \text{ ist} \\ = & \nu(\delta(\nu^{-1}(B))) & \text{da } \nu^{-1} \text{ nicht auf den frischen Variablen in } \omega(K) \text{ operiert} \end{array}$$

Nun zeigen wir, dass $\nu \circ \delta \circ \nu^{-1}$ auch mgu von $\nu(A_i)$ und $B$ ist. Sei hierzu $\tau$ ein weiterer Unifikator von $\nu(A_i)$ und $B$. Es gilt also $\tau(\nu(A_i)) = \tau(B)$ und da $\nu$ nicht auf den frischen Variablen in $\omega(K)$ operiert und somit $\nu(B) = B$ ist, folgt auch $\tau(\nu(A_i)) = \tau(\nu(B))$. Somit ist also $\tau \circ \nu$ Unifikator von $A_i$ und $B$ und da $\delta$ ihr mgu ist, existiert also eine Substitution $\xi$ mit $\tau \circ \nu = \xi \circ \delta$. Es gilt also auch $\tau \circ \nu \circ \nu^{-1} = \xi \circ \delta \circ \nu^{-1}$. Damit ergibt sich $\tau = \xi \circ \delta \circ \nu^{-1}$ und schließlich $\tau = \xi \circ \nu^{-1} \circ \nu \circ \delta \circ \nu^{-1}$. Daher ist $\tau$ in der Tat eine Instanz von $\nu \circ \delta \circ \nu^{-1}$. □

**Beispiel 4.3.6** *Das obige Lemma illustrieren wir wieder mit dem Programm aus Bsp. 4.3.3. Es gilt*

$$(\{\neg \mathsf{p}(X,Y), \neg \mathsf{q}(X)\}, \sigma) \vdash_{\mathcal{P}} (\{\neg \mathsf{r}(Y), \neg \mathsf{q}(Y)\}, \{X/Y, Z/Y\} \circ \sigma)$$

*für alle Substitutionen $\sigma$. Sei nun $\nu = \{X/Y, Y/U, U/X\}$ eine Variablenumbenennung. Dann gilt*

$$\begin{array}{rcl} (\nu(\{\neg \mathsf{p}(X,Y), \neg \mathsf{q}(X)\}), \nu \circ \sigma) & = & (\{\neg \mathsf{p}(Y,U), \neg \mathsf{q}(Y)\}, \nu \circ \sigma) \\ & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(U), \neg \mathsf{q}(U)\}, \{Y/U, Z/U\} \circ \nu \circ \sigma) \\ & = & (\nu(\{\neg \mathsf{r}(Y), \neg \mathsf{q}(Y)\}), \{Y/U, Z/U\} \circ \nu \circ \sigma). \end{array}$$

*Hierbei gilt*

$$\begin{array}{rcl} \{Y/U, Z/U\} \circ \nu & = & \{Y/U, Z/U\} \circ \{X/Y, Y/U, U/X\} \\ & = & \{X/U, Y/U, Z/U, U/X\}. \end{array}$$

*Dies ist daher die gleiche Substitution wie*

$$\begin{array}{rcl} \nu \circ \{X/Y, Z/Y\} & = & \{X/Y, Y/U, U/X\} \circ \{X/Y, Z/Y\} \\ & = & \{X/U, Y/U, Z/U, U/X\}. \end{array}$$

Nun zeigen wir den gewünschten Satz, der besagt, dass man sich auf kanonische Berechnungen einschränken kann. Jede erfolgreiche Berechnung ist auch dann noch möglich. Somit kann man den Indeterminismus 2. Art also eliminieren. Wir werden uns daher im Folgenden auf kanonische Berechnungen beschränken.

**Satz 4.3.7 (Auflösung des Indeterminismus 2. Art)** *Sei $\mathcal{P}$ ein Logikprogramm und $G$ eine Anfrage. Dann existiert zu jeder Berechnung $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma)$ eine kanonische Berechnung $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma')$ gleicher Länge, wobei sich $\sigma$ und $\sigma'$ nur durch eine Variablenumbenennung unterscheiden.*

**Beweis.** Die Berechnung hat die Gestalt

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1),$$

d.h., die leere Klausel wird in $l$ Schritten berechnet und es gilt $\sigma = \delta_l \circ \ldots \circ \delta_1$. Sei $i$ die Anzahl der Schritte bis zum ersten nicht-kanonischen Berechnungsschritt.

$$(G_i, \delta_i \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}} (G_{i+1}, \delta_{i+1} \circ \ldots \circ \delta_1)$$

mit $i \geq 0$ ist also der erste nicht-kanonische Schritt, d.h., hier wird zum ersten Mal nicht mit dem ersten Literal in der jeweiligen Anfrage $G_i$ resolviert. Wir definieren hierbei $G_0 = G$ und $G_l = \Box$. Falls die Berechnung bereits kanonisch ist, so gilt $i = l$. Weiterhin sei $j$ die Länge der *ersten nicht-kanonischen* Teilberechnungsfolge. Wir durchlaufen die Folge also bis zum ersten nicht-kanonischen Berechnungsschritt von $G_i$ zu $G_{i+1}$. Die Zahl $j$ ist die Anzahl der nun folgenden nicht-kanonischen Berechnungsschritte, bis wieder ein kanonischer Berechnungsschritt folgt. Wir beweisen den Satz durch Induktion über $(l - i, j)$. Hierbei ist $l - i$ die Anzahl der Schritte nach der ersten kanonischen Berechnungsfolge. Als Induktionsrelation verwenden wir die lexikographische Ordnung auf Zahlen. Man darf den Satz also für solche Berechnungen der Länge $l$ voraussetzen, die entweder mehr kanonische Berechnungsschritte am Anfang haben oder die zwar gleich viele solche Schritte haben, aber dafür eine kürzere erste nicht-kanonische Berechnungsfolge.

Falls die Berechnung bereits kanonisch ist, so ist der Satz trivial. Ansonsten existiert tatsächlich ein $i < l$, so dass der Schritt von $G_i$ zu $G_{i+1}$ der erste nicht-kanonische Schritt ist. Dennoch muss es später wieder einen kanonischen Berechnungsschritt geben, denn das erste Literal in der Anfrage muss irgendwann durch Resolution eliminiert werden, um schließlich die leere Klausel $\Box$ zu erhalten. Für die Länge $j$ dieser ersten nicht–kanonischen Berechnungsfolge gilt also $i + j < l$. Wir haben demnach

$$\begin{array}{ll} & (G_i, \delta_i \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{j-1} & (G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}} & (G_{i+j}, \delta_{i+j} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}} & (G_{i+j+1}, \delta_{i+j+1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{l-i-j-1} & (\Box, \sigma). \end{array}$$

Hierbei ist der Schritt von $(G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1)$ zu $(G_{i+j}, \delta_{i+j} \circ \ldots \circ \delta_1)$ nicht-kanonisch und der Schritt von $(G_{i+j}, \delta_{i+j} \circ \ldots \circ \delta_1)$ zu $(G_{i+j+1}, \delta_{i+j+1} \circ \ldots \circ \delta_1)$ ist kanonisch.

Wir wenden nun das Vertauschungslemma 4.3.2 an, um diese beiden Berechnungsschritte zu vertauschen. Dies ist möglich, da wir o.B.d.A. davon ausgehen können, dass die Variablen in den Programmklauseln jeweils so umbenannt werden, dass sie zu den jeweiligen Anfragen variablendisjunkt sind. Nach dem Vertauschungslemma existiert also eine Variablenumbenennung $\nu$ mit

$$(G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}}^{2} (\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \ldots \circ \delta_1).$$

Hierbei bezeichnet $\vdash_{\mathcal{P}}^{2}$ eine Berechnung in zwei Schritten, wobei der erste Schritt jetzt kanonisch ist. Da $(G_{i+j+1}, \delta_{i+j+1} \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}}^{l-i-j-1} (\Box, \sigma)$ eine Berechnung mit $l - i - j - 1$ Schritten ist, ist nach Lemma 4.3.5 auch $(\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}}^{l-i-j-1} (\Box, \nu \circ \sigma)$

eine Berechnung mit $l-i-j-1$ Schritten. Insgesamt erhalten wir also die Berechnung

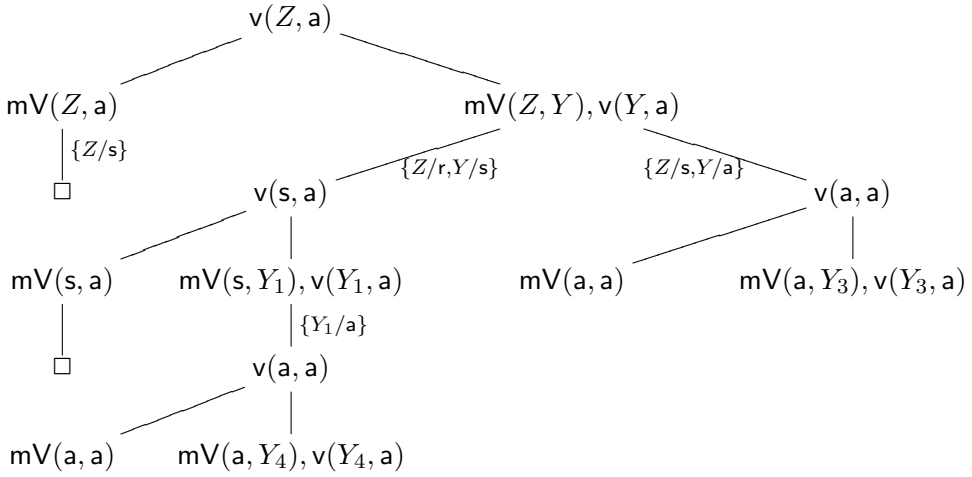
$$\begin{array}{ll} & (G, \varnothing) \\ \vdash_{\mathcal{P}}^{i} & (G_i, \delta_i \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{j-1} & (G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{2} & (\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{l-i-j-1} & (\square, \nu \circ \sigma). \end{array}$$

Falls $j = 1$ ist, so ist die Länge der ersten kanonischen Berechnungsfolge nun mindestens $i+1$ und somit ist die Induktionshypothese anwendbar. Falls $j > 1$ ist, so hat die erste kanonische Berechnungsfolge immer noch die Länge $i$, aber die Länge der ersten Folge nicht-kanonischer Schritte beträgt nur noch $j-1$. Auch dann ist also die Induktionshypothese anwendbar.

Nach der Induktionshypothese gibt es dann auch eine kanonische Berechnung $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma')$, wobei sich $\sigma'$ von $\nu \circ \sigma$ und damit auch von $\sigma$ nur durch eine Variablenumbenennung unterscheidet. □

Eine direkte Folge des obigen Satzes ist, dass die Vollständigkeit der SLD–Resolution erhalten bleibt, auch wenn man sich auf kanonische Berechnungen einschränkt. Dasselbe Resultat gilt für jede beliebige weitere Selektionsfunktion, die jeweils das zur Resolution zu verwendende Literal aus der negativen Klausel auswählt.

**Beispiel 4.3.8** *In Bsp. 4.3.1 können wir uns nun also auf* kanonische *Berechnungen einschränken und dabei sicher sein, dass wir immer noch alle Lösungen finden. Wenn man in dem Baum alle nicht–kanonischen Berechnungen löscht (d.h., wenn man immer nur mit dem ersten Literal der jeweiligen Anfrage resolviert), dann ergibt sich der folgende neue Baum. Diesen Baum bezeichnet man als* SLD–Baum.




*Man erkennt insbesondere, dass der Baum jetzt* endlich *ist. Bei den Anfragen der Art* $\{\neg\mathsf{mutterVon}(\mathsf{aline}, Y_i), \neg\mathsf{vorfahre}(Y_i, \mathsf{aline})\}$ *sind zwar (unendliche) Berechnungen möglich, aber keine weiteren kanonischen Berechnungen, da* $\mathsf{mutterVon}(\mathsf{aline}, Y_i)$ *mit keinem Literal aus Programmklauseln mehr resolviert werden kann.*

Wie das vorige Beispiel schon andeutet, kann der Indeterminismus 2. Art durchaus das Terminierungsverhalten von Logikprogrammen beeinflussen. Dies wird im folgenden Beispiel noch deutlicher.

**Beispiel 4.3.9** *Wir betrachten das folgende Programm.*

```
p :- p
q(a).
```

*Die Anfrage*

```
?- q(b),p.
```

*terminiert in* Prolog*, da es von* $(\{\neg \mathsf{q}(\mathsf{b}), \neg \mathsf{p}\}, \varnothing)$ *aus keine kanonischen Berechnungsschritte gibt. Der SLD–Baum besteht daher nur aus dem einzigen Knoten, der mit der Anfrage markiert ist. Hingegen existiert eine unendliche nicht–kanonische Berechnung*

$$(\{\neg \mathsf{q}(\mathsf{b}), \neg \mathsf{p}\}, \varnothing) \vdash_{\mathcal{P}} (\{\neg \mathsf{q}(\mathsf{b}), \neg \mathsf{p}\}, \varnothing) \vdash_{\mathcal{P}} \ldots$$

*In einer Programmiersprache, die stets das* rechteste *Literal der Anfrage zur Resolution selektiert, würde diese Anfrage also nicht terminieren.*

Die folgende Definition führt den Begriff des *SLD–Baums* formal ein.

**Definition 4.3.10 (SLD–Baum)** *Sei $\mathcal{P}$ ein Logikprogramm und $G$ eine Anfrage. Der* SLD–Baum *von $\mathcal{P}$ bei der Anfrage $G$ ist ein endlicher oder unendlicher Baum, dessen Knoten mit Folgen von atomaren Formeln markiert sind und dessen Kanten mit Substitutionen markiert sind. Der SLD–Baum ist der kleinste Baum, für den folgendes gilt:*

- *Falls $G = \{\neg A_1, \ldots, \neg A_k\}$ ist, so ist die Wurzel des Baums mit $A_1, \ldots, A_k$ markiert.*
- *Sei nun ein Knoten mit $B_1, \ldots, B_n$ markiert und sei $B_1$ mit den positiven Literalen von $k$ Programmklauseln $K_1, \ldots, K_k$ unifizierbar, wobei die Klauseln in dieser Reihenfolge im Programm auftreten. Dann hat der Knoten $k$ Nachfolger. Der $i$-te Nachfolger ist mit den Atomen markiert, die sich bei einem kanonischen Berechnungsschritt durch Resolution mit der Klausel $K_i$ ergeben. Falls diese Berechnung also die Gestalt $(\{\neg B_1, \ldots, \neg B_n\}, \varnothing) \vdash_{\mathcal{P}} (\{\neg C_1, \ldots, \neg C_m\}, \sigma)$ hat, so ist der $i$–te Nachfolgerknoten mit $C_1, \ldots, C_m$ markiert und die Kante zu diesem Knoten ist mit der Substitution $\sigma$ eingeschränkt auf die Variablen in $B_1, \ldots, B_n$ markiert.*[3]

*Hierbei werden Berechnungsschritte, die durch Resolution mit Hilfe des gleichen Literals und der gleichen Programmklausel entstanden sind und sich nur durch Variablenumbenennungen unterscheiden, natürlich nicht unterschieden. Falls $B_1$ also mit den positiven Literalen von $k$ Programmklauseln unifizierbar ist, so hat der Knoten mit der Markierung $B_1, \ldots, B_n$ genau $k$ Nachfolger.*

Die Antwortsubstitutionen lassen sich nun aus den Pfaden ablesen, die mit einem Blatt $\square$ enden. Wenn die Kanten von der Wurzel zu einem $\square$–Blatt mit $\delta_1, \ldots, \delta_l$ beschriftet sind, so ergibt sich die Antwortsubstitution $\delta_l \circ \ldots \circ \delta_1$, eingeschränkt auf die Variablen aus der ursprünglichen Anfrage $G$. Aus Satz 4.3.7 folgt, dass man durch die Betrachtung des

[3] Hierbei können sich die Markierungen natürlich auch durch Variablenumbenennungen unterscheiden.

SLD–Baums immer noch alle Antwortsubstitutionen (bis auf Variablenumbenennungen) erhält.

Neben diesen Pfaden, die erfolgreiche Berechnungen repräsentieren, gibt es auch Pfade von der Wurzel zu Blättern, die nicht die leere Klausel sind. Diese Pfade repräsentieren einen *endlichen Fehlschlag* (finite failure), denn sie enden mit einer Anfrage, deren erstes Literal mit keinem Atom einer Programmklausel resolvierbar ist. Diese Anfrage kann also niemals mehr zu einer erfolgreichen Berechnung führen.

Schließlich kann es auch unendliche Pfade geben, die unendliche Berechnungen repräsentieren.

In der obigen Definition des SLD–Baums haben wir nicht nur den Indeterminismus 2. Art aufgelöst, sondern wir haben zur Behandlung des Indeterminismus 1. Art auch die Reihenfolge der Klauseln im Programm berücksichtigt. Die Reihenfolge der Kinder eines Knotens entspricht also jetzt der Reihenfolge der Klauseln im Programm.[4]

Eine *Auswertungsstrategie* ist nun ein Verfahren, das angibt, wie ein SLD–Baum zu durchlaufen (bzw. aufzubauen) ist. Solch eine Strategie löst dann den Indeterminismus 1. Art auf. Hierbei kann man noch danach unterscheiden, ob man nur an *einer* erfolgreichen Berechnung interessiert ist (dann stoppt man das Verfahren, sobald man einmal $\square$ gefunden hat) oder ob man an *allen* erfolgreichen Berechnungen bzw. Antwortsubstitutionen interessiert ist. In Prolog wird zunächst jeweils nur die erste Lösung berechnet. Wie bereits erwähnt kann man durch anschließende Eingabe von "`;`" danach die Durchsuchung des SLD–Baums aber weiter fortsetzen.

Die Auswertungsstrategie der *Breitensuche* ist eine vollständige Strategie. Diese Strategie berechnet erst alle Knoten der Tiefe 0, dann alle Knoten der Tiefe 1, etc. Daher findet sie auch alle Blätter $\square$ im SLD–Baum und aufgrund der Vollständigkeit der kanonischen SLD–Resolution wird auf diese Weise jede erfolgreiche Berechnung (d.h. jede Antwortsubstitution) irgendwann gefunden. Sofern man nur an *einer* erfolgreichen Berechnung interessiert ist und die ursprüngliche Anfrage aus dem Programm folgt, so terminiert das Verfahren, da man abbricht, sobald das erste Blatt $\square$ gefunden wurde. Sie terminiert hingegen nicht, wenn der Baum unendlich ist und keine leere Klausel enthält. (Es ist also nur semi–entscheidbar, ob der SLD–Baum tatsächlich ein Blatt $\square$ hat.) Falls man an *allen* erfolgreichen Berechnungen interessiert ist, muss man den Baum komplett aufbauen und damit terminiert das Verfahren nur dann, wenn der SLD–Baum endlich ist.

Der Nachteil der Breitensuche ist, dass sie sehr ineffizient ist (sowohl im Zeit– wie im Platzbedarf). Daher verwendet man in der Programmiersprache Prolog stattdessen als Auswertungsstrategie die *Tiefensuche*. Hierbei steigt man rekursiv in die Teilbäume ab. Sofern man nur nach *einer* erfolgreichen Berechnung sucht, findet eine Rückkehr zu den Elternknoten (Backtracking) nur dann statt, wenn man eine (endliche) nicht–erfolgreiche Berechnung gefunden hat (d.h., ein Blatt, das von $\square$ verschieden ist). Aufgrund der möglicherweise unendlichen Pfade im SLD–Baum ist die Strategie der Tiefensuche allerdings unvollständig. Nicht jede erfolgreiche Berechnung wird also durch Tiefensuche gefunden und es kann sogar sein, dass die Tiefensuche gar keine erfolgreiche Berechnung findet, obwohl es eine gibt.

[4] In der Literatur findet man häufig eine alternative Definition des *SLD–Baums*, bei dem die Reihenfolge der Kinder nicht festgelegt ist und somit auch die Reihenfolge der Programmklauseln noch keine Rolle spielt.

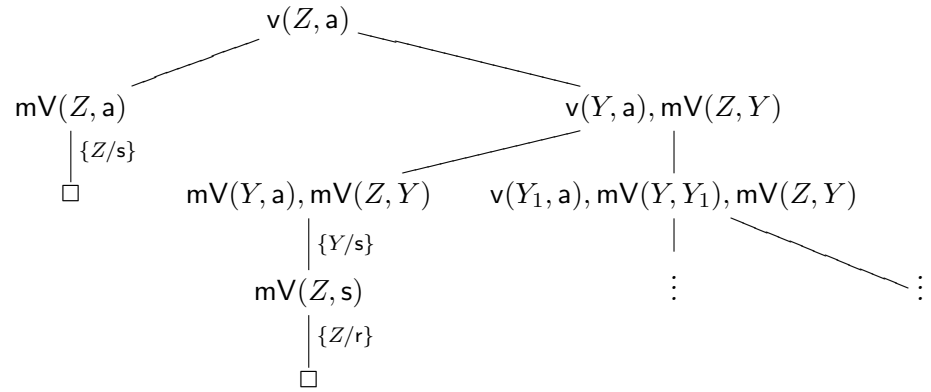
**Beispiel 4.3.11** *Wir betrachten noch einmal das Programm aus Bsp. 4.3.1 bzw. Bsp. 4.3.8. Diesmal ersetzen wir jedoch in den Programmklauseln*

```
vorfahre(V,X) :- mutterVon(V,X).
vorfahre(V,X) :- mutterVon(V,Y), vorfahre(Y,X).
```

*die zweite Klausel durch folgende Modifikation, in der wir die beiden Literale im Rumpf vertauschen.*

```
vorfahre(V,X) :-  vorfahre(Y,X), mutterVon(V,Y).
```

*Es ergibt sich der folgende SLD–Baum:*



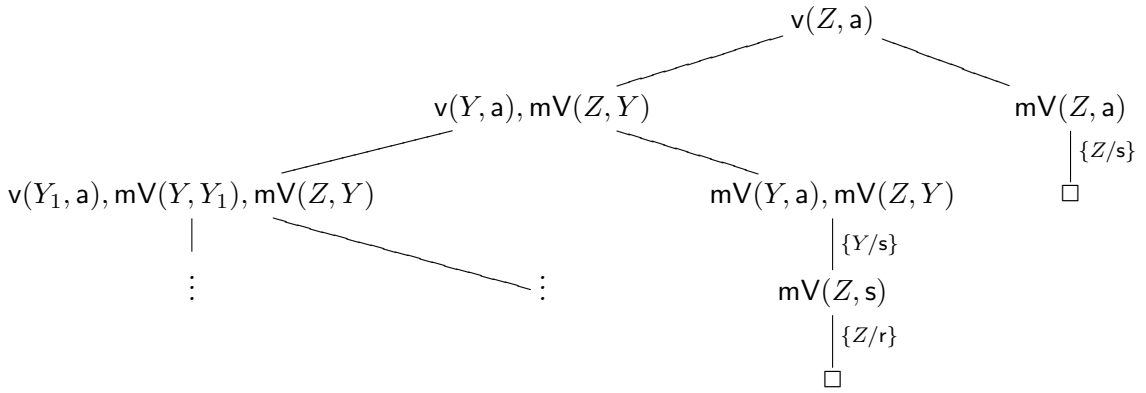

*Der rechteste Pfad in diesem Baum ist nun unendlich. Falls man also eine Tiefensuche wählt, die immer von* rechts *nach* links *vorgeht, dann findet diese Strategie in diesem Beispiel keine Lösung. Die Tiefensuche, die in Prolog gewählt wird, geht allerdings von links nach rechts vor. Sie würde daher die ersten beiden Lösungen finden. Wenn man danach nach einer weiteren Lösung sucht, würde sie nicht mehr terminieren.*

Das obige Beispiel illustriert auch noch einmal den Effekt des Indeterminismus 2. Art auf die Terminierung: Die Vertauschung von Literalen im Rumpf einer Programmklausel kann einen Einfluss auf die Unendlichkeit des SLD–Baums haben und damit auch auf die Terminierung der Auswertung. Das folgende Beispiel zeigt den Einfluss des Indeterminismus 1. Art.

**Beispiel 4.3.12** *Wir setzen das obige Beispiel fort, indem wir nun auch noch die Reihenfolge der Klauseln vertauschen.*

```
vorfahre(V,X) :- vorfahre(Y,X), mutterVon(V,Y).
vorfahre(V,X) :- mutterVon(V,X).
```

*Es ergibt sich somit der folgende SLD–Baum.*




*Nun würde die Tiefensuche in Prolog nicht terminieren und keine Lösung finden.*

Das Beispiel illustriert also, dass man bei der Auswertungsstrategie von Prolog stets Fakten (bzw. nicht–rekursive) Klauseln vor den rekursiven Regeln desselben Prädikats anordnen sollte.

Zusammenfassend ergibt sich also:

- In Prolog werden Literale in Anfragen *von links nach rechts* bearbeitet. Diese Auflösung des Indeterminismus 2. Art beeinflusst zwar das Terminierungsverhalten, aber nicht die Vollständigkeit des SLD–Baums.
- In Prolog werden Programmklauseln *von oben nach unten* abgearbeitet. Dies entspricht einem Durchlauf durch den SLD–Baum in Tiefensuche, wobei linke Kinder stets zuerst betrachtet werden.

# Kapitel 5

# Die Programmiersprache Prolog

Nachdem wir nun die Grundlagen der Logikprogrammierung kennen gelernt haben, wollen wir eine konkrete Programmiersprache betrachten, die auf diesem Prinzip beruht. Die bekannteste logische Programmiersprache ist Prolog. Diese Sprache wurde von Kowalski und Colmerauer in der ersten Hälfte der 70er Jahre entwickelt. Die Akzeptanz als eine der wesentlichen Sprachen der KI kam insbesondere dadurch, dass Prolog als Hauptsprache für das japanische *Fifth Generation Project* 1981 gewählt wurde.

Die Syntax (einfacher) Prolog-Programme entspricht genau der in Def. 4.1.1 festgelegten Syntax von Logikprogrammen. Hierbei wird die bereits bekannte Schreibweise mit "`:-`" für Regeln und mit "`?-`" für Anfragen verwendet. Die Signatur eines Prolog-Programms ergibt sich aus den auftretenden Funktions- und Prädikatssymbolen, die jeweils mit einem Kleinbuchstaben beginnen müssen. Darüber hinaus sind auch Strings aus Sonderzeichen erlaubt (z.B. `<-->`) und Strings in Anführungszeichen, (z.B. `'X'`). Variablen beginnen mit einem Großbuchstaben oder einem Unterstrich. Eine Besonderheit ist die *anonyme* Variable "`_`". Mehrfache Vorkommen dieser Variablen werden als verschieden betrachtet und bei Antwortsubstitutionen werden die Belegungen dieser Variable nicht mit angegeben. Bei einem Programm mit dem Faktum "`p(a,b,c).`" würde die Anfrage "`?- p(_,_,X).`" also die Antwortsubstitution `X = c` liefern.

Prolog erlaubt das Überladen von Funktions- und Prädikatssymbolen. Das obige Programm mit dem Faktum "`p(a,b,c).`" könnte also z.B. um das Faktum "`p(a,b).`" ergänzt werden. Hier ist `p` ein weiteres zweistelliges Prädikatssymbol, das nichts mit dem dreistelligen Prädikatssymbol `p` zu tun hat. Ebenso könnte man das obige Programm auch um das Faktum "`p(p(a,b),c,c)`" ergänzen. Nun ist das innere `p` ein zweistelliges Funktionssymbol, das unabhängig von dem dreistelligen äußeren Prädikatssymbol `p` ist. Da Symbole mit unterschiedlicher Stelligkeit als verschieden betrachtet werden, gibt man ihre Stelligkeit oft nach ihrem Namen mit einem Schrägstrich an (d.h. `p/2` und `p/3`).

Die Semantik von Prolog entspricht der Semantik von Logikprogrammen aus dem vorigen Kapitel. Hierbei wird also der SLD–Baum in Tiefensuche durchlaufen. Bei der ersten gefundenen Antwortsubstitution stoppt Prolog. Gibt der Benutzer daraufhin ein "`;`" ein, so wird der SLD–Baum weiter bis zum nächsten Vorkommen der leeren Klausel durchsucht, etc. Wie im vorigen Kapitel deutlich wurde, ist dies keine ganz reine deklarative Programmierung, bei der der Programmierer nur das Problem, aber nicht das Vorgehen zur Lösung spezifiziert. Der Grund ist, dass sich der Programmierer über Prologs Auswertungsstrategie

im Klaren sein muss und daher z.B. Fakten vor rekursiven Klauseln schreiben sollte, damit Prolog auch tatsächlich Lösungen findet. Die Logik ist damit nicht vollkommen von der Kontrolle zur Ausführung des Programms getrennt.

Ein wichtiger Unterschied zur Semantik der Logikprogramme im vorigen Kapitel ist allerdings, dass die meisten Prolog–Implementierungen bei der Unifikation aus Effizienzgründen keinen *Occur Check* durchführen. Wenn also eine Variable $X$ mit einem Teilterm $t \neq X$ unifiziert werden muss, wird nicht überprüft, ob $X$ in dem Term $t$ auftritt. Stattdessen wird in die Speicherzelle, die der Variable $X$ entspricht, ein Verweis auf die Speicherzelle geschrieben, die dem Term $t$ entspricht. Wenn man also $X$ mit dem Term $\mathtt{f}(X)$ unifizieren will, dann ergibt sich eine Substitution, bei der jedes Vorkommen von $X$ mit dem entsprechend instantiierten Term $\mathtt{f}(X)$ belegt wird. Man erhält also als Unifikator die Substitution $\{X/\mathtt{f}(\mathtt{f}(\mathtt{f}(\ldots)))\}$. Die Variable $X$ wird demnach mit dem *unendlichen* Term aus lauter f-Symbolen belegt. Solch eine Antwortsubstitution erhält man z.B. bei dem Programm mit dem Faktum "`p(X,f(X)).`" und der Anfrage "`?- p(X,X).`". Prolog besitzt daher auch Techniken, um mit unendlichen Termen arbeiten zu können (wobei diese unendlichen Terme durch endliche zyklische Graphen dargestellt werden, d.h., es handelt sich dabei nur um die Teilklasse der sogenannten *rationalen* Terme).

Prolog besitzt zahlreiche *vordefinierte* Prädikate, von denen einige auch im Folgenden vorgestellt werden. Insbesondere gibt es auch ein vordefiniertes Prädikat mit dem Namen `unify_with_occurs_check`, das eine korrekte Unifikation (mit Occur Check) durchführt. Die Anfrage "`?- unify_with_occurs_check(X,f(X)).`" ergibt also die Antwort `false`. Hingegen ergibt die Anfrage "`?- unify_with_occurs_check(X,f(Y)).`" die Antwortsubstitution `X = f(Y)`.

Da Logikprogramme ja nur auf Termen als Daten arbeiten, müssen Datenobjekte als Terme mit Hilfe geeigneter Funktionssymbole dargestellt werden. Für bestimmte Datenstrukturen (insbesondere die ganzen Zahlen und die (linearen) Listen) existieren in Prolog allerdings bestimmte Schreibweisen und Unterstützung, um die Effizienz und Lesbarkeit der Programme zu verbessern. Die Behandlung der Arithmetik und der Listen in Prolog wird in Abschnitt 5.1 und 5.2 dargestellt. Anschließend gehen wir in Abschnitt 5.3 darauf ein, wie der Benutzer neben den eingebauten arithmetischen Operatoren weitere Operatoren selbst definieren kann (d.h. Funktionssymbole mit Infix-, Präfix- oder Postfix-Notation). In Abschnitt 5.4 führen wir das vordefinierte *Cut*-Prädikat ein, das dazu dient, den SLD-Baum zu beschneiden und wir zeigen, wie man dadurch *Negation* implementieren kann. Abschnitt 5.5 erklärt die Ein- und Ausgabebehandlung in Prolog. In Abschnitt 5.6 gehen wir darauf ein, wie man mit Prolog selbst wieder Prolog-Programme manipulieren kann. Schließlich wird in Abschnitt 5.7 gezeigt, wie man in Prolog leicht Parser für kontextfreie Sprachen implementieren kann.

## 5.1 Arithmetik

Wie in Def. 4.2.3 kann man natürliche Zahlen als Terme über den Funktionssymbolen $\mathtt{0} \in \Sigma_0$ und $\mathtt{s} \in \Sigma_1$ darstellen. Einige Logikprogramme auf dieser Datenstruktur wurden bereits in Bsp. 4.2.4 vorgestellt. Beispielsweise hatten wir das folgende Programm zur Berechnung der Addition betrachtet. (Hier verwenden wir das Funktionssymbol `add`, da `plus` in Prolog

bereits vordefiniert ist.)

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

Hierbei steht "`add(X,Y,Z)`" für die Aussage "`X + Y = Z`". Sofern man also eine Anfrage stellt, bei der die ersten beiden Argument von `add` vorgegeben sind, berechnet dieses Programm die Addition. Der Aufruf "`?- add(s(0),s(s(0)),X).`" ergibt also die Antwort `X = s(s(s(0)))`.

Da es in Logikprogrammen jedoch keine festgelegten Ein- und Ausgabeargumente gibt, kann man dieses Programm auch zum Subtrahieren verwenden, indem man z.B. das zweite und das dritte Argument festlegt. Man bezeichnet dieses Verhalten auch als *Bidirektionalität*. Um "3-2" zu berechnen, stellt man die Anfrage "`?- add(X,s(s(0)),s(s(s(0))))`." und erhält die Antwort `X = s(0)`.

Man kann auch zu einer Zahl durch eine Anfrage wie "`?- add(X,Y,s(s(s(0))))`." alle Paare von Summanden berechnen lassen. Die Anfrage "`?- add(X,s(s(0)),Z)`" liefert ebenfalls ein sinnvolles Ergebnis (alle Paare von Werten, so dass der erste Wert um 2 kleiner als der zweite ist, d.h. `Z = s(s(X))`). Hingegen gibt es zu der Anfrage "`?- add(s(0),Y,Z)`" unendlich viele Antwortsubstitutionen. Dies zeigt, dass es zu nahezu jedem Prolog-Programmen auch Anfragen gibt, die zur Nicht–Terminierung bzw. zu einem unendlichen SLD–Baum führen.

Ein Nachteil der Darstellung natürlicher Zahlen als Terme über `0` und `s` ist allerdings, dass dies (aufgrund der fehlenden arithmetischen Grundoperationen und wegen der Größe der Terme) oft zu ineffizienten und schlecht lesbaren Programmen führt. Aus diesem Grund erlaubt Prolog die übliche Schreibweise für ganze Zahlen und stellt einige Grundfunktionen und vordefinierte Prädikate zur Verfügung.

Ein *arithmetischer Ausdruck* ist ein Term, der induktiv aus Zahlen, Variablen und binären Infix–Funktionen wie `+`, `-`, `*`, `//` (Ganzzahldivision), `**` (für die Potenz), etc. sowie der einstelligen Negation `-` aufgebaut ist. Diese Ausdrücke kann man wie bisher als Terme auffassen und mit normaler syntaktischer Unifikation behandeln. Falls man also ein Programm mit dem Faktum "`equal(X,X).`" hat, so führt die Anfrage "`?- equal(3,1+2).`" zum Ergebnis `false`. Die Anfrage "`?- equal(X,1+2).`" ergibt die Lösung `X = 1+2`.

Während bei der normalen Auswertung von Anfragen nur die normale syntaktische Unifikation verwendet wird, gibt es weitere vordefinierte Prädikate, die entsprechende vordefinierte Implementierungen der Funktionen `+`, `-`, `*`, `//`, `**`, etc. verwenden.

Zum *Vergleich* arithmetischer Ausdrücke werden in Prolog unter anderem die zweistelligen Infix–Prädikate *op* mit *op* $\in$ { `<`, `>`, `=<`, `>=`, `=:=`, `=\=` } angeboten. Hierbei stehen die letzten beiden Operatoren für Gleichheit und Ungleichheit. Eine Anfrage "`?-` $t_1$ *op* $t_2$." ist erfolgreich, falls $t_1$ und $t_2$ zum Zeitpunkt der Auswertung voll instantiierte arithmetische Ausdrücke sind (d.h., sie dürfen keine Variablen mehr enthalten) und wenn nach der Auswertung der vordefinierten Infix–Funktionssymbole die Werte $z_1$ und $z_2$ von $t_1$ und $t_2$ in der Relation *op* stehen. Die obigen Prädikatssymbole erzwingen also eine Auswertung. Falls $t_1$ oder $t_2$ kein voll instantiierter arithmetischer Ausdruck ist, so scheitert die Anfrage nicht mit Fehlschlag, sondern sie führt zu einem Programmabbruch. Die folgenden Anfragen haben also folgende Ergebnisse:

- "`?- 1 < 2.`" oder "`?- -2 < -1.`" oder "`?- 1*1 < 1+1.`" führen zum Ergebnis `true`.
- "`?- 2 < 1.`" oder "`?- 6//3 < 5-4.`" führen zum Ergebnis `false`.
- "`?- a < 1.`" oder "`?- X < 1.`" führen zu einem Programmfehler.

Die Forderung, dass zum Zeitpunkt der Ausführung alle Variablen instantiiert sein müssen, bedeutet, dass diese Prädikatssymbole nicht verwendet werden können, um Variablen durch Unifikation zu instantiieren. Eine Anfrage wie "`?- X =:= 2.`" führt nicht zu der Antwortsubstitution `X = 2`, sondern zu einem Programmfehler. Aus diesem Grund gibt es ein weiteres vordefiniertes Prädikatssymbol `is`. Eine Anfrage "`?-` $t_1$ `is` $t_2$." ist erfolgreich, falls $t_2$ zum Zeitpunkt der Auswertung ein voll instantiierter arithmetischer Ausdruck ist, der zu einem Wert $z_2$ auswertet, und wenn $t_1$ mit $z_2$ unifiziert. Falls $t_2$ kein voll instantiierter arithmetischer Ausdruck ist, so scheitert die Anfrage nicht mit Fehlschlag, sondern sie führt zu einem Programmabbruch. Die folgenden Anfragen haben also folgende Ergebnisse:

- "`?- 2 is 1+1.`" oder "`?- 2 is 2.`" führen zum Ergebnis `true`.
- "`?- 1+1 is 2.`" oder "`?- 1+1 is 1+1.`" oder "`?- X+1 is 1+1.`" führen zum Ergebnis `false`.
- "`?- X is 2.`" oder "`?- X is 1+1.`" führen zur Antwortsubstitution `X = 2`.
- "`?- X is 3+4, Y is X+1.`" führt zur Antwortsubstitution `X = 7` und `Y = 8` (denn zum Zeitpunkt der Auswertung ist das `X` in "`Y is X+1`" mit einem voll instantiierten arithmetischen Ausdruck belegt).
- "`?- X is X.`", "`?- 2 is X.`", "`?- X is a.`" oder "`?- Y is X+1, X is 3+4.`" führen zu einem Programmfehler.

Darüber hinaus gibt es auch ein vordefiniertes Prädikatssymbol `=` für die Unifikation beliebiger Terme. Dieses Symbol wird so behandelt, als ob es durch das Faktum "`X = X.`" definiert wäre. Es ist also nicht auf arithmetische Ausdrücke beschränkt. Im Unterschied zu den obigen speziellen Symbolen für die Arithmetik findet hierbei (wie auch bei den anderen selbstdefinierten Prädikatssymbolen) keine Auswertung der Funktionssymbole `+`, `-`, `*`, `//`, `**`, etc. statt. Die folgenden Anfragen haben also folgende Ergebnisse:

- "`?- a = a.`" oder "`?- 2 = 2.`" oder "`?- 1+1 = 1+1.`" führen zum Ergebnis `true`.
- "`?- 2 = 1+1.`" oder "`?- 1+1 = 2.`" führen zum Ergebnis `false`.
- "`?- X+1 = 1+1.`" oder "`?- 1 = X.`" führen zur Antwortsubstitution `X = 1`.
- "`?- X = 1+1.`" führt zur Antwortsubstitution `X = 1+1`.
- "`?- X = X.`" führt zur Antwort `true`, d.h., zur identischen (leeren) Antwortsubstitution.
- "`?- 1+X = Y+1.`" führt zur Antwortsubstitution mit `X = 1` und `Y = 1`.
- "`?- X = 3+4, Y is X+1.`" führt zur Antwortsubstitution `X = 3+4` und `Y = 8`.

Wir haben insofern verschiedene Arten von Gleichheit:

- *Wertgleichheit* "$t_1$ `=:=` $t_2$", wobei $t_1$ und $t_2$ ausgewertet werden und keine Unifikation stattfindet.
- *Wertzuweisung* "$t_1$ `is` $t_2$", wobei $t_2$ ausgewertet wird und danach Unifikation stattfindet.
- *Termgleichheit* "$t_1$ `=` $t_2$", wobei nicht ausgewertet wird, sondern nur Unifikation stattfindet.
- *Syntaktische Gleichheit* "$t_1$ `==` $t_2$", wobei nur untersucht wird, ob $t_1$ und $t_2$ syntaktisch gleich sind.

Beispielsweise führt also `X == X` zum Ergebnis `true`, aber `X == Y` führt zum Ergebnis `false`. Analog dazu führt `f(a,X) == f(a,X)` zum Ergebnis `true`, aber `f(a,X) == f(a,Y)` führt zum Ergebnis `false`.

Das folgende Beispiel zeigt, wie man das Additionsprogramm vom Anfang des Abschnitts entsprechend mit Hilfe der vordefinierten arithmetischen Funktionen programmieren kann. (Alternativ wäre natürlich auch das Programm "`add(X,Y,Z) :- Z is X+Y.`" möglich.)

```
add(X,0,X).
add(X,Y,Z) :- Y > 0, Y1 is Y-1, add(X,Y1,Z1), Z is Z1+1.
```

Wie erwartet, führt die Anfrage "`?- add(1,2,X).`" zur Antwortsubstitution `X = 3`. Der Vorteil der Verwendung der vordefinierten Funktionen und Prädikate auf Zahlen ist die bessere Effizienz und Lesbarkeit. Ein Nachteil ist, dass die Bidirektionalität verloren gehen kann. Die Anfrage "`?- add(X,2,3).`" führt daher zu einem Programmfehler (denn in der Anfrage "`Z is Z1+1`" ist dann `Z1` nicht voll instantiiert).

Das Programm

```
add(X,0,X).
add(X,Y+1,Z+1) :- add(X,Y,Z).
```

würde sich hingegen nicht wie erwartet verhalten. Die Anfrage "`?- add(1,2,X).`" führt zu `false`, da `2` weder mit `0` noch mit `Y+1` unifiziert. Die Anfrage "`?- add(1,0+1,X).`" ergibt die Antwortsubstitution `X = 1+1`.

Als weitere typische Beispiele zeigen wir die Programmierung der Fakultät und des Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

```
fak(0,1).
fak(X,Y) :- X > 0, X1 is X-1, fak(X1,Y1), Y is X*Y1.

ggT(X,0,X).
ggT(0,X,X).
ggT(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X, ggT(X,Y1,Z).
ggT(X,Y,Z) :- Y < X, Y > 0, X1 is X-Y, ggT(X1,Y,Z).
```

Die Anfrage "`?- fak(3,X).`" ergibt die Antwortsubstitution `X = 6` und die Anfrage "`?- ggT(28,36,X).`" ergibt die Antwortsubstitution `X = 4`.

Um die Typen von Termen zu überprüfen, gibt es in Prolog vordefinierte Prädikate wie z.B. `number/1`. Hierbei ist `number(t)` wahr, falls `t` zu diesem Zeitpunkt eine Zahl ist (d.h., die Anfragen "`?- number(2)`" oder "`?- X is 1+1, number(X).`" ergeben `true`, während "`?- number(1+1).`" oder "`?- number(X).`" das Ergebnis `false` hat).

## 5.2 Listen

Um Listen als Terme darzustellen, verwendet man typischerweise ein nullstelliges Funktionssymbol für die leere Liste und ein zweistelliges Funktionssymbol, das das Einfügen eines Elements am Anfang der Liste repräsentiert. Falls man diese Symbole `nil` $\in \Sigma_0$ und `cons` $\in \Sigma_2$ nennt, so ergibt sich z.B. das folgende Prolog-Programm zur Berechnung der Länge einer Liste (das Prädikat `length/2` ist vordefiniert).

```
len(nil,0).
len(cons(X,Xs),Y) :- len(Xs,Y1), Y is Y1+1.
```

Die Anfrage "`?- len(cons(7,cons(3,nil)), X).`" ergibt also die Antwortsubstitution `X = 2`.

Falls man anstelle von `nil` das Funktionssymbol `[]` $\in \Sigma_0$ und anstelle von `cons` das Funktionssymbol `.` $\in \Sigma_2$ nimmt, so unterstützt Prolog dafür lesbarere Kurzschreibweisen. Wie bisher kann man jetzt den folgenden Algorithmus schreiben:

```
len([],0).
len(.(X,Xs),Y) :- len(Xs,Y1), Y is Y1+1.
```

Es sind aber auch die folgenden Kurzschreibweisen möglich:

- `.(`$t_1$`, `$t_2$`)` = `[`$t_1$`|`$t_2$`]`
- `.(`$t_1$`, [])` = `[`$t_1$`]`
- `.(`$t_1$`, .(`$t_2$`, .(`$t_3$`, `$t$`)))` = `[`$t_1$`, `$t_2$`, `$t_3$`|`$t$`]`
- `.(`$t_1$`, .(`$t_2$`, .(`$t_3$`, [])))` = `[`$t_1$`,`$t_2$`,`$t_3$`]` = `[`$t_1$`,`$t_2$`|[`$t_3$`|[]]]` = `[`$t_1$`|[`$t_2$`,`$t_3$`|[]]]` etc.

Diese Kurzschreibweisen werden als *identisch* zu der Schreibweise mit "`.`" und "`[]`" aufgefasst. Beispielsweise gilt:

- Die Anfragen "`?- [1,2] = [1|[2]].`" oder "`?- [1,2] = .(1,[2])`" oder "`?- [1, 2,3] = [1|[2,3|[]]].`" oder "`?- .(1,.(2,[3])) = [1,2,3]`" oder "`?- .(1,2) = [1|2]`" ergeben `true`.
- "`?- .(1,X) = [1,2,3]`" ergibt die Antwortsubstitution `X = [2,3]`.
- "`?- [X,[1|X]] = [[2],Y].`" ergibt die Antwortsubstitution `X = [2], Y = [1,2]`.

Man erkennt, dass "." wirklich ein beliebiges zweistelliges Funktionssymbol ist, das man daher auch zur Darstellung von Binärbäumen verwenden kann. So ist `.(.(1,2),.(3,4))` eigentlich ein Binärbaum, der sich aber auch als `[[1|2]|[3|4]]` darstellen lässt.

Das folgende Beispielprogramm untersucht, ob ein Element in einer Liste enthalten ist.

```
member(X,[X|_]).
member(X,[_|Ys]) :- member(X,Ys).
```

Die Anfrage "`?- member(X,[[a,b],1,[]]).`" liefert die Antwortsubstitution `X = [a,b]`. Falls man durch Eingabe von ";" weitere Lösungen sucht, erhält man noch `X = 1` und `X = []`. (Eine weitere Eingabe von ";" ergibt dann `false`.) Die Anfrage "`?- member(b,X).`" sucht nach allen Listen, die `b` enthalten. Man erhält nacheinander die unendlich vielen Lösungen `X = [b|Xs]` (alle Listen mit dem ersten Element `b`), `X = [Y,b|Xs]` (alle Listen mit dem zweiten Element `b`), etc.

Ein weiteres klassisches Prädikat ist `app` zur Konkatenation von Listen (`append/3` ist vordefiniert).

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

Die Anfrage "`?- app([1,2],[3,4,5],Xs).`" ergibt die Antwortsubstitution `Xs = [1,2,3,4,5]`. Die Anfrage "`?- app(Xs,Ys,[1,2,3]).`" ergibt die Antwortsubstitution `Xs = [], Ys=[1,2,3]` und wiederholte Eingabe von ";" erzeugt die drei weiteren Lösungen. Die Anfrage "`?- app(Xs,[],Zs).`" hat hingegen wieder unendlich viele Lösungen.

## 5.3 Operatoren

Die Standard–Notation für *Terme* und *Atome* in Prolog verwendet eine Präfix-Notation, wobei die Argumente in Klammern angegeben sind. Beispielsweise schreibt man `p(X,f(a))` für die Prädikats- bzw. Funktionssymbole `p/2`, `f/1` und `a/0`. Solche Prädikats- und Funktionssymbole bezeichnet man als *Funktoren*.

Stattdessen ist es auch möglich, zweistellige Prädikats- oder Funktionssymbole in Infix-Schreibweise und einstellige Prädikats- oder Funktionssymbole in Präfix- oder Postfix-Schreibweise (ohne Klammern für die Argumente) zu verwenden. Hierzu müssen die entsprechenden Symbole als *Operatoren* deklariert werden. Der Vorteil hiervon ist eine benutzerfreundlichere Syntax mit besserer Lesbarkeit. Man kommt somit dem Ziel des "Programmierens in natürlicher Sprache" näher.

Beispielsweise ist `+` bereits als Operator vordefiniert. Daher darf man den Term `2+3` schreiben. Dieser wird von Prolog in den Term `+(2,3)` umgewandelt, d.h., die Anfrage "`?- 2+3 = +(2,3).`" ergibt `true`.

Zur Definition von Operatoren verwendet man sogenannte *Direktive* der Form

`:- op(`*Präzedenz*`,` *Typ*`,` *Name(n)*`).`

Es handelt sich also um eine Klausel mit leerem Kopf und dem Systemprädikat `op`. Die in der Direktive deklarierten Operatoren sind dann im Programm hinter der Stelle ihrer

Deklaration verwendbar. Direktive sind Anfragen, die beim Laden des Programms bearbeitet werden, d.h., in dem Moment, in dem die Direktive geladen wird, versucht man, sie mit dem bislang bereits geladenen Programm zu beweisen. Für die Operatoren `+`, `-` und `*` existieren die folgenden vordefinierten Direktiven:

```
:- op(500,yfx,[+,-]).
:- op(400,yfx,*).
```

Im letzten Argument von `op` steht jeweils das Symbol bzw. die Liste von Symbolen, die hier als Operator deklariert werden. Die *Präzedenzen* werden benötigt, um auszudrücken, wie stark welches Funktionssymbol bindet. So bindet `*` stärker als `+` und dies wird dadurch ausgedrückt, dass `*` eine kleinere Präzedenz hat. (Eine *kleinere* Präzedenz bedeutet also eine *stärkere* Bindung.)

Der *Typ* bestimmt die Reihenfolge von Operator und Argument. Hierbei steht `f` für den Operator und `y` und `x` für die Argumente. Für Infix–Symbole gibt es die Typen `xfx`, `yfx` und `xfy`. Für Präfix–Symbole gibt es die Typen `fx` und `fy` und für Postfix–Symbole gibt es die Typen `xf` und `yf`.

Hierbei muss die Präzedenz von `x`–Argumenten *echt kleiner* als die Präzedenz des Operators `f` sein. Die Präzedenz von `y`–Argumenten muss *kleiner oder gleich* der Präzedenz von `f` sein. Die Präzedenz eines Arguments ist die Präzedenz des führenden Operators (und die Präzedenz von Funktoren und von Argumenten in Klammern ist 0).

Beim Typ `yfx` dürfen also nur links vom Operator Argumente mit gleich hoher Präzedenz stehen. Dies bedeutet, dass `1+2+3` als `(1+2)+3` gelesen wird. Somit ergibt die Anfrage "`?- 1+2+3 = (1+2)+3.`" die Antwort `true` und die Anfrage "`?- 1+2+3 = 1+(2+3).`" ergibt `false`. Solche Operatoren sind also *linksassoziativ*. Damit ist auch klar, dass `5-4-3` als `(5-4)-3` gelesen wird. Die Anfrage "`?- X is 5-4-3`" liefert daher die Antwortsubstitution `X = -2`. Analog dazu deklariert `xfy` rechtsassoziative Operatoren und `xfx` deklariert Operatoren ohne Assoziativität.

Der Term `1+2*3+4` muss als `(1+(2*3))+4` gelesen werden. Der Grund ist, dass jeder Operator nur Argumente mit gleicher oder niedrigerer Präzedenz haben darf. Da `*` aber eine niedrigere Präzedenz als `+` hat, können die beiden `+`-Terme nicht die Argumente von `*` sein.

Selbstverständlich dürfen auch Operatoren überladen werden. So gibt es den weiteren vordefinierten einstelligen Operator `-`.

```
:- op(200,fy,-).
```

Der Ausdruck `-2-3` steht somit also für `(-2)-3`.

Das folgende Beispiel zeigt, wie man eigene Operatoren definieren kann, um damit eine einfache Form von Sprachverarbeitung durchzuführen. Wir verwenden das englische Verb "`was`" in Infix–Schreibweise. Dieses sollte keine Assoziativität haben, denn Sätze der Art "`laura was young was beautiful`" machen keinen Sinn. Weiter benutzen wir das Wort "`of`" in Infix–Schreibweise, wobei `of` nach rechts assoziieren soll. Damit steht "`secretary of son of john`" also für "`secretary of (son of john)`". Damit `of` stärker bindet als `was`, sollte `of` eine niedrigere Präzedenz erhalten. Dann steht "`laura was secretary of john`" für "`laura was (secretary of john)`". Schließlich führen wir noch das Wort "`the`" ein. Dies ist ein Präfix–Operator ohne Assoziativität (denn "`the the son`" macht keinen

Sinn). Die Präzedenz sollte niedriger als die Präzedenz von "`of`" sein. Dann steht "`the secretary of the son`" für "`(the secretary) of (the son)`".

Nun können wir das folgende Prolog-Programm schreiben:

```
:- op(300,xfx,was).
:- op(250,xfy,of).
:- op(200,fx,the).

laura was the secretary of the head of the department.
```

Das Faktum "`laura was the secretary of the head of the department.`" steht also für die folgende atomare Formel mit dem Prädikatssymbol `was` und den Funktionssymbolen `of` und `the`.

```
was(laura,of(the(secretary),of(the(head),the(department))))
```

Man kann jetzt die folgenden Anfragen stellen:

```
?-  Who was the secretary of the head of the department.
Who = laura

?- laura was What.
What = the secretary of the head of the department

?- Who was the secretary of the head of What.
Who = laura
What = the department
```

## 5.4 Das Cut-Prädikat und Negation

In Abschnitt 5.4.1 führen wir das Cut–Prädikat ein, das es ermöglicht, den Backtracking–Mechanismus von Prolog zu kontrollieren. Hiermit ist es insbesondere möglich, Meta–Prädikate wie die *Negation* zu programmieren, wie in Abschnitt 5.4.2 gezeigt wird.

### 5.4.1 Das Cut–Prädikat

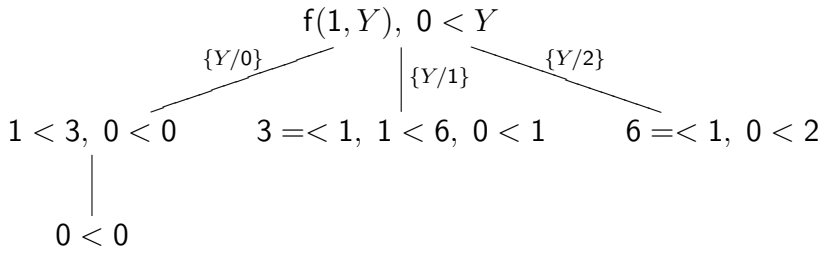
Prolog führt automatisch ein Backtracking durch, wenn es ein Blatt des SLD–Baums erreicht hat, bei dem ein endlicher Fehlschlag erkannt wurde (d.h., ein Blatt, das nicht die leere Klausel $\square$ ist). Während dies oftmals von Vorteil ist, kann es auch Fälle geben, in denen man dies vermeiden will. Der Grund ist, dass das automatische Backtracking bei Fehlschlag sowohl zeit- als auch speicheraufwändig ist, denn alle Knoten mit Wahlmöglichkeiten müssen beim Beweis gespeichert werden. Ein weiterer Grund ist, dass das Backtracking und Durchlaufen aller Zweige zu unnötiger Nicht–Terminierung führen kann, falls manche Zweige unendlich sind.

Betrachten wir hierzu das folgende einfache Programm zur Berechnung der Funktion $f$ mit

$$f(x) = \begin{cases} 0, \text{falls } x < 3 \\ 1, \text{falls } 3 \leq x < 6 \\ 2, \text{falls } 6 \leq x \end{cases}$$

```
f(X,0) :- X<3.
f(X,1) :- 3=<X, X<6.
f(X,2) :- 6=<X.
```

Wir stellen nun die Anfrage "`?- f(1,Y), 0<Y.`". Diese führt zum folgenden SLD–Baum.




Die Anfrage scheitert also und das Ergebnis ist `false`. Hierzu wird zuerst der linkeste Pfad des SLD–Baums durchlaufen. Hier gelingt der Beweis des ersten Beweisziels `X<3`, da `X` mit `1` belegt ist, aber das zweite Beweisziel `0<Y` scheitert aufgrund der Belegung von `Y` mit `0`. Anschließend wird zurückgesetzt und es werden die anderen beiden Pfade durchlaufen, die der zweiten und dritten Programm–Klausel entsprechen.

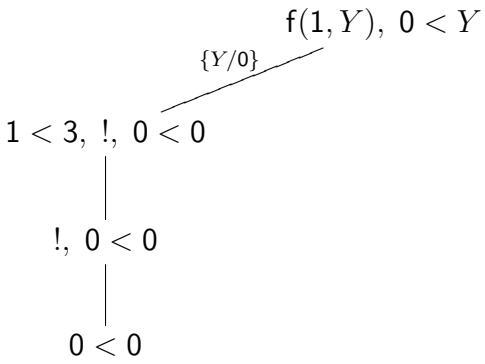
Man erkennt aber, dass sich die Bedingungen "`X<3`" bzw. "`3=<X, X<6`" bzw. "`6=<X`" der drei `f`–Klauseln gegenseitig ausschließen. Sobald also der Beweis einer dieser drei Bedingungen gelingt, muss man die anderen `f`–Klauseln gar nicht mehr betrachten. Da bei unserer Anfrage bereits die Bedingung `X<3` der ersten `f`–Klausel beweisbar ist, braucht man also nicht mehr zurücksetzen und die anderen beiden `f`–Klauseln betrachten, da von vornherein klar ist, dass ihre Bedingungen nicht zutreffen können. Wir wollen daher den mittleren und rechten Pfad des SLD–Baums *abschneiden*.

Um dies durchzuführen, bietet Prolog das sogenannte *Cut*–Prädikat an, das durch "`!`" dargestellt wird. Dieses nullstellige Prädikatssymbol kann auf rechten Seiten von Regeln und in Anfragen auftreten. Es ist immer beweisbar, aber es hat den Effekt, dass es alternative Pfade im SLD–Baum abschneidet. Um dies zu illustrieren, modifizieren wir unser Programm nun wie folgt:

```
f(X,0) :- X<3, !.
f(X,1) :- 3=<X, X<6, !.
f(X,2) :- 6=<X.
```

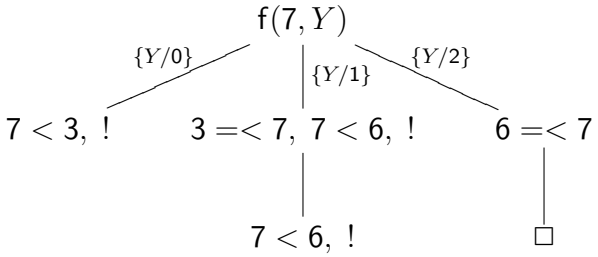
Der Effekt des Cuts in der ersten `f`–Klausel ist, dass man bei einer Anfrage `f(...)` nach der Resolution mit der ersten `f`–Klausel und dem Gelingen des Beweisziels `X<3` nicht mehr zurücksetzen kann, um alternative Beweise für `X<3` oder für `f(...)` zu versuchen. Somit wird ein Beweisversuch mit der zweiten oder dritten `f`–Klausel abgeschnitten. Der

entstehende SLD–Baum für unsere obige Anfrage sieht wie folgt aus:




Die Menge der obigen Cuts bezeichnet man auch als *grüne* Cuts, da sie nur die Effizienz, aber nicht die Ergebnisse oder das Terminierungsverhalten des Programms beeinflussen. Wenn man die Cuts weglässt, ergeben sich immer noch die gleichen Lösungen.

Die Cuts im obigen Programm haben den Effekt, dass man nach dem erfolgreichen Beweis der Bedingungen in der ersten oder zweiten `f`–Klausel nie versucht, weitere `f`–Klauseln anzuwenden. Dies ermöglicht eine weitere Verbesserung des obigen Programms, um die Effizienz weiter zu steigern. Hierzu betrachten wir den SLD–Baum für die Anfrage "`?- f(7,Y).`".




Nachdem im ersten Pfad das Beweisziel `X<3` mit der Belegung von `X` mit `7` gescheitert ist, wird im zweiten Ziel noch einmal das entsprechende negierte Beweisziel `3=<X` aufgerufen. Dies ist aber unnötig, denn wenn `X<3` scheitert, muss der Beweis von `3=<X` gelingen. Analog dazu scheitert der Beweis von `X<6` bei der Belegung von `X` mit `7` im mittleren Pfad. Dann ist es aber unnötig, im rechtesten Pfad noch einmal `6<=X` zu beweisen, da dies dann gelingen muss. Aufgrund der Verwendung von Cuts kann man also das Programm wie folgt verändern:

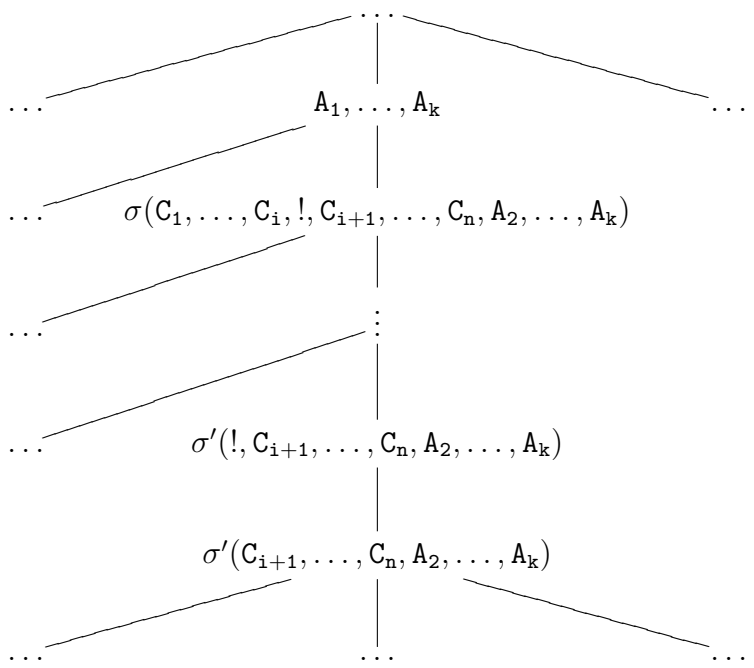
```
f(X,0) :- X<3, !.
f(X,1) :- X<6, !.
f(X,2).
```

Diese Cuts sind *rote* Cuts, da man jetzt andere Lösungen erhält, wenn man sie weglässt. Beispielsweise hat die Anfrage "`?- f(1,Y)`" dann nicht nur die Lösung `Y = 0`, sondern auch die Lösungen `Y = 1` und `Y = 2`.

Man muss auch beachten, dass man bei der Einführung von Cuts meist eine bestimmte Verwendung des Prädikats (mit Ein- und Ausgabepositionen) im Sinn hat. Falls man im obigen Prädikat nicht die zweite Position als Ausgabeposition betrachtet, können nun nämlich ungewünschte Effekte entstehen. So ergibt die Anfrage `?- f(0,2)` nämlich `true`,

obwohl die gewünschte Funktion ja $f(0) = 0$ liefert. Ebenso ergibt die Anfrage "`?- p(X).`" bei dem Programm mit den Klauseln "`p(0) :- !.`" und "`p(1) :- !.`" nur die Antwort `X = 0`, obwohl eigentlich auch die Lösung `X = 1` möglich gewesen wäre.

Die genaue Bedeutung des Cuts ist wie folgt. Falls eine Anfrage "`?-` $\mathtt{A}_1,\ldots,\mathtt{A}_k$" mit einer Programmklausel "`B :-` $\mathtt{C}_1,\ldots,\mathtt{C}_i$`, !,` $\mathtt{C}_{i+1},\ldots,\mathtt{C}_n$" resolviert wird und anschließend die entsprechend instantiierten Teilziele $\mathtt{C}_1, \ldots, \mathtt{C}_i$ bewiesen werden, so entsteht der folgende SLD–Baum:



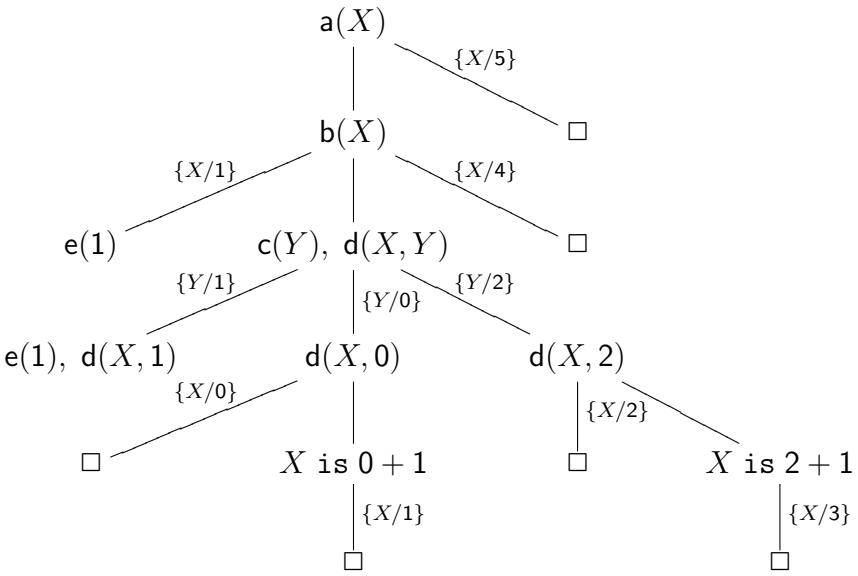

Der Cut bedeutet also, dass bei allen Knoten zwischen dem Knoten mit der Markierung "$\mathtt{A}_1,\ldots,\mathtt{A}_k$" und dem Knoten mit der Markierung "$\sigma'$`(!,`$\mathtt{C}_{i+1},\ldots,\mathtt{C}_n,\mathtt{A}_2,\ldots,\mathtt{A}_k$`)`" keine Alternativen (auf der rechten Seite) betrachtet werden. Hingegen werden sowohl oberhalb als auch unterhalb dieser Knoten Alternativen untersucht. Falls hingegen der Beweisversuch vor Erreichen des Cuts scheitert (d.h. wenn die instantiierten Beweisziele $\mathtt{C}_1, \ldots, \mathtt{C}_i$ nicht alle beweisbar sind), dann werden nach wie vor Alternativen versucht.

Dies wird durch folgendes Beispiel illustriert.

```
a(X) :- b(X).
a(5).

b(1) :- e(1).
b(X) :- c(Y), d(X,Y).
b(4).

c(1) :- e(1).
c(0).
c(2).

d(X,X).
```

```
d(X,Y) :- X is Y+1.

e(0).
```

Bei der Anfrage "`?- a(X).`" erhält man den folgenden SLD–Baum.

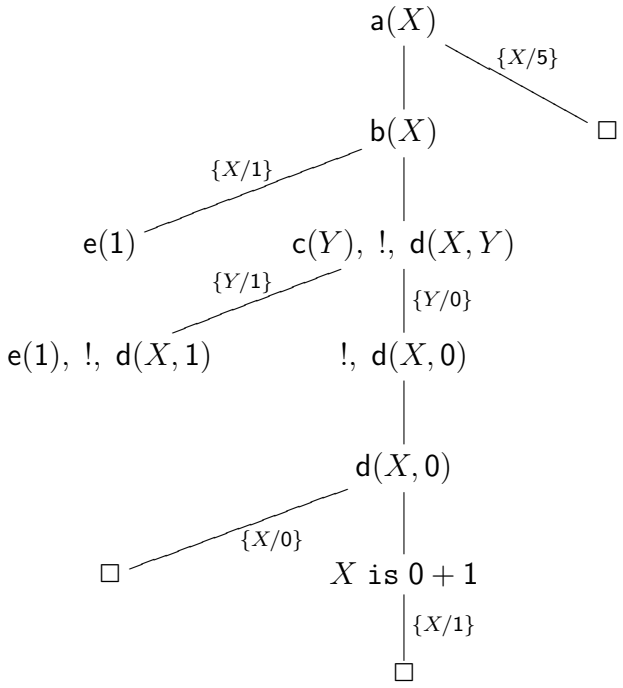



Bei wiederholter Eingabe von "`;`" werden also folgende Lösungen ausgegeben: `X = 0`, `X = 1`, `X = 2`, `X = 3`, `X = 4`, `X = 5`.

Nun ändern wir die zweite `b`–Klausel durch Einfügen eines Cuts zu:

```
b(X) :- c(Y), !, d(X,Y).
```

Der Effekt ist, dass nach wie vor die Alternativen für `a` und `d` betrachtet werden, aber nicht die Alternativen für `b` und `c` (nach dem ersten Gelingen). Es ergibt sich der folgende SLD–Baum:



Bei wiederholter Eingabe von ";" werden also folgende Lösungen ausgegeben: `X = 0`, `X = 1`, `X = 5`.

Als Beispiel für die Verwendung des Cuts betrachten wir unser ggT-Programm von Abschnitt 5.1.

```
ggT(X,0,X).
ggT(0,X,X).
ggT(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X, ggT(X,Y1,Z).
ggT(X,Y,Z) :- Y < X, Y > 0, X1 is X-Y, ggT(X1,Y,Z).
```

Falls eine der ersten beiden Klauseln mit der Anfrage unifiziert, sollte man keine weiteren Beweisversuche mit anderen `ggT`–Klauseln durchführen. Daher führen wir hier jeweils einen Cut ein. Die stellt sicher, dass man die unteren beiden Klauseln nur erreicht, wenn `X` und `Y` beide größer als 0 sind (falls wir uns bei `X` und `Y` auf natürliche Zahlen einschränken). Damit kann man also die Literale `X > 0` und `Y > 0` weglassen. Schließlich führen wir in der vorletzten Klausel einen Cut hinter dem Literal `X =< Y` ein. Dies stellt sicher, dass wir die letzte Klausel nur erreichen, wenn `Y < X` ist. Somit kann man in der letzten Klausel dieses Literal weglassen.

```
ggT(X,0,X) :- !.
ggT(0,X,X) :- !.
ggT(X,Y,Z) :- X =< Y, !, Y1 is Y-X, ggT(X,Y1,Z).
ggT(X,Y,Z) :- X1 is X-Y, ggT(X1,Y,Z).
```

Schließlich präsentieren wir noch ein natürliches Beispiel zur Verwendung des Cuts. Das Prädikat `remove(X,Xs,Ys)` trifft zu, falls die Liste `Ys` aus der Liste `Xs` entsteht, indem alle Vorkommen von `X` aus `Xs` gelöscht werden.

```
remove(_,[],[]).
remove(X,[X|Xs],Ys) :- !,remove(X,Xs,Ys).
remove(X,[Y|Xs],[Y|Ys]) :- remove(X,Xs,Ys).
```

Die Anfrage "`?- remove(1,[0,1,2,1],Ys).`" hat dann nur die Lösung `Ys = [0,2]`. Ohne den Cut gäbe es noch die weiteren Lösungen `Ys = [0,2,1]`, `Ys = [0,1,2]` und `Ys = [0,1,2,1]`.

### 5.4.2 Meta–Variablen und Negation

Prolog erlaubt die Verwendung von *Meta–Variablen*. Dies sind Variablen, die für *Formeln* statt für *Terme* stehen. Die Unifikation kann nun auch solche Variablen instantiieren. Ebenso haben wir Meta-Prädikate, die als Argumente selbst wieder *Formeln* statt nur *Terme* haben können.

Als einfaches Beispiel betrachten wir das Programm

```
p(a).
a.
```

Hierbei ist `a` ein nullstelliges *Prädikatssymbol*, d.h., das Prädikatssymbol `p` hat als Argument eine Formel und keinen Term. Bei der Anfrage "`?- p(X), X.`" ist `X` eine Meta-Variable, die mit einer Formel instantiiert wird. Die Anfrage gelingt hier mit der Antwortsubstitution `X = a`. Eine uninstantiierte Meta-Variable allein darf nicht zur Resolution verwendet werden (sie muss hierzu stets vorher instantiiert worden sein). Eine Anfrage wie "`?- p(X), X, Y.`" führt daher zu einem Programmfehler.

Ein weiteres Beispiel für die Verwendung von Meta-Variablen ist das folgende Programm:

```
or(X,Y) :- X.
or(X,Y) :- Y.
```

Dieses Prädikat ist in Prolog unter dem Namen ";" vordefiniert. Hierbei ist ";" ein Infix-Operator mit der Direktive `:- op(1100,xfy,;)`.[1] Die Anfrage "`?- X = 4 ; X = 5.`" ergibt also die Antwortsubstitution `X = 4` und erneute Eingabe von ";" führt zur Antwortsubstitution `X = 5`.

Durch die Verwendung des Cuts kann man insbesondere Meta–Prädikate programmieren, die bestehende Prädikate negieren oder auf aussagenlogische Weise verknüpfen. Das folgende Programm dient z.B. zur Realisierung von "`if A then B else C`".

```
if(A,B,C) :- A, !, B.
if(A,B,C) :- C.
```

Der Cut ist hierbei nötig, um weitere Beweise zu verhindern, wenn `A` wahr ist und `B` fehlschlägt. Sonst würde `if(A,B,C)` immer beweisbar sein, wenn `C` beweisbar ist. In Prolog ist ein ähnliches Prädikat vordefiniert. Statt `if(A,B,C)` kann man auch "`A -> B;C`" schreiben.

Ein weiteres wichtiges Hilfsprädikat ist die *Negation*. Bisher können wir aus einem Logikprogramm nur positive Aussagen ableiten (d.h., existenzquantifizierte Aussagen der Form $A_1 \wedge \ldots \wedge A_k$). Unser Ziel ist, jetzt auch Konjunktionen herzuleiten, die negierte atomare Formeln $\neg A$ enthalten dürfen. Bei der Realisierung der Negation in Prolog werden folgende Annahmen getroffen:

- Aus dem Programm sind alle wahren Aussagen über die Welt herleitbar (*Closed World Assumption*). Falls also eine Aussage $A$ nicht herleitbar ist, dann ist sie auch nicht wahr und somit ist $\neg A$ wahr.

- Falls eine Aussage aus dem Programm nicht herleitbar ist, dann wird das in endlicher Zeit festgestellt.

Damit wird die Negation als *endlicher Fehlschlag* interpretiert (*Negation as Failure*). Um $\neg A$ zu beweisen, versucht man also, stattdessen $A$ zu beweisen. Falls der SLD–Baum zu $A$ endlich ist und keinen erfolgreichen Ast besitzt, dann wird dadurch $\neg A$ bewiesen. Die Negation lässt sich wie folgt mit Hilfe des Cuts implementieren:

[1] Der Operator "," für die Konjunktion hat die Präzedenz 1000, d.h., er bindet stärker. Somit ergibt die Anfrage "`?- p(X,Y).`" im Programm mit der Klausel "`p(X,Y) :- X = 1, Y = 1; X = 2, Y = 2.`" die Antwortsubstitutionen `X = 1, Y = 1` und `X = 2, Y = 2`.

```
not(A) :- A, !, fail.
not(A).
```

Hierbei ist `fail` ein vordefiniertes Prädikat, das stets fehlschlägt. Der Cut ist wieder nötig, da sonst `not(A)` stets wahr wäre. Das Prädikat `not` ist in Prolog vordefiniert (und es kann auch als Präfix–Operator `\+` geschrieben werden).

Ein Beispiel für die Verwendung der Negation ist die Definition der Ungleichheit:

```
not_equal(X,Y) :- not(X=Y).
```

Wie erwartet führt die Anfrage "`?- not_equal(1,2)`" zum Ergebnis `true` und die Anfrage "`?- not_equal(1,1)`" ergibt `false`. Die Anfrage "`?-X=2, not_equal(1,X)`" ergibt `X = 2`, aber "`?- not_equal(1,X), X=2`" ergibt `false`. Der Grund ist, dass die Negation den Existenzquantor in einen Allquantor verwandelt. Die Anfrage `not_equal(1,X)` ist also die Frage, ob *alle* `X` verschieden von `1` sind. Dies ist natürlich nicht der Fall. Wenn hingegen vorher `X` mit `2` instantiiert wird, dann trifft diese Aussage hingegen zu.

Die Negation in Prolog entspricht nicht unbedingt der intuitiven Bedeutung der Negation. So wird nicht jeder Fehlschlag bemerkt, da unendliche erfolglose SLD–Bäume nicht als erfolglos erkannt werden. (Das Problem ist, dass es nicht entscheidbar ist, ob die leere Klausel durch SLD–Resolution herleitbar ist.) Hierzu betrachten wir das folgende Programm:

```
even(0).
even(X) :- X1 is X-2, even(X1).
```

Zwar ist `even(1)` nicht beweisbar, aber der Beweisbaum ist unendlich. Somit terminiert weder die Anfrage "`?- even(1).`" noch die Anfrage "`?- not(even(1)).`".

In der Version

```
even(0).
even(X) :- X >= 2, X1 is X-2, even(X1).
```

terminiert `even(t)` zwar für jede Zahl `t`, aber da die Anfrage "`?- even(-2).`" nicht beweisbar ist, kann man "`?- not(even(-2)).`" beweisen. Die Closed World Assumption muss also nicht unbedingt zutreffen (dies ist ein Problem der vollständigen Modellierung des Anwendungsbereichs).

Eine alternative Version, die auf allen ganzen Zahlen korrekt arbeitet, ist

```
even(0) :- !.
even(X) :- X > 0, !, X1 is X-1, not(even(X1)).
even(X) :- X1 is X+1, not(even(X1)).
```

## 5.5 Ein- und Ausgabe

Bislang waren Anfragen die einzige Möglichkeit, um *Eingaben* an ein Programm zu übergeben. Eine *Ausgabe* von Programmen war nur durch die Antwortsubstitutionen des Programms bzw. durch die Ausgabe `true` oder `false` möglich. Prolog verfügt aber auch über extra-logische Prädikate, um Seiteneffekte mit Ein- und Ausgabe durchzuführen.

Das Prädikatssymbol `write/1` dient dazu, einen Term in den aktuellen Ausgabe–Stream zu schreiben. Dieser ist standardmäßig der Bildschirm des Benutzers. Die Anfrage `write(t)` gelingt also für jeden Term `t` und gibt als Seiteneffekt `t` (bzw. eine variablenumbenannte Variante von `t`) aus. Beispielsweise wird bei der Anfrage "`?- X is 2+3, write(X).`" als Seiteneffekt `5` ausgegeben und die Antwortsubstitution lautet `X = 5`. Bei der Anfrage "`?- write('Dies ist eine Konstante').`" wird die Konstante "`Dies ist eine Konstante`" als Seiteneffekt ausgegeben.

Bei dem Programm

```
mult(X,Y) :- Ergebnis is X*Y, write(X*Y), write(' = '), write(Ergebnis).
```

erhält man also folgendes Ergebnis:

```
?- mult(3,4).
3*4 = 12.
```

Hierbei muss man darauf achten, dass beim Backtracking die Seiteneffekte eines `write`-Literals nicht mehr rückgängig gemacht werden können. Im Programm

```
q(a).
q(b).
p :- q(X), write(X), X = b.
```

ergibt sich also

```
?- p.
ab
```

Ein weiteres vordefiniertes Prädikat ist `nl/0` (für **n**ew**l**ine), das einen Zeilenvorschub im aktuellen Ausgabefile bewirkt. Es ergibt sich

```
?- write(a),nl,write(b),nl,write(c).
a
b
c
```

Darüber hinaus existieren zahlreiche weitere Prädikate zur Ausgabe und zur Ausgabeformatierung.

Zur Eingabe existiert ein vordefiniertes Prädikat `read/1`. Hierbei liest `read(t)` einen Term `s` von der Eingabe und versucht dann `t` und `s` zu unifizieren. Falls `t` und `s` nicht unifizierbar sind, schlägt das Beweisziel `read(t)` fehl. Um das Ende des Terms `s` zu markieren, muss dieser mit einem "`.`" enden.

Hierzu betrachten wir das folgende Programm:

```
sqr(X,Y) :- Y is X*X.

sqr :- nl,
       write('Bitte geben Sie eine Zahl oder "stop" ein: '),
       read(X),
```

```
       proc(X).

proc(stop) :- !.
proc(X) :- sqr(X,Y),
           write('Das Quadrat von '),
           write(X),
           write(' ist '),
           write(Y),
           sqr.
```

Um das Programm auszuführen, stellt man die Anfrage `?- sqr.` Ein möglicher Programmlauf sieht dann wie folgt aus.

```
?- sqr.

Bitte geben Sie eine Zahl oder "stop" ein: 3.
Das Quadrat von 3 ist 9
Bitte geben Sie eine Zahl oder "stop" ein: -4.
Das Quadrat von -4 ist 16
Bitte geben Sie eine Zahl oder "stop" ein: stop.
```

Es ist auch möglich, Ein– und Ausgabe mit Files durchzuführen. Hierzu muss man den momentanen Ein– bzw. Ausgabe–Stream auf die jeweiligen Files setzen. Hierzu dienen die Prädikate `see/1` und `tell/1`. Die Anfrage `see(t)` setzt den Eingabe–Stream auf das File mit dem Namen `t`. Analog dazu arbeitet `tell(t)`. Die Prädikate `seen` und `told` schließen den Ein– bzw. Ausgabe–Stream und setzen den aktuellen Stream wieder zurück auf `user`.[2] Man kann nun das obige Programm wie folgt ändern:

```
sqr(X,Y) :- Y is X*X.

start  :- nl,
          write('Bitte geben Sie den Namen eines Eingabefiles ein: '),
          read(Eingabefile),
          write('Bitte geben Sie den Namen eines Ausgabefiles ein: '),
          read(Ausgabefile),
          see(Eingabefile),
          tell(Ausgabefile),
          sqr,
          seen,
          told.

sqr :- read(X),
       proc(X).

proc(end_of_file) :- !.
```

[2]Darüber hinaus existieren weitere ähnliche Prädikate wie `open` und `close`.

```
proc(X) :- sqr(X,Y),
           write('Das Quadrat von '),
           write(X),
           write(' ist '),
           write(Y),
           nl,
           sqr.
```

Das Programm liest nun Terme im Eingabe–File. Wenn das Ende des Files erreicht ist, liefert `read(X)` die Belegung `X = end_of_file`. Falls `eingabe` das File mit dem Inhalt

```
3. -4.
```

ist, dann ist folgender Programmlauf möglich:

```
?- start.

Bitte geben Sie den Namen eines Eingabefiles ein: eingabe.
Bitte geben Sie den Namen eines Ausgabefiles ein: ausgabe.
```

Danach steht im File `ausgabe` der Inhalt:

```
Das Quadrat von 3 ist 9
Das Quadrat von -4 ist 16
```

## 5.6 Meta–Programmierung

In diesem Abschnitt führen wir mehrere vordefinierte Prädikate von Prolog ein, die Terme manipulieren können (Abschnitt 5.6.1) und die sogar das momentane Programm "während der Laufzeit" verändern können (Abschnitt 5.6.2).

### 5.6.1 Verarbeitung von Termen und atomaren Formeln

In Abschnitt 5.1 hatten wir bereits das vordefinierte Prädikat `number/1` kennen gelernt, mit dem man überprüfen kann, ob ein Objekt eine Zahl ist. Prolog bietet darüber hinaus noch zahlreiche weitere Prädikate an, um verschiedene Arten von Termen und atomaren Formeln zu erkennen:

- `var(t)` ist wahr, falls `t` eine nicht-instantiierte Variable ist. Beispielsweise ergibt die Anfrage "`?- var(X).`" die Antwort `true`. Die Anfrage "`?- X = 2, var(X).`" ergibt hingegen `false`.
- `nonvar(t)` ist wahr, falls `t` keine Variable ist. Die Anfrage "`?- nonvar(a).`" ergibt daher `true` und "`?- X = 2, nonvar(X).`" ergibt die Antwortsubstitution `X = 2`. Die Anfrage "`?- nonvar(X).`" ergibt `false`, obwohl es natürlich "Antwortsubstitutionen" für `X` gäbe, bei denen die Instantiierung von `X` keine Variable ist.

- `atomic(t)` ist wahr, falls `t` ein nullstelliges Prädikats- oder Funktionssymbol oder eine Zahl ist.[3] Beispielsweise führen die Anfragen "`?- atomic(a).`", "`?- atomic(-).`", "`?- atomic(2).`" und "`?- atomic(-2).`" zu `true` und "`?- atomic(X).`" und "`?- atomic(a(a)).`" zu `false`.

- `compound(t)` ist wahr, falls `t` ein Term oder eine atomare Formel ist, die nicht nur aus einem nullstelligen Funktions- oder Prädikatssymbol oder einer Zahl besteht. So ergeben die Anfragen "`?- compound(a).`", "`?- compound(X).`", "`?- compound(2).`" und "`?- compound(-2).`" die Antwort `false` und "`?- compound(1+2).`" und "`?- compound(a(a)).`" ergeben `true`.

Neben den obigen Prädikaten zur *Erkennung* bestimmter Arten von Termen gibt es in Prolog auch drei Prädikate, um Bestandteile von Termen zu extrahieren und um neue Terme zu konstruieren. Die Idee ist hierbei, dass man Terme bzw. atomare Formeln wie `f(a,b)` auch als *Listen* schreiben könnte. Das erste Element der Liste ist das äußerste Funktions- bzw. Prädikatssymbol `f` und die verbleibenden Elemente der Liste sind die Argumente dieses Symbols. Zu `f(a,b)` ergibt sich dann also die Liste `[f,a,b]`. Hierbei ist das erste Element der Liste ein *nullstelliges* Symbol `f/0` wohingegen das äußere Symbol des Terms `f(a,b)` ein *zweistelliges Symbol* `f/2` ist.

In Prolog existiert hierzu ein vordefiniertes zweistelliges Prädikatssymbol "`=..`", wobei "`t =.. l`" wahr ist, falls `l` die Listendarstellung des Terms bzw. der atomaren Formel `t` ist. Bei der Anfrage "`?- f(a,b) =.. L.`" erhält man also die Antwortsubstitution `L = [f,a,b]` und bei "`?- 1+2 =.. L.`" erhält man `L = [+,1,2]`. Analog dazu erhält man bei "`?- T =.. [f,a,b].`" die Antwort `T = f(a,b)` und bei "`?- T =.. [f].`" die Antwort `T = f`. (Anfragen wie "`?- X =.. Y.`", "`?- X =.. [Y,a,b].`" oder "`?- X =.. [f|L].`", bei denen der Funktor oder die Länge der Argumentliste nicht feststehen, sind hingegen nicht erlaubt und führen zu einem Programmabbruch.)

Die Überführung in Listendarstellung geschieht bei "`=..`" nur auf der obersten Ebene. Die Anfrage "`?- p(f(X),2,g(X,Y)) =.. L.`" ergibt also `L = [p, f(X), 2, g(X,Y)]`.

Als Beispiel betrachten wir ein Programm zur Vergrößerung geometrischer Figuren. Für verschiedene Arten von Figuren seien verschiedene Funktionssymbole verwendet worden, z.B.:

`square(Side)`
`rectangle(Side1, Side2)`
`triangle(Side1, Side2, Side3)`
`circle(Radius)`
⋮

Unser Ziel ist die Programmierung eines Prädikats `enlarge/3`, wobei `enlarge(Fig, Factor, NewFig)` gelten soll, falls `NewFig` durch Vergrößerung der Figur `Fig` um den Faktor `Factor` entsteht. Die naive Lösung bestände darin, für jeden Figurentyp eine eigene `enlarge`–Klausel zu programmieren:

[3] Wenn man Zahlen ausschließen will, kann man stattdessen das Prädikat `atom/1` verwenden.

```
enlarge(square(Side),
        Factor,
        square(NewSide)) :- NewSide is Factor*Side.

enlarge(rectangle(Side1, Side2),
        Factor,
        rectangle(NewSide1, NewSide2)) :- NewSide1 is Factor*Side1,
                                          NewSide2 is Factor*Side2.

enlarge(triangle(Side1, Side2, Side3),
        Factor,
        triangle(NewS1, NewS2, NewS3)) :- NewS1 is Factor*Side1,
                                          NewS2 is Factor*Side2,
                                          NewS3 is Factor*Side3.

enlarge(circle(Radius),
        Factor,
        circle(NewRadius)) :- NewRadius is Factor*Radius.
...
```

Der Nachteil ist ein unnötig großer Programmieraufwand. Die Klauseln sind ja sehr analog, denn jedes Mal werden einfach nur die Parameter der Figur mit dem Faktor `Factor` multipliziert. Darüber hinaus hat diese Lösung das Problem, das hierzu bereits alle jemals vorkommenden Figurentypen bekannt sein müssen.

Durch Verwendung des Prädikats "`=..`" kann man dieses Programm deutlich eleganter formulieren, indem die Terme jeweils dekomponiert und entsprechend neue Terme konstruiert werden:

```
enlarge(Fig,Factor,NewFig) :- Fig =.. [Type | Param],
                              multiplylist(Param, Factor, NewParam),
                              NewFig =.. [Type | NewParam].

multiplylist([],_,[]).
multiplylist([X|L],Factor,[NewX|NewL]) :- NewX is Factor*X,
                                          multiplylist(L,Factor,NewL).
```

Hier steht die Variable `Type` für das äußere Funktionssymbol von `Fig`, d.h., für den Typ der Figur.

Neben dem vordefinierten Prädikat "`=..`" existieren noch die Prädikate `functor/3` und `arg/3`, um auf das führende Funktionssymbol und die Argumente eines Terms bzw. einer atomaren Formel zuzugreifen. Hierbei ist `functor(t,f,n)` erfüllt, falls `f` das führende Symbol des Terms (bzw. der atomaren Formel) `t` ist und wenn `n` die Stelligkeit von `f` ist. Die Anfrage "`?- functor(g(f(X),X,g), F, N).`" ergibt also `F = g` und `N = 3`. Die Anfrage "`?- functor(T,g,3).`" erzeugt hingegen `T = g(X,Y,Z)`.

Die Aussage `arg(n,t,a)` ist wahr, falls `a` das `n`-te Argument des Terms `t` ist, wobei die Nummerierung der Argumente mit 1 beginnt. Die Anfrage "`?- arg(3, g(f(X),X,g),`

`A).`" ergibt also `A = g`. Als weiteres Beispiel betrachten wir die folgende Anfrage zur Konstruktion eines Terms mit Hilfe von `functor` und `arg`:

```
?- functor(D,date,3),
   arg(1,D,29),
   arg(2,D,june),
   arg(3,D,1982).
```

Die Antwortsubstitution ist `D = date(29,june,1982)`.

Als abschließendes Beispiel definieren wir ein Prädikat `grund/1`, wobei `grund(t)` wahr ist, falls `t` keine Variablen enthält:

```
grund(T) :- nonvar(T),
            T =.. [Functor|Argumentlist],
            grundlist(Argumentlist).

grundlist([]).
grundlist([T|Ts]) :- grund(T), grundlist(Ts).
```

### 5.6.2 Verarbeitung von Programmen

Ein Prolog-Programm kann als Datenbank aufgefasst werden. Man kann den Inhalt der Datenbank (d.h., den Programmtext) während der Laufzeit lesen, indem man das vordefinierte Prädikat `clause/2` verwendet. Die Anfrage "`?- clause(`$t_1, t_2$`)`" ist wahr, falls es eine Programmklausel `B :-` $C_1, \ldots, C_k$ gibt, so dass `clause(`$t_1, t_2$`)` und `clause(B, (`$C_1, \ldots, C_k$`))` unifizieren. Wir betrachten hierzu das folgende Programm:

```
times(_,0,0).
times(X,Y,Z) :- Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X.
```

Die Anfrage "`?- clause(times(X,Y,Z),Body).`" ergibt die Antwortsubstitution `Y = 0`, `Z = 0` und `Body = true`. Hierbei ist `true/0` ein vordefiniertes Prädikat, das stets wahr ist. Gibt man nun ";" ein, so erhält man die zweite Antwortsubstitution `Body = Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X`. (Man beachte, dass "," ein zweistelliges Prädikatssymbol ist, das auch in Infix–Schreibweise verwendet werden kann.)

Die Programm–Datenbank aus Fakten und Regeln wurde bislang als "statisch" aufgefasst. Es ist aber möglich, diese Datenbank (d.h. das Prolog-Programm) *während seiner Ausführung* zu verändern. Hierzu dienen die Prädikate `assert/1` und `retract/1`.

Der Beweis von `assert(t)` gelingt stets und als Seiteneffekt wird *die Klausel* `t` ans Ende des Programms *hinzugefügt*. Wenn man also z.B. die Anfrage "`?- assert(p(0)).`" stellt, so wird das bisherige Programm um das Faktum `p(0)` ergänzt. Stellt man nun die Anfrage "`?- p(X).`", so erhält man die Antwortsubstitution `X = 0`. Stellt man darüber hinaus die Anfrage "`?- assert(square(X,Y) :- times(X,X,Y)).`", so wird das Programm auch noch um die entsprechende Regel ergänzt. Die Anfrage "`?- square(3,Y).`" ergibt daher `Y = 9`. Neben dem Prädikatssymbol `assert` existiert auch noch das Prädikat `asserta/1`, das eine Klausel am Anfang des Programm einfügt.

Eine solche Veränderung von Programmklauseln ist nur bei solchen Prädikatssymbolen möglich, die als *dynamisch* vereinbart wurden. Dies ist automatisch für alle Prädikate wie `p` und `square` der Fall, die durch `assert` eingeführt wurden. Wenn man dem bisherigen Programm aber auch noch Klauseln mit `times/3` im Kopf hinzufügen will, so muss `times/3` im Programm als *dynamisch* deklariert werden. Hierzu dient das Prädikatssymbol `dynamic/1`, das als Argument den Namen des Prädikats, einen Schrägstrich und die Stelligkeit des Prädikats bekommt. Das Prädikat `dynamic` ist auch als Präfix–Operator deklariert, so dass man keine Klammern um das Argument schreiben muss. Die `dynamic`–Deklaration muss als Direktive in das Programm vor der Verwendung des jeweiligen Prädikatssymbols geschrieben werden. Wir modifizieren daher das obige `times`–Programm wie folgt:

```
:- dynamic times/3.

times(_,0,0).
times(X,Y,Z) :- Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X.
```

Nun kann man z.B. das Programm während der Laufzeit um das Faktum `times(X,1,X)` ergänzen. Man stellt dann z.B. die Anfrage "`?- asserta(times(X,1,X)).`". Wenn man nun "`?- clause(times(X,Y,Z),Body).`" fragt, so ergibt sich als erste Antwortsubstitution `X = Z`, `Y = 1` und `Body = true`.

Man kann während der Laufzeit nicht nur das Programm um neue Klauseln ergänzen, sondern auch Klauseln aus dem Programm löschen. Hierzu dient das Prädikat `retract/1`. Der Beweis von `retract(t)` gelingt dann, wenn es eine Programmklausel gibt, die mit `t` unifiziert. In diesem Fall wird die erste solche Programmklausel gelöscht.

Wenn man also im obigen Programm, das um die Klausel `times(X,1,X)` am Anfang ergänzt wurde, die Anfrage "`?- retract(times(X,Y,X) :- Body).`" stellt, so wird die erste unifizierende Programmklausel "`times(X,1,X)`" gelöscht. Gibt man daraufhin "`;`" ein, so wird auch noch die zweite Programmklausel "`times(_,0,0)`" gelöscht. Bei einer erneuten Eingabe von "`;`" kann auch noch die letzte Programmklausel für `times` gelöscht werden. Man hätte auch die Anfrage "`?- retract(times(X,Y,Z)).`" stellen können. Diese kann allerdings nur die beiden ersten `times`-Klauseln löschen, da hier der Klauselrumpf `true` sein muss.

Eine sinnvolle Verwendung von Prädikaten wie `assert` besteht darin, Berechnungsergebnisse zu speichern, so dass man später schneller wieder auf sie zugreifen kann. Ein Beispiel ist das folgende Programm, das eine Tabelle speichert, die alle Produkte `X * Y` für Zahlen `X` und `Y` zwischen 0 und 9 erzeugt.

```
maketable :- L = [0,1,2,3,4,5,6,7,8,9],
             member(X,L),
             member(Y,L),
             Z is X * Y,
             assert(times(X,Y,Z)),
             fail.
```

Die Anfrage "`?- maketable.`" schlägt zwar fehl, aber durch das Backtracking werden dem Programm 100 Fakten hinzugefügt, in denen bereits alle Multiplikationen von Zahlen zwischen 0 und 9 gespeichert sind. Die Anfrage "`?- times(X,Y,8).`" führt dann zu 4 möglichen Antwortsubstitutionen: `X = 1` und `Y = 8`, `X = 2` und `Y = 4`, etc.

Generell lässt sich aber sagen, dass die Prädikate `assert` und `retract` sehr zurückhaltend verwendet werden sollten, da dies zu einem sehr schlechten Programmierstil mit vollkommen unverständlichen Programmen führen kann.

Schließlich wollen wir zeigen, wie man mit Hilfe von `assert` ein Prädikat `findall/3` programmieren kann, das nicht nur nach der ersten Lösung für eine Anfrage sucht und dann auf eine Benutzereingabe wartet, sondern selbst alle Lösungen sucht und diese dann als Liste zurückliefert. Dieses Prädikat ist in Prolog vordefiniert.

Genauer soll `findall(t,g,l)` wahr sein, falls folgendes gilt. Man versucht, die Anfrage `g` zu lösen und berechnet *alle* möglichen Lösungen (d.h., man baut den SLD–Baum komplett auf). Bei jeder Lösung wird die gefundene Antwortsubstitution $\sigma$ genommen und $\sigma$`(t)` in die Liste eingefügt. Damit ist `findall(t,g,l)` also wahr, falls `l` die Liste [$\sigma_1$`(t),...,` $\sigma_n$`(t)]` ist, wobei $\sigma_1, \ldots, \sigma_n$ die Antwortsubstitutionen sind, die bei Tiefensuche von links nach rechts in dieser Reihenfolge im SLD–Baum gefunden werden. Als Beispiel betrachten wir das folgende Programm:

```
mutterVon(monika, karin).
mutterVon(monika, klaus).
mutterVon(renate, susanne).
mutterVon(renate, peter).
mutterVon(susanne, aline).
mutterVon(susanne, dominique).

verheiratet(werner, monika).
verheiratet(gerd, renate).
verheiratet(klaus, susanne).

vaterVon(V,K) :- verheiratet(V,F), mutterVon(F,K).
```

Die Anfrage "`?- findall(K, vaterVon(gerd,K), L).`" bewirkt, dass `L` mit der Liste *aller* Kinder von `gerd` instantiiert wird. Man erhält als Antwortsubstitution also `L = [susanne, peter]`. Bei der Anfrage "`?- findall(vaterVon(gerd,K), vaterVon(gerd,K), L).`" erhält man stattdessen `L = [vaterVon(gerd, susanne), vaterVon(gerd, peter)]`.

Um `findall` selbst zu programmieren, kann man z.B. das folgende Programm verwenden:

```
findall(X, Anfrage, Xlist) :- Anfrage,
                              assert(loesung(X)),
                              fail
                              ;
                              sammleLoesungen(Xlist).

sammleLoesungen([X | Rest]) :- retract(loesung(X)),
                               !,
                               sammleLoesungen(Rest).
sammleLoesungen([]).
```

In der Klausel für `findall` wird zunächst die Anfrage gelöst und die erste gefundene Lösung in die Datenbank geschrieben (als Argument des einstelligen Prädikatssymbols `loesung`). Anschließend schlägt die Gesamtanfrage fehl, so dass Rücksetzen erzwungen wird. Auf diese Weise wird also der gesamte SLD–Baum für die Anfrage aufgebaut und alle Lösungen nacheinander gespeichert. Schließlich wird `sammleLoesungen` aufgerufen. Man versucht solange wie möglich, Lösungen aus der Datenbank zu löschen und in die Ergebnisliste einzufügen. Der Cut hinter `retract(loesung(X))` ist nötig, damit man bei der Suche nach weiteren Lösungen nicht zurücksetzt und noch einmal eine neue Lösung berechnet.

Da Prolog–Programme selbst wieder als Terme aufgefasst werden können (durch Verwendung des Prädikatssymbols `clause`), eignet sich Prolog auch zum Schreiben von *Meta–Programmen*. Dies sind Programme wie Compiler oder Interpreter, die selbst wieder andere Programme verarbeiten. Insbesondere kann man auch *Meta–Interpreter* schreiben, d.h., Interpreter für eine Programmiersprache, die selbst in dieser Programmiersprache geschrieben sind. Man kann also in Prolog einen Prolog–Interpreter schreiben. Prolog eignet sich auch besonders für das *Rapid Prototyping*, um z.B. prototypische Interpreter für neue Programmiersprachen oder für neue Auswertungsstrategien von Programmiersprachen zu schreiben.

Wir betrachten nun mehrere Meta–Interpreter für Prolog, die gegenüber dem normalen Prolog–Interpreter erweiterte Funktionalität aufweisen. Der einfachste Meta–Interpreter lässt sich wie folgt implementieren:

```
prove(Goal) :- Goal.
```

Dieser Meta–Interpreter ist allerdings ziemlich nutzlos, da er die gesamte "Arbeit" an den originalen Prolog–Interpreter delegiert.

Ein alternativer Meta–Interpreter ist folgender. Er eignet sich allerdings nur für reine Logikprogramme (ohne vordefinierte Prädikate). Dafür kann er als Ausgangspunkt genommen werden, um modifizierte Meta–Interpreter zu entwickeln.

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Die Cuts sind hierbei nötig, um unerwünschte Effekte beim Rücksetzen zu vermeiden. Ohne die Cuts würde z.B. beim Programm

```
p(0).
```

und der Anfrage "`?- prove(p(X)).`" nach der ersten Lösung `X = 0` ein Programmfehler auftreten, da man nun die dritte Klausel verwendet und somit das Beweisziel `clause(true, Body)` lösen muss. Für das vordefinierte Symbol `true` darf `clause` aber nicht aufgerufen werden. Würde man nur den ersten Cut verwenden, so würde die Anfrage "`?- prove((p(X), p(X))).`" zu unendlich vielen Lösungen `X = 0` führen, da das Komma über die Klausel "`X, Y :- X, Y`" definiert ist.

Nun kann man z.B. eine Variante des Meta–Interpreters schreiben, die Literale in Anfragen nicht von links nach rechts, sondern von rechts nach links abarbeitet:

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal2), prove(Goal1).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Eine weitere Variante wäre ein Meta–Interpreter, der jeweils die Länge des Beweises mit ausgibt:

```
prove(true,0) :- !.
prove((Goal1,Goal2),N) :- !, prove(Goal1,N1), prove(Goal2,N2), N is N1+N2.
prove(Goal,N) :- clause(Goal, Body), prove(Body,N1), N is N1+1.
```

Zahlreiche andere Varianten sind denkbar. Ein Beispiel wäre ein Meta–Interpreter, der die Terminierung erzwingt, indem er nur Beweise durchführt, die kürzer als eine vorgegebene Länge sind. Eine andere Möglichkeit wäre ein Meta–Interpreter, der den Beweisbaum mit ausgibt, etc.

## 5.7 Differenzlisten und definite Klauselgrammatiken

Prolog bietet eine eigene Notation für Regeln von kontextfreien Grammatiken, sogenannte *definite Klauselgrammatiken*. Um diese möglichst effizient zu implementieren, wird eine spezielle Repräsentation für Listen (sogenannte *Differenzlisten*) verwendet. Wir stellen zunächst Differenzlisten in Abschnitt 5.7.1 vor und führen danach definite Klauselgrammatiken in Abschnitt 5.7.2 ein.

### 5.7.1 Differenzlisten

Viele Operationen auf Listen lassen sich deutlich effizienter programmieren, wenn man eine andere Listendarstellung (sogenannte *Differenzlisten*) wählt. Hierzu betrachten wir noch einmal die bisherige Realisierung des Prädikats `app/3` zur Listenkonkatenation von Abschnitt 5.2.

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

Der Aufwand zur Konkatenation von zwei Listen ist dabei $O(n)$, wobei $n$ die Länge der ersten Liste ist. Beispielsweise benötigt der Beweis der Anfrage "`?- app([1,2,3], [4,5], Zs).`" vier Resolutionsschritte, um die Antwortsubstitution `Zs = [1,2,3,4,5]` zu berechnen. Der Grund ist, dass die erste Liste jeweils elementweise durchlaufen wird, bis die leere Liste erreicht wird.

Unser Ziel ist nun eine alternative Listendarstellung, bei der die Listenkonkatenation in nur einem Berechnungsschritt durchgeführt werden kann. Hierzu repräsentieren wir Listen jetzt als *Differenzlisten*, d.h., als die Differenz zweier Listen. Die Liste `[1,2,3]` kann also z.B. durch die beiden Listen `[1,2,3,4,5]` und `[4,5]` repräsentiert werden, denn es gilt

```
[1,2,3,4,5] - [4,5] = [1,2,3].
```

Bei dieser Repräsentation muss natürlich die zweite Liste (d.h. `[4,5]`) jeweils ein Endstück der ersten Liste (d.h. `[1,2,3,4,5]`) sein. Selbstverständlich ist diese Repräsentation nicht eindeutig. Die Liste `[1,2,3]` kann auch als die Differenz von `[1,2,3]` und `[]` oder als die Differenz von `[1,2,3,4,5 | Ys]` und `[4,5 | Ys]` oder als die Differenz von `[1,2,3`

`| Ys]` und `Ys` dargestellt werden. Diese letzte Darstellung ist die allgemeinste Darstellung von Differenzlisten.

Nun kann man folgende alternative Implementierung von `app` definieren, die nur noch aus einem einzigen Faktum besteht:

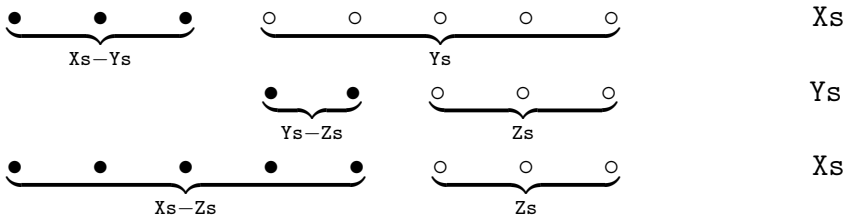
```
app(Xs - Ys, Ys, Xs).
```

Betrachten wir nun eine Anfrage "`?- app(`$t_1$`, `$t_2$`, Xs)`". Falls das erste Argument $t_1$ von `app` also eine Liste in allgemeinster Differenzlisten–Darstellung wie "`[1,2,3 | Ys] - Ys`" ist, so wird in einem einzigen Resolutionsschritt `Ys` mit dem zweiten `app`-Argument $t_2$ instantiiert und es ergibt sich als Ergebnis die Antwortsubstitution `Xs = [1,2,3 |` $t_2$`]`. Die Anfrage "`?- app([1,2,3 | Ys] - Ys, [4,5], Xs)`" führt also nach nur einem einzigen Resolutionsschritt zur Antwortsubstitution `Ys = [4,5]` und `Xs = [1,2,3,4,5]`. Der Aufwand ist also jetzt nicht mehr linear, sondern konstant (d.h. $O(1)$).

Ein Nachteil des obigen Algorithmus ist aber, dass nur das erste Argument von `app` als Differenzliste dargestellt ist. Das Ergebnis von `app` kann daher nicht wieder direkt als Eingabe für `app` im ersten Argument verwendet werden. Daher formulieren wir nun eine Version von `app`, bei der alle drei Argumente Differenzlisten sind.

```
app(Xs - Ys, Ys - Zs, Xs - Zs).
```

Das folgende Schaubild verdeutlicht die Arbeitsweise des obigen Programms:




Wenn man jeweils die allgemeinsten Darstellungen von Differenzlisten verwendet, so berechnet dieses Programm wieder die Listenkonkatenation in konstanter Zeit und liefert die resultierende Liste ebenfalls in allgemeinster Differenzlistendarstellung. Die Anfrage "`?- app([1,2,3 | Ys] - Ys, [4,5 | Zs] - Zs, Erg).`" ergibt also in einem Schritt die Antwortsubstitution `Ys = [4,5 | Zs]`, `Erg = [1,2,3,4,5 | Zs] - Zs`.

Hierbei ist zu beachten, dass das obige Programm nur dann die Listen konkatenieren kann, wenn die Differenzlistendarstellung so gewählt wurde, dass die beiden ersten `app`-Argumente "kompatibel" sind. Eine Anfrage "`?- app(`$t_1$` - `$t_2$`, `$s_1$` - `$s_2$`, Erg).`" scheitert, falls $t_2$ und $s_1$ nicht unifizierbar sind. So ergibt also die Anfrage "`?- app([1,2,3,6] - [6], [4,5] - [], Erg).`" die Antwort `false`. Ebenso führt auch die folgende Anfrage zu `false`:

```
?- L = [1|Ys] - Ys,
   app(L, [2 | Zs] - Zs, Erg1),
   app(L, [3 | Ws] - Ws, Erg2).
```

Der Grund ist, dass nach Lösung des zweiten Atoms in der Anfrage die Variable `L` mit `[1,2 | Zs] - [2 | Zs]` und `Ys` mit `[2 | Zs]` instantiiert ist. Die Terme `[2 | Zs]` und `[3 | Ws]` sind aber nicht unifizierbar.

Als ein weiteres Beispiel betrachten wir einen Algorithmus `flatten/2` zur Umwandlung geschachtelter in lineare Listen. (Dieses Prädikat ist in Prolog vordefiniert.) Hierbei ist `append/3` das normale (und vordefinierte) Prädikat zur Konkatenation von Listen in der klassischen Darstellung (d.h., in der Darstellung ohne Verwendung von Differenzlisten).

```
flatten([X|Xs],Ys) :- !,
                      flatten(X, X1),
                      flatten(Xs,Xs1),
                      append(X1,Xs1,Ys).
flatten([],[]) :- !.
flatten(X,[X]).
```

Beispielsweise ergibt die Anfrage "`?- flatten([[1,2],[[3]] | [[4,[5,6]]]],Ys).`" die Antwort `Ys = [1,2,3,4,5,6]`. Der Aufwand zur Berechnung von "`?- flatten(t,Ys)`" für eine Liste $t$ mit $n$ Elementen ist $O(n^2)$, da `append` jeweils linearen Aufwand hat.

Um den Aufwand zu reduzieren, sollte man daher die Version von `append` verwenden, die auf Differenzlisten basiert.

```
flatten([X|Xs],Ys - Zs) :- !,
                           flatten(X, X1 - X2),
                           flatten(Xs, Xs1 - Xs2),
                           app(X1 - X2, Xs1 - Xs2, Ys - Zs).
flatten([],Ys - Ys) :- !.
flatten(X,[X|Ys] - Ys).
```

Die Anfrage "`?- flatten([[1,2],[[3]] | [[4,[5,6]]]],Ys).`" ergibt nun die Antwort `Ys = [1,2,3,4,5,6|Ws] - Ws`. Da app-Anfragen nun mit konstantem Aufwand gelöst werden können, ergibt sich als Gesamtaufwand nur noch $O(n)$. Falls man auch als Ausgabe keine Differenzliste haben möchte, so kann man die zusätzliche Klausel

```
flat(Xs,Ys) :- flatten(Xs, Ys - []).
```

verwenden. Die Anfrage "`flat([[1,2],[[3]] | [[4,[5,6]]]],Ys).`" ergibt die Antwort `Ys = [1,2,3,4,5,6]`.

Man erkennt, dass man das obige `flatten`–Programm noch vereinfachen kann. Das Atom `app(X1 - X2, Xs1 - Xs2, Ys - Zs)` im Rumpf der ersten Klausel gelingt genau dann, wenn `X2` und `Xs1` unifizierbar sind und wenn sowohl `X1` und `Ys` sowie `Xs2` und `Zs` unifizierbar sind. Wir können daher direkt `X2` durch `Xs1` ersetzen, `X1` durch `Ys` ersetzen und `Xs2` durch `Zs` ersetzen. So ergibt sich das folgende Programm:

```
flatten([X|Xs],Ys - Zs) :- !,
                           flatten(X, Ys - Xs1),
                           flatten(Xs, Xs1 - Zs).
flatten([],Ys - Ys) :- !.
flatten(X,[X|Ys] - Ys).
```

### 5.7.2 Definite Klauselgrammatiken

Wir zeigen nun, wie man kontextfreie Grammatiken in Prolog repräsentieren kann und direkt einen Algorithmus erhält, der das Wortproblem löst (d.h., einen Algorithmus, der zu jedem Wort erkennt, ob es in der Sprache liegt oder nicht). Auf diese Weise lassen sich in Prolog leicht Parser für verschiedene (Programmier-)Sprachen implementieren.

Eine kontextfreie Grammatik $G$ ist ein 4-Tupel $G = (N, T, S, P)$, wobei $N$ eine Menge von Nichtterminalsymbolen und $T$ eine Menge von Terminalsymbolen ist. $S \in N$ ist das Startsymbol und $P$ ist eine Menge von Produktionen (oder "Regeln") der Form $A \to \alpha$ mit $A \in N$ und $\alpha \in (N \cup T)^*$. Die zugehörige Ableitungsrelation $\Rightarrow_G$ zwischen Wörtern aus $(N \cup T)^*$ ist definiert als $\beta \Rightarrow_G \gamma$ falls es eine Produktion $A \to \alpha$ in $P$ gibt, so dass $\beta = \beta_1\ A\ \beta_2$ und $\gamma = \beta_1\ \alpha\ \beta_2$ ist. Die *Sprache* von $G$ ist $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$.

Als Beispiel betrachten wir die folgende Grammatik $G = (N, T, S, P)$:

- $N = \{$ Satz, Nominalphrase, Verbalphrase, Artikel, Nomen, Verb $\}$
- $T = \{$ a, the, cat, mouse, scares, hates $\}$
- $S =$ Satz
- $P$ besteht aus folgenden Regeln:

| | | |
|---|---|---|
| Satz | $\to$ | Nominalphrase Verbalphrase |
| Nominalphrase | $\to$ | Artikel Nomen |
| Verbalphrase | $\to$ | Verb |
| Verbalphrase | $\to$ | Verb Nominalphrase |
| Artikel | $\to$ | a |
| Artikel | $\to$ | the |
| Nomen | $\to$ | cat |
| Nomen | $\to$ | mouse |
| Verb | $\to$ | scares |
| Verb | $\to$ | hates |

In der Sprache dieser Grammatik sind z.B. die folgenden Worte:

```
a cat scares the mouse
the mouse hates the cat
a mouse scares a mouse
a mouse hates
```

Zur Darstellung in Prolog verwendet man die folgenden Konventionen:

- Nicht-Terminalsymbole von $N$ werden als Konstanten (d.h. als 0-stellige Prädikatssymbole) geschrieben.

- Terminalsymbole von $T$ werden als einelementige Listen mit einer Konstanten geschrieben.
- Worte aus $T^*$ werden als Listen von Konstanten geschrieben. Das leere Wort $\varepsilon$ wird als `[]` geschrieben.
- Worte aus $(N \cup T)^*$ werden als durch Kommas getrennte Sequenzen aus Konstanten und Listen von Konstanten geschrieben.
- In Regeln schreibt man `-->` statt $\rightarrow$.

Die obige Grammatik lässt sich also wie folgt als ein Prolog-Programm schreiben:

```
satz --> nominalphrase, verbalphrase.
nominalphrase --> artikel, nomen.
verbalphrase --> verb.
verbalphrase --> verb, nominalphrase.
artikel --> [a].
artikel --> [the].
nomen --> [cat].
nomen --> [mouse].
verb --> [scares].
verb --> [hates].
```

Das Wort "a mouse Verbalphrase" würde als "`[a, mouse], verbalphrase`" geschrieben.

Die obige Darstellung muss von Prolog nun in normale Klauseln übersetzt werden. Das Ziel ist es, daraus ein Prolog-Programm zu generieren, mit dem man für jedes Wort überprüfen kann, ob es in der jeweiligen Sprache liegt.

Eine erste Idee wäre, jedem Nicht–Terminalsymbol ein einstelliges Prädikat zuzuordnen, welches testet, ob sein Argument aus dem entsprechenden Nicht–Terminalsymbol mit den Regeln der Grammatik herleitbar ist.

Eine Grammatikregel der Form "`a --> [a1, a2, a3]`" (mit Nichtterminalsymbol `a` und Terminalsymbolen `a1`, `a2`, `a3`) wird dann in das Faktum

```
a([a1, a2, a3]).
```

übersetzt. So erhält man im obigen Beispiel z.B. das Faktum `verb([scares])`. Analog dazu wird eine Regel der Form "`a --> a1`" in

```
a(A) :- a1(A).
```

übersetzt und "`a --> a1, a2`" wird in die folgende Regel übersetzt.

```
a(A) :- append(A1, A2, A),
        a1(A1),
        a2(A2).
```

Man erhält also im Beispiel die Regel

```
satz(S) :- append(NP, VP, S), nominalphrase(NP), verbalphrase(VP).
```

Hierbei wird append allerdings jeweils mit nicht–instantiierten ersten Argumenten aufgerufen, was zu einem sehr ineffizienten Programm führt.

Es wäre daher besser, Listen wieder als Differenzlisten zu repräsentieren. Dann würde `a(A - B)` zutreffen, falls man aus dem Nichtterminalsymbol `a` die Liste von Terminalsymbolen `A` ohne ihr Ende `B` herleiten könnte. Bei der automatischen Übersetzung von Grammatikregeln in Klauseln verwendet Prolog eine Notation, bei der man Differenzlisten nicht in der Form "`A - B`" darstellt, sondern stattdessen *zwei* Argumente verwendet. Somit verwenden wir also nun für jedes Nichtterminalsymbol `a` ein *zweistelliges* Prädikatssymbol, wobei `a(A, B)` zutrifft, falls man aus dem Nichtterminalsymbol `a` die Liste von Terminalsymbolen `A` ohne ihr Ende `B` herleiten kann. Eine Regel der Form "`a --> a1`" wird nun in

```
a(A,B) :- a1(A,B)
```

übersetzt. Eine Regel "`a --> a1, a2`" würde in einem ersten Ansatz in die folgende Regel übersetzt. Hierbei ist `app/3` die Realisierung der Listenkonkatenation mit Differenzlisten.

```
a(A,B) :- app(Xs - Ys, Vs - Ws, A - B),
          a1(Xs, Ys),
          a2(Vs, Ws).
```

Das app–Atom ist genau dann wahr, wenn `Xs` und `A`, `Ws` und `B`, und `Ys` und `Vs` unifizierbar sind. Die obige Regel lässt sich daher umformulieren zu

```
a(A,B) :- a1(A, C),
          a2(C, B).
```

Eine Grammatikregel der Form "`a --> [a1, a2, a3]`" (mit Nichtterminalsymbol `a` und Terminalsymbolen `a1`, `a2`, `a3`) wird in das folgende Faktum übersetzt:

```
a([a1, a2, a3 | Xs], Xs).
```

Eine ähnliche Übersetzung ist auch möglich, falls vor einem Nichtterminalsymbol auf der rechten Seite ein oder mehrere Terminalsymbole stehen. Eine Regel der Form "`a --> a1, [a2, a3], a4`" kann in folgende Klausel übersetzt werden:

```
a(A,B) :- a1(A, [a2, a3 | C]),
          a4(C2, B).
```

Das obige Prolog-Programm mit der Beispielgrammatik würde dann also in folgende Klauseln übersetzt:

```
satz(S, R) :- nominalphrase(S, VP), verbalphrase(VP, R)
nominalphrase(NP, R) :- artikel(NP, N), nomen(N, R).
verbalphrase(VP, R) :- verb(VP, R).
verbalphrase(VP, R) :- verb(VP, NP), nominalphrase(NP, R).
artikel([a | R], R).
```

```
artikel([the | R], R).
nomen([cat | R], R).
nomen([mouse | R], R).
verb([scares | R], R).
verb([hates | R], R).
```

Die Anfrage "`?- satz([the, cat, scares, a, mouse],[]).`" ergibt nun wie erwartet `true`. Analog dazu erhalten wir auch bei der Anfrage "`?- satz([the, cat, scares, a, mouse, trash],[trash]).`" die Antwort `true`.

Ebenso kann man sich alle Elemente der Sprache berechnen lassen. Die Anfrage "`?- satz(S,[]).`" ergibt:

```
S = [a, cat, scares] ;
S = [a, cat, hates] ;
S = [a, cat, scares, a, cat] ;
S = [a, cat, scares, a, mouse] ;
...
```

In der Notation der definiten Klauselgrammatiken sind auch Erweiterungen möglich, um Kontextabhängigkeiten zu beschreiben oder um Syntaxbäume aufzubauen, welche beim Parsing von Programmiersprachen benötigt werden.

# Kapitel 6

# Logikprogrammierung mit Constraints

Nachdem wir nun sowohl die "reine" Logikprogrammierung als auch ihre Implementierung in der Sprache Prolog betrachtet haben, wollen wir uns zum Schluss mit einer wichtigen Erweiterung der Logikprogrammierung, der sogenannten *Logikprogrammierung mit Constraints* ("*Constraint Logic Programming*", CLP) beschäftigen. In Abschnitt 6.1 führen wir zunächst die Logikprogrammierung mit Constraints formal ein. Anschließend diskutieren wir in Abschnitt 6.2, wie die Integration von Constraints in der Programmiersprache Prolog gelöst wird. Weitere Literatur zu dem Thema ist beispielsweise [FA10, MS98].

## 6.1 Syntax und Semantik von Constraint-Logikprogrammen

Unter einem *Constraint* versteht man eine Einschränkung oder Bedingung. Solche Bedingungen werden typischerweise als atomare Formeln formuliert. Die folgende Definition führt diesen Begriff formal ein. Unser Ziel ist nachher, Logikprogramme über einer Signatur $(\Sigma, \Delta)$ um Constraints über einer Teilsignatur $(\Sigma', \Delta')$ zu erweitern. Hierbei erweist es sich als sinnvoll, das Gleichheits-Prädikatssymbol "=" sowie die speziellen Prädikatssymbole $\mathsf{true}$ und $\mathsf{fail}$ gesondert zu behandeln.

**Definition 6.1.1 (Constraint-Signatur und Constraints)** *Sei $(\Sigma, \Delta)$ eine Signatur mit $\mathsf{true}, \mathsf{fail} \in \Delta_0$ und $= \in \Delta_2$. Seien $\Sigma' \subseteq \Sigma$ und $\Delta' \subseteq \Delta$ Teilmengen der Signatur, wobei $\Delta'$ keines der Prädikatssymbole $\mathsf{true}$, $\mathsf{fail}$ oder $=$ enthält. Dann heißt $(\Sigma, \Delta, \Sigma', \Delta')$* Constraint-Signatur*. Die atomaren Formeln aus $\mathcal{A}t(\Sigma', \Delta', \mathcal{V}) \cup \mathcal{A}t(\Sigma, \{=\}, \mathcal{V}) \cup \{\mathsf{true}, \mathsf{fail}\}$ werden dann als* Constraints *über der Signatur $(\Sigma, \Delta, \Sigma', \Delta')$ bezeichnet.*

Bei einer Constraint-Signatur bestimmen wir also eine Teilmenge $\Sigma'$ der Funktionssymbole und eine Teilmenge $\Delta'$ der Prädikatssymbole, aus denen die Constraints gebildet werden. Hierbei dürfen Prädikate aus $\Delta'$ nur auf Terme angewendet werden, die keine Funktionssymbole außer denjenigen aus $\Sigma'$ enthalten. Das Gleichheitszeichen $=$ darf hingegen auf beliebige Terme angewendet werden. Alle atomaren Formeln mit Prädikaten aus $\Delta' \cup \{\mathsf{true}, \mathsf{fail}, =\}$ heißen dann *Constraints*.

**Beispiel 6.1.2** *Als Beispiel betrachten wir eine Constraint-Signatur* $(\Sigma, \Delta, \Sigma', \Delta')$*, bei der* $\Sigma'$ *und* $\Delta'$ *Funktions- bzw. Prädikatssymbole zur Verarbeitung ganzer Zahlen enthalten:*

$$\begin{aligned}
\Sigma'_0 &= \mathbb{Z} \\
\Sigma'_1 &= \{-, \mathsf{abs}\} \\
\Sigma'_2 &= \{+, -, *, /, \mathsf{mod}, \mathsf{min}, \mathsf{max}\} \\
\Delta'_2 &= \{\#>=,\ \#=<,\ \#=,\ \#\backslash=,\ \#>,\ \#<\}
\end{aligned}$$

*Man bezeichnet diese Mengen* $\Sigma'$ *und* $\Delta'$ *auch als* $\Sigma_{FD}$ *und* $\Delta_{FD}$*. Hierbei steht "FD" für Finite Domains. Falls* $\mathsf{f} \in \Sigma_1$*, so ergeben sich unter anderem folgende Constraints:*

$$\begin{aligned}
X + Y \quad &\#> \quad Z * 3 \\
\mathsf{max}(X, Y) \quad &\#= \quad X \bmod 2 \\
\mathsf{f}(X) + 2 \quad &= \quad Y + Z
\end{aligned}$$

Um zu entscheiden, wann ein Constraint wahr ist, benötigt man darüber hinaus eine *Constraint-Theorie* $CT$. Hierbei ist $CT$ eine Menge von Formeln und ein Constraint ist wahr, wenn es aus der Formelmenge $CT$ folgt.

**Definition 6.1.3 (Constraint-Theorie)** *Sei* $(\Sigma, \Delta, \Sigma', \Delta')$ *eine Constraint-Signatur. Falls* $CT \subseteq \mathcal{F}(\Sigma', \Delta', \mathcal{V})$ *erfüllbar ist und nur geschlossene Formeln enthält, so bezeichnen wir* $CT$ *als* Constraint-Theorie*.*

**Beispiel 6.1.4** *Sei* $S_{FD} = (\mathbb{Z}, \alpha)$ *die Struktur mit den ganzen Zahlen als Träger und der "intuitiven" Deutung der Funktions- und Prädikatssymbole. Es gilt also* $\alpha_n = n$ *für alle* $n \in \mathbb{Z}$*. Weiterhin ist* $\alpha_+$ *die Additionsfunktion, etc. und* $\alpha_{\#>}$ *ist die Menge aller Zahlenpaare* $(n, m)$ *mit* $n > m$*, etc. Die naheliegende Constraint-Theorie zu der Signatur aus Bsp. 6.1.2 ist dann die Menge* $CT_{FD}$*, so dass für alle geschlossenen Formeln* $\varphi \in \mathcal{F}(\Sigma_{FD}, \Delta_{FD}, \mathcal{V})$ *gilt:*

$$\varphi \in CT_{FD} \quad \textit{gdw.} \quad S_{FD} \models \varphi$$

*Man beachte, dass* $CT_{FD}$ *nicht nur unendlich, sondern nicht entscheidbar (und nicht einmal semi-entscheidbar) ist. Dies liegt daran, dass wir auch Funktionen wie die Multiplikation in* $\Sigma_{FD}$ *aufgenommen haben. Würden wir uns nur auf die Addition beschränken (man bezeichnet dies dann als* Presburger Arithmetik*), dann könnte man* $CT_{FD}$ *als endliche Menge definieren. Mit anderen Worten, es existiert eine endliche Menge von Axiomen, aus denen dann genau alle wahren Formeln über ganze Zahlen folgen.*

Wir geben nun die Syntax und Semantik von Logikprogrammen mit Constraints an. Die Syntax unterscheidet sich nicht von der Syntax normaler Logikprogramme. Wir gehen dabei davon aus, dass Logikprogramme immer bereits die Klauseln zur Definition von $\mathsf{true}$ und $=$ enthalten und es keine Klausel zur Definition von $\mathsf{fail}$ gibt. Auf den linken Seiten anderer Regeln dürfen die Prädikatssymbole der Constraints nicht auftreten. Außerdem dürfen die Prädikatssymbole aus $\Delta'$ nur auf Terme mit Funktionssymbolen aus $\Sigma'$ angewendet werden.

**Definition 6.1.5 (Syntax von Logikprogrammen mit Constraints)** *Eine nicht–leere endliche Menge $\mathcal{P}$ von definiten Hornklauseln über einer Constraint-Signatur $(\Sigma, \Delta, \Sigma', \Delta')$ heißt* Logikprogramm mit Constraints *über $(\Sigma, \Delta, \Sigma', \Delta')$, falls $\{\mathsf{true}\} \in \mathcal{P}$ und $\{X = X\} \in \mathcal{P}$ und falls für alle anderen Klauseln $\{B, \neg C_1, \ldots, \neg C_n\} \in \mathcal{P}$ gilt:*

*(a) Wenn $B = p(t_1, \ldots, t_m)$, dann gilt $p \notin \Delta' \cup \{\mathsf{true}, \mathsf{fail}, =\}$.*

*(b) Wenn $C_i = p(t_1, \ldots, t_m)$ und $p \in \Delta'$, dann gilt $t_j \in \mathcal{T}(\Sigma', \mathcal{V})$ für alle $1 \leq j \leq m$.*

*Ebenso muss die Bedingung (b) auch für alle Anfragen $\{\neg C_1, \ldots, \neg C_n\}$ gelten.*

**Beispiel 6.1.6** *Wir betrachten wieder eine Constraint-Signatur $(\Sigma, \Delta, \Sigma_{FD}, \Delta_{FD})$. Ein Beispiel für ein Logikprogramm mit Constraints ist dann die folgende Klauselmenge $\mathcal{P}$. Hierbei verwenden wir wieder die übliche Schreibweise für Logikprogramme (als Fakten und Regeln anstelle von Klauseln).*

```
fakt(0,1).
fakt(X,Y) :- X #> 0, X1 #= X-1, fakt(X1,Y1), Y #= X*Y1.
```

*Man erkennt die Ähnlichkeit zu dem Programm zur Berechnung der Fakultät aus Abschnitt 5.1:*

```
fak(0,1).
fak(X,Y) :- X > 0, X1 is X-1, fak(X1,Y1), Y is X*Y1.
```

Nun definieren wir die Semantik der Logikprogrammierung mit Constraints. Analog zu Abschnitt 4.1.1 und 4.1.2 werden wir dies sowohl auf deklarative als auch auf operationelle Weise tun. (Eine Fixpunkt-Semantik der Logikprogrammierung mit Constraints wäre ebenfalls möglich.)

Die deklarative Semantik der Logikprogrammierung mit Constraints ist sehr naheliegend. Bei der normalen Logikprogrammierung werden alle Instantiierungen einer Anfrage als "wahr" betrachtet, die aus den Klauseln des Programms $\mathcal{P}$ folgen. Die Programmklauseln werden hier also als *Axiome* behandelt. Bei der Logikprogrammierung mit Constraints werden die Axiome aus $\mathcal{P}$ einfach um die zusätzlichen Axiome $CT$ erweitert.

**Definition 6.1.7 (Deklarative Semantik eines Constraint-Logikprogramms)** *Sei $\mathcal{P}$ ein Logikprogramm mit Constraints und $CT$ eine dazugehörige Constraint-Theorie. Sei $G = \{\neg A_1, \ldots, \neg A_k\}$ eine Anfrage. Dann ist die* deklarative Semantik *von $\mathcal{P}$ und $CT$ bezüglich $G$ definiert als*

$$D[\![\mathcal{P}, CT, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k),\ \sigma \text{ ist Grundsubstitution}\}.$$

**Beispiel 6.1.8** *Wir betrachten das Programm $\mathcal{P}$ aus Bsp. 6.1.6, die Constraint-Theorie $CT_{FD}$ aus Bsp. 6.1.4 und die Anfrage $G = \{\neg\mathsf{fakt}(1, Z)\}$. Die einzige Grundsubstitution mit $\mathcal{P} \cup CT_{FD} \models \sigma(\mathsf{fakt}(1, Z))$ ist $\sigma(Z) = 1$. Es gilt also $D[\![\mathcal{P}, CT, G]\!] = \{\mathsf{fakt}(1, 1)\}$. Bei der Anfrage $G' = \{\neg\mathsf{fakt}(X, 1)\}$ gilt $\mathcal{P} \cup CT_{FD} \models \sigma_1(\mathsf{fakt}(X, 1))$ und $\mathcal{P} \cup CT_{FD} \models \sigma_2(\mathsf{fakt}(X, 1))$ für die Substitutionen $\sigma_1(X) = 0$ und $\sigma_2(X) = 1$. Man erwartet daher bei der Anfrage*

```
?- fakt(1,Z).
```

*die Antwort* `Z = 1` *und bei der Anfrage*

```
?- fakt(X,1).
```

*die Antworten* `X = 0` *und* `X = 1`.

Def. 6.1.7 zeigt deutlich, dass die normale Logikprogrammierung ein Spezialfall der Logikprogrammierung mit Constraints ist. Offensichtlich entsprechen Logikprogramme mit leerer Constraint-Theorie $CT$ gerade den bislang betrachteten normalen Logikprogrammen.

**Korollar 6.1.9** *Sei $\mathcal{P}$ ein Logikprogramm mit Constraints über $\Sigma' = \varnothing$ und $\Delta' = \varnothing$. Dann gilt für alle Anfragen $G$: $D[\![\mathcal{P}, \varnothing, G]\!] = D[\![\mathcal{P}, G]\!]$.*

Um zu erklären, wie Logikprogramme mit Constraints *ausgewertet* werden, wollen wir nun auch die prozedurale Semantik definieren. Dies ist jedoch nicht ganz so einfach wie die deklarative Semantik. Das Problem ist, dass die Axiome in $CT$ beliebige Formeln sein können (und nicht unbedingt nur definite Hornklauseln). Ob eine Anfrage aus den Programmklauseln von $\mathcal{P}$ folgt, lässt sich mit binärer SLD-Resolution untersuchen. Dies ist bei den Axiomen in $CT$ so nicht möglich. Stattdessen gehen wir ja davon aus, dass wir eine Technik besitzen, die Folgerbarkeit aus $CT$ automatisch überprüft. Das Problem besteht jetzt darin, diese Technik (zur Überprüfung der Folgerbarkeit aus $CT$) mit der SLD-Resolution (zur Überprüfung der Folgerbarkeit aus $\mathcal{P}$) zu verbinden. Die Idee hierzu besteht darin, auch die SLD-Resolutionsschritte mit Hilfe von Constraints darzustellen. Auf diese Weise erhält man eine einheitliche Darstellung beider Arten von Beweisschritten (den Schritten, die SLD-Resolution mit Klauseln aus $\mathcal{P}$ durchführen, und den Schritten, die Constraints mit Hilfe der Constraint-Theorie $CT$ lösen). Wir werden daher Unifikations-Informationen explizit mit Hilfe von Gleichheiten als Constraints repräsentieren. Wie das folgende Beispiel zeigt, könnte man diese Darstellung auch bei normalen Logikprogrammen ohne Constraints verwenden.

**Beispiel 6.1.10** *Wir betrachten hierzu das folgende (reine) Logikprogramm aus Abschnitt 5.1.*

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

*Bei der bisherigen Definition der prozeduralen Semantik bestand eine Konfiguration $(G, \sigma)$ aus der momentanen Anfrage $G$ und der bereits berechneten Substitution. Die Berechnung*

*startete mit der leeren (oder "identischen") Substitution $\varnothing$. Bei der Anfrage* $\{\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U)\}$ *ergab sich also*

$$\begin{array}{ll} & (\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U),\ \varnothing) \\ \vdash_{\mathcal{P}} & (\neg\mathsf{add}(\mathsf{s}(0), 0, Z),\ \{X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(Z)\}) \\ \vdash_{\mathcal{P}} & (\square,\ \underbrace{\{X'/\mathsf{s}(0),\ Z/\mathsf{s}(0)\} \circ \{X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(Z)\}}_{\{X'/\mathsf{s}(0), Z/\mathsf{s}(0), X/\mathsf{s}(0), Y/0, U/\mathsf{s}(\mathsf{s}(0))\}}) \end{array}$$

*Aus der zum Schluss erhaltenen Substitution erhält man die Antwortsubstitution, indem man sie auf die Variable $U$ aus der ursprünglichen Anfrage einschränkt. Es ergibt sich also die Antwortsubstitution* $\{U/\mathsf{s}(\mathsf{s}(0))\}$.

*Nun ist die Idee, die Unifikationen nicht durchzuführen, sondern anstelle der benötigten Unifikatoren einfach nur die Unifikationsprobleme aufzusammeln. Hierbei steht "$\overline{A = B}$" für das Problem, die beiden atomaren Formeln $A$ und $B$ zu unifizieren. Die Konfigurationen haben nun die Gestalt $(G, CO)$, wobei $CO$ eine Konjunktion von Unifikationsproblemen der Form "$\overline{A = B}$" ist. Man startet hierbei mit der leeren Konjunktion, d.h., mit der Formel* $\mathsf{true}$, *die stets wahr ist.*

$$\begin{array}{ll} & (\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U),\ \mathsf{true}) \\ \vdash_{\mathcal{P}} & (\neg\mathsf{add}(X, Y, Z),\ \overline{\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U) = \mathsf{add}(X, \mathsf{s}(Y), \mathsf{s}(Z))}) \\ \vdash_{\mathcal{P}} & (\square,\ \overline{\mathsf{add}(X, Y, Z) = \mathsf{add}(X', 0, X')} \wedge \overline{\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U) = \mathsf{add}(X, \mathsf{s}(Y), \mathsf{s}(Z))}) \end{array}$$

*Die zum Schluss erhaltene Konjunktion von Unifikationsproblemen kann man natürlich weiter vereinfachen. Sie ist im Endeffekt äquivalent zu der Bedingung*

$$X' = \mathsf{s}(0) \wedge Z = \mathsf{s}(0) \wedge X = \mathsf{s}(0) \wedge Y = 0 \wedge U = \mathsf{s}(\mathsf{s}(0)) \tag{6.1}$$

*Die entspricht also gerade der Substitution, die man bislang bei der prozeduralen Semantik von Logikprogrammen erhalten hat.*

Der Unterschied zur prozeduralen Semantik normaler Logikprogramme aus Def. 4.1.5 ist also, dass wir nun nicht mehr Konfigurationen der Form $(G, \sigma)$ betrachten, bei denen $\sigma$ eine Substitution ist. Stattdessen haben die Konfigurationen nun die Gestalt $(G, CO)$. Hierbei soll $CO$ eine Konjunktion von Constraints sein. $CO$ darf also Gleichungen zwischen Termen enthalten. Das Gleichheits-Prädikatssymbol kann aber nicht auf zwei atomare Formeln angewendet werden (d.h. man kann nicht $A = B$ schreiben). Aus diesem Grund definieren wir $\overline{A = B}$ als Abkürzung für eine entsprechende Konjunktion von Gleichheiten zwischen Termen.

**Definition 6.1.11 (Gleichheit von Atomen)** *Seien $A$ und $B$ Atome. Dann definieren wir die Formel $\overline{A = B}$ wie folgt:*

- $\overline{A = B}$ *ist die Formel* $\mathsf{fail}$, *falls $A = p(\ldots)$, $B = q(\ldots)$ mit $p \neq q$*
- $\overline{A = B}$ *ist die Formel* $\mathsf{true}$, *falls $A = B = p$*
- $\overline{A = B}$ *ist die Formel $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$, falls $A = p(s_1, \ldots, s_n)$ und $B = p(t_1, \ldots, t_n)$*

**Beispiel 6.1.12** *Mit der obigen Definition von $\overline{A = B}$ sehen die Rechenschritte von Bsp. 6.1.10 nun folgendermaßen aus:*

$$\begin{array}{ll} & (\neg\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U),\ \mathsf{true}) \\ \vdash_{\mathcal{P}} & (\neg\mathsf{add}(X,Y,Z),\ \underbrace{\overline{\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U) = \mathsf{add}(X,\mathsf{s}(Y),\mathsf{s}(Z))}}_{\mathsf{s}(0)=X\,\wedge\,\mathsf{s}(0)=\mathsf{s}(Y)\,\wedge\,U=\mathsf{s}(Z)}) \\ \vdash_{\mathcal{P}} & (\square,\ \underbrace{\overline{\mathsf{add}(X,Y,Z) = \mathsf{add}(X',0,X')}}_{X=X'\,\wedge\,Y=0\,\wedge\,Z=X'} \wedge\ \mathsf{s}(0) = X\ \wedge\ \mathsf{s}(0) = \mathsf{s}(Y)\ \wedge\ U = \mathsf{s}(Z)) \end{array}$$

*Es ergibt sich zum Schluss also die folgende Konjunktion von Constraints:*

$$X = X' \wedge Y = 0 \wedge Z = X' \wedge \mathsf{s}(0) = X \wedge \mathsf{s}(0) = \mathsf{s}(Y) \wedge U = \mathsf{s}(Z) \tag{6.2}$$

Um die hierdurch entstehenden Konjunktionen von Constraints $CO$ zu vereinfachen, kann man eine Funktion "simplify" verwenden, die Konjunktionen von Constraints in hierzu äquivalente Konjunktionen von (einfacheren) Constraints überführt. Dabei ist zu überlegen, was "Äquivalenz" hier bedeuten soll. Die bislang einzigen möglichen Prädikatssymbole in $CO$ sind $\mathsf{true}$, $\mathsf{fail}$ und $=$. Man sollte bei der Vereinfachung von $CO$ daher die Axiome über $\mathsf{true}$, $\mathsf{fail}$ und $=$ beachten. Wir verlangen daher für die Funktion simplify, dass

$$\{\forall X\ X = X, \mathsf{true}\} \models \forall\ (CO \leftrightarrow \mathrm{simplify}(CO))$$

gilt. Zu jeder quantorfreien Formel $\varphi$ mit den Variablen $X_1, \ldots, X_n$ bezeichnet $\forall\,\varphi$ den *Allabschluss* von $\varphi$, d.h. die Formel $\forall X_1, \ldots, X_n\,\varphi$. Analog dazu ist $\exists\,\varphi$ der *Existenzabschluss* von $\varphi$, d.h. die Formel $\exists X_1, \ldots, X_n\,\varphi$.

In Bsp. 6.1.12 könnte dann simplify( (6.2) ) = (6.1) sein, da offensichtlich bereits

$$\forall X\ X = X \quad \models \quad \forall X', X, Y, Z, U\ \ (6.2) \leftrightarrow (6.1)$$

gilt. Selbstverständlich kann man natürlich auch schon nach jedem Berechnungsschritt "simplify" anwenden, um die zwischendurch erhaltenen Konjunktionen von Constraints zu vereinfachen.

Man erkennt, dass anstelle der Unifikatoren nun also nur noch die Gleichheiten zwischen den zu unifizierenden Termen aufgesammelt werden. Dass dies in der Tat zu einem äquivalenten Ansatz führt, liegt daran, dass das Axiom "$\forall X\ X = X$" sicher stellt, dass zwei Terme nur dann als "gleich" betrachtet werden, wenn sie syntaktisch gleich sind. Die Formel "$\forall X\ X = X$" dient somit also zur "Axiomatisierung der Unifikation". Dies wird durch das folgende Lemma ausgedrückt, das wir benötigen werden, um die Äquivalenz der deklarativen und der prozeduralen Semantik zu beweisen.

**Lemma 6.1.13 (Gleichheit und Unifikation)** *Für alle Terme $s, t$, alle Atome $A, B$ und alle Substitutionen $\sigma$ gilt:*

*(a)* $\forall X\ X = X \ \models\ \sigma(s = t)$ *gdw.* $\sigma(s)$ *und* $\sigma(t)$ *syntaktisch gleich sind.*

*(b)* $\{\forall X\ X = X, \mathsf{true}\} \ \models\ \sigma(\overline{A = B})$ *gdw.* $\sigma(A)$ *und* $\sigma(B)$ *syntaktisch gleich sind.*

**Beweis.**

(a) Wir zeigen zunächst die Richtung von rechts nach links. Sei $I$ eine beliebige Interpretation, die Modell von $\forall X\ X = X$ ist. Da $\sigma(s) = \sigma(t)$ und somit $I(\sigma(s)) = I(\sigma(t))$ ist, folgt auch $I \models \sigma(s) = \sigma(t)$.

Nun beweisen wir die Richtung von links nach rechts. Wir betrachten die Interpretation $I = (\mathcal{T}(\Sigma, \mathcal{V}), \alpha, \beta)$ mit $\alpha_f = f$ für alle $f \in \Sigma$, $\alpha_= = \{(r, r) \mid r \in \mathcal{T}(\Sigma, \mathcal{V})\}$ und $\beta(X) = X$. Hier gilt also $I(r) = r$ für alle Terme $r$ und $I \models r_1 = r_2$ gilt gdw. $I(r_1) = I(r_2)$ ist, d.h. gdw. $r_1$ und $r_2$ syntaktisch gleich sind. Offensichtlich ist $I$ Modell von $\forall X\ X = X$. Aus der Voraussetzung folgt also, dass $I \models \sigma(s = t)$ und somit $\sigma(s) = \sigma(t)$.

(b) Falls $A = p(\ldots)$, $B = q(\ldots)$ mit $p \neq q$, so existiert kein $\sigma$, so dass $\sigma(A)$ und $\sigma(B)$ syntaktisch gleich sind. Da $\overline{A = B} = \mathsf{fail}$ ist, gilt daher auch $\{\forall X\ X = X, \mathsf{true}\} \not\models \sigma(\overline{A = B})$.

Falls $A = B = p$, so sind $\sigma(A)$ und $\sigma(B)$ für alle Substitutionen $\sigma$ syntaktisch gleich. Da in diesem Fall $\overline{A = B} = \mathsf{true}$ ist, gilt hier auch $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A = B})$.

Falls $A = p(s_1, \ldots, s_n)$ und $B = p(t_1, \ldots, t_n)$ ist, so gilt $\sigma(A) = \sigma(B)$, falls $\sigma(s_i) = \sigma(t_i)$ für alle $1 \leq i \leq n$ gilt. Nach (a) ist dies genau dann der Fall, wenn $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(s_i = t_i)$ für alle $i$ gilt. Dies ist also gleichbedeutend mit $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A = B})$. □

Da"$\{\forall X\ X = X, \mathsf{true}\}$" die Unifizierbarkeit axiomatisiert, kann man auf diese Weise natürlich auch feststellen, wann zwei Terme nicht unifizierbar sind. Man sollte eine Konfiguration $(G_1, CO_1)$ daher nur dann in eine neue Konfiguration $(G_2, CO_2)$ überführen, wenn die neue Bedingung $CO_2$ unter diesen Axiomen erfüllbar ist, d.h. wenn $\{\forall X\ X = X, \mathsf{true}\} \models \exists\, CO_2$. Auf diese Weise wird z.B. verhindert, dass in Bsp. 6.1.12 die erste Programmklausel direkt bei der Anfrage $\{\neg\mathsf{add}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), U)\}$ angewendet wird. Man würde dann nämlich die Konjunktion $\mathsf{s}(\mathsf{0}) = X \wedge \mathsf{s}(\mathsf{0}) = \mathsf{0} \wedge U = X$ erhalten. Es gilt aber

$$\{\forall X\ X = X, \mathsf{true}\} \not\models \exists X, U\ \mathsf{s}(\mathsf{0}) = X \wedge \mathsf{s}(\mathsf{0}) = \mathsf{0} \wedge U = X.$$

Um nun die prozedurale Semantik von Logikprogrammen mit Constraints zu definieren, erweitern wir die in Bsp. 6.1.10 und 6.1.12 vorgestellte Idee wie folgt: Die zweite Komponente $CO$ einer Konfiguration kann nun nicht nur Constraints mit $\mathsf{true}$, $\mathsf{fail}$ und $=$ enthalten, sondern es sind nun auch Constraints möglich, die mit Prädikatssymbolen aus $\Delta'$ gebildet werden. Entsprechend muss man nun auch die Axiome $CT$ zusätzlich zu den Axiomen $\forall X\ X = X$ und $\mathsf{true}$ betrachten, wenn man die Erfüllbarkeit von $CO$ untersucht und wenn man $CO$ mit "simplify" vereinfacht. Hierbei geht man davon aus, dass man ein Verfahren zur Verfügung hat, um die Gültigkeit existenzquantifizierter Constraints zu entscheiden. Mit anderen Worten, falls $CO$ eine Konjunktion von Constraints ist, so geht man davon aus, dass es entscheidbar ist, ob $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\, CO$ gilt. (In der Realität ist dies natürlich bei Constraint-Theorien wie $CT_{FD}$ nicht der Fall. In Abschnitt 6.2 werden wir daher diskutieren, wie man dieses Problem in der Praxis "löst".)

**Definition 6.1.14 (Prozedurale Semantik eines Constraint-Logikprogramms)**
*Sei $\mathcal{P}$ ein Logikprogramm mit Constraints und $CT$ eine dazugehörige Constraint-Theorie.*

- *Eine* Konfiguration *ist ein Paar $(G, CO)$, wobei $G$ eine Anfrage oder die leere Klausel $\square$ ist und wobei $CO$ eine Konjunktion von Constraints ist.*

- *Es gibt einen* Rechenschritt $(G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2)$ *gdw. $G_1 = \{\neg A_1, \ldots, \neg A_k\}$ mit $k \geq 1$ und eine der beiden folgenden Möglichkeiten (A) oder (B) zutrifft:*

*(A) Es gibt ein $1 \leq i \leq k$, so dass $A_i$ kein Constraint ist. Dann:*

  - *existiert eine Programmklausel $K \in \mathcal{P}$ und eine Variablenumbenennung $\nu$ mit $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ und $n \geq 0$, so dass*
    - *$\nu(K)$ keine gemeinsamen Variablen mit $G_1$ oder $CO_1$ hat*
    - *$CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ (CO_1 \wedge \overline{A_i = B})$*
  - $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}$
  - $CO_2 = CO_1 \wedge \overline{A_i = B}$

*(B) Es gibt ein $1 \leq i \leq k$, so dass $A_i$ ein Constraint ist. Dann:*

  - $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO_1 \wedge A_i$
  - $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg A_{i+1}, \ldots, \neg A_k\}$
  - $CO_2 = CO_1 \wedge A_i$

- *Eine* Berechnung *von $\mathcal{P}$ bei Eingabe von $G = \{\neg A_1, \ldots, \neg A_k\}$ ist eine (endliche oder unendliche) Folge von Konfigurationen der Form*

$$(G, \mathsf{true}) \vdash_{\mathcal{P}} (G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2) \vdash_{\mathcal{P}} \ldots$$

- *Eine mit $(\square, CO)$ terminierende Berechnung, die mit $(G, \mathsf{true})$ startet, heißt* erfolgreich. *Die berechneten* Antwortconstraints *sind* $\text{simplify}(CO)$, *wobei $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \forall\ (CO \leftrightarrow \text{simplify}(CO))$.*

*Damit ist die* prozedurale Semantik *von $\mathcal{P}$ bezüglich $G$ definiert als*

$$P[\![\mathcal{P}, CT, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid (G, \mathsf{true}) \vdash_{\mathcal{P}}^{+} (\square, CO), \\ \sigma \text{ ist Grundsubstitution mit} \\ CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)\}.$$

Es gibt also nun zwei Möglichkeiten für einen Berechnungsschritt, je nachdem ob das ausgewählte Atom $A_i$ aus der Anfrage ein Constraint ist oder nicht. Falls es kein Constraint ist (Fall (A)), so findet wie bisher SLD-Resolution mit einer Programmklausel statt. Allerdings wird die benötigte Unifikation von $A_i$ und dem Kopf $B$ der Programmklausel nicht durchgeführt, sondern man nimmt stattdessen die Constraints $\overline{A_i = B}$ mit auf. Hierbei muss jeweils geprüft werden, ob die aufgesammelte Konjunktion von Constraints erfüllbar bleibt. Falls $A_i$ hingegen ein Constraint ist (Fall (B)), so wird $A_i$ direkt in die aufgesammelte Konjunktion von Constraints mit aufgenommen, sofern diese erfüllbar bleibt.

**Beispiel 6.1.15** *Wir betrachten wieder das Programm $\mathcal{P}$ aus Bsp. 6.1.6, die Constraint-Theorie $CT_{FD}$ aus Bsp. 6.1.4 und die Anfrage $G = \{\neg\mathsf{fakt}(1, Z)\}$. Während Bsp. 6.1.8 zeigte, dass sich bei der deklarativen Semantik $D[\![\mathcal{P}, CT, G]\!] = \{\mathsf{fakt}(1, 1)\}$ ergibt, wollen wir nun die prozedurale Semantik illustrieren. Hierbei wenden wir die Funktion "*simplify*" nach jedem Rechenschritt an, um die erhaltene Konjunktion von Constraints zu vereinfachen. Außerdem haben wir die Negationen vor den einzelnen Literalen weggelassen, um die Lesbarkeit zu erhöhen. Man erhält*

$$
\begin{array}{ll}
 & (\mathsf{fakt}(1, Z), \quad \mathsf{true}) \\
\vdash_{\mathcal{P}} & (X\#{>}\, 0, X_1\#{=}\, X - 1, \mathsf{fakt}(X_1, Y_1), Y\#{=}\, X * Y_1, \quad \underbrace{\mathsf{true} \wedge \overline{\mathsf{fakt}(1, Z) = \mathsf{fakt}(X, Y)}}_{X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}} & (X_1\#{=}\, X - 1, \mathsf{fakt}(X_1, Y_1), Y\#{=}\, X * Y_1, \quad \underbrace{X\#{>}\, 0 \wedge X = 1 \wedge Z = Y}_{X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}} & (\mathsf{fakt}(X_1, Y_1), Y\#{=}\, X * Y_1, \quad \underbrace{X_1\#{=}\, X - 1 \wedge X = 1 \wedge Z = Y}_{X_1=0 \wedge X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}} & (Y\#{=}\, X * Y_1, \quad \underbrace{\overline{\mathsf{fakt}(X_1, Y_1) = \mathsf{fakt}(0, 1)} \wedge X_1 = 0 \wedge X = 1 \wedge Z = Y}_{X_1=0 \wedge Y_1=1 \wedge X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}} & (\Box, \quad \underbrace{Y\#{=}\, X * Y_1 \wedge X_1 = 0 \wedge Y_1 = 1 \wedge X = 1 \wedge Z = Y}_{Y=1 \wedge X_1=0 \wedge Y_1=1 \wedge X=1 \wedge Z=1})
\end{array}
$$

*Das Ergebnis der abschließenden Simplifikation ist somit die folgende Formel CO:*

$$Y = 1 \wedge X_1 = 0 \wedge Y_1 = 1 \wedge X = 1 \wedge Z = 1$$

*Die einzige Grundsubstitution mit $CT \cup \{\forall X \; X = X, \mathsf{true}\} \models \sigma(CO)$ ist daher $\sigma = \{Y/1, X_1/0, Y_1/1, X/1, Z/1\}$. Somit ergibt sich $P[\![\mathcal{P}, CT, G]\!] = \{\mathsf{fakt}(1, 1)\}$.*

Nun können wir die Äquivalenz der deklarativen und der prozeduralen Semantik für Logikprogramme mit Constraints beweisen. Dies zeigt dann, dass eine Implementierung der Logikprogrammierung mit Constraints wie in Def. 6.1.14 tatsächlich der gewünschten (deklarativen) Semantik entspricht und dass so (aufgrund von Korollar 6.1.9) tatsächlich auch die normale Logikprogrammierung korrekt implementiert wird, sofern die Constraint-Theorie $CT$ leer ist. Zum Beweis des folgenden Satzes gehen wir ähnlich wie im Beweis des entsprechenden Satzes für normale Logikprogramme (Satz 4.1.8) vor.

**Satz 6.1.16 (Äquivalenz der deklarativen und prozeduralen Semantik)** *Sei $\mathcal{P}$ ein Logikprogramm mit Constraints und $CT$ eine dazugehörige Constraint-Theorie. Weiter sei $G = \{\neg A_1, \ldots, \neg A_k\}$ eine Anfrage. Dann gilt $D[\![\mathcal{P}, CT, G]\!] = P[\![\mathcal{P}, CT, G]\!]$.*

**Beweis.** Wir zeigen zuerst die Korrektheit der prozeduralen Semantik bezüglich der deklarativen Semantik, d.h. $P[\![\mathcal{P}, CT, G]\!] \subseteq D[\![\mathcal{P}, CT, G]\!]$. Sei $\sigma(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, CT, G]\!]$. Dann gibt es eine erfolgreiche Berechnung der Form

$$(G, \mathsf{true}) \vdash_{\mathcal{P}} (G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2) \ldots \vdash_{\mathcal{P}} (\Box, CO)$$

mit $CT \cup \{\forall X \; X = X, \mathsf{true}\} \models \sigma(CO)$.

Zu zeigen ist, dass dann auch $\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k)$ gilt. Wir verwenden Induktion über die Länge $l$ der Berechnung. Genauer ist $l$ die Anzahl der $\vdash_{\mathcal{P}}$-Schritte in der Berechnung. Es existiert also ein Atom $A_i$ in der Anfrage $G$, das im ersten Schritt $(G, \mathsf{true}) \vdash_{\mathcal{P}} (G_1, CO_1)$ in der Anfrage ersetzt bzw. weggelassen wird.

1. Fall: $A_i$ ist kein Constraint

Dann existiert eine Programmklausel $K \in \mathcal{P}$ und eine Variablenumbenennung $\nu$ mit $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ und $n \geq 0$. Hierbei haben $G$ und $\nu(K)$ keine gemeinsamen Variablen, es gilt $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ \overline{A_i = B}$ und wir haben

$$G_1 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\} \text{ und } CO_1 = \overline{A_i = B}. \tag{6.3}$$

Im Induktionsanfang ist $l = 1$. Dann ist $G_1 = \square$ und somit $i = k = 1$, $n = 0$, $\nu(K) = \{B\}$ (d.h., die Programmklausel $K$ ist ein Faktum) und $CO = CO_1$. Da $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ gilt, folgt also $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_1 = B})$ und somit auch $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_1 = B})$, da $CT$ die Prädikatssymbole $\mathsf{true}$, $\mathsf{fail}$ und $=$ nicht enthält. Nach Lemma 6.1.13 (b) folgt also, dass $\sigma$ ein Unifikator von $A_1$ und $B$ ist, d.h. $\sigma(A_1) = \sigma(B)$. Da außerdem $B$ eine Klausel von $\mathcal{P}$ ist, gilt demnach $\mathcal{P} \models \sigma(B)$ bzw. $\mathcal{P} \cup CT \models \sigma(B)$ und somit auch $\mathcal{P} \cup CT \models \sigma(A_1)$.

Nun betrachten wir den Induktionsschritt $l > 1$. Für $G_1$ wie in (6.3) ist dann offensichtlich auch

$$(G_1, \mathsf{true}) \vdash_{\mathcal{P}} (G_2, CO_2') \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, CO')$$

eine Berechnung, wobei $CO = \overline{A_i = B} \wedge CO'$ ist. Aus $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ folgt daher auch $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO')$. Nach der Induktionshypothese ergibt sich also

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge C_1 \wedge \ldots \wedge C_n \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Da $\mathcal{P} \models C_1 \wedge \ldots \wedge C_n \to B$ gilt, folgt offensichtlich auch

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge B \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Da außerdem aus $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ auch $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_i = B})$ folgt (und somit $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_i = B})$), erhält man aus Lemma 6.1.13 (b), dass $\sigma(A_i) = \sigma(B)$ gilt. So ergibt sich

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

2. Fall: $A_i$ ist ein Constraint

Dann ist $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ A_i$ und wir haben

$$G_1 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg A_{i+1}, \ldots, \neg A_k\} \text{ und } CO_1 = A_i. \tag{6.4}$$

Im Induktionsanfang ist $l = 1$. Dann gilt wieder $G_1 = \square$ und somit $i = k = 1$ und $CO = CO_1$. Da $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ gilt, folgt also $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_1)$ und somit auch $\mathcal{P} \cup CT \models \sigma(A_1)$, da $\mathcal{P}$ auch $\{\forall X\ X = X, \mathsf{true}\}$ enthält.

Nun betrachten wir den Induktionsschritt $l > 1$. Für $G_1$ wie in (6.4) ist dann offensichtlich auch

$$(G_1, \mathsf{true}) \vdash_{\mathcal{P}} (G_2, CO_2') \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\Box, CO')$$

eine Berechnung, wobei $CO = A_i \wedge CO'$ ist. Aus $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ folgt daher auch $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO')$. Nach der Induktionshypothese ergibt sich also

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Da aus $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ auch $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_i)$ folgt, ergibt sich

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_k),$$

da $\mathcal{P}$ auch $\{\forall X\ X = X, \mathsf{true}\}$ enthält.

Nun zeigen wir die Vollständigkeit der prozeduralen Semantik bezüglich der deklarativen Semantik, d.h. $D[\![\mathcal{P}, CT, G]\!] \subseteq P[\![\mathcal{P}, CT, G]\!]$. Sei $\sigma(A_1 \wedge \ldots \wedge A_k) \in D[\![\mathcal{P}, CT, G]\!]$. Dann gilt $\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k)$. Wir trennen nun die einzelnen Atome $A_1, \ldots, A_k$ danach auf, ob sie Constraints sind oder nicht. O.B.d.A. seien $A_1, \ldots, A_j$ keine Constraints und $A_{j+1}, \ldots, A_k$ seien Constraints (wobei $0 \leq j \leq k$). Da $\mathcal{P}$ (außer den Klauseln $\{X = X\}$ und $\{\mathsf{true}\}$) keine positiven Literale mit Prädikatssymbolen aus $\Delta' \cup \{\mathsf{true}, \mathsf{fail}, =\}$ enthält und andererseits $CT$ nur die Prädikatssymbole aus $\Delta'$ enthält, ist $\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k)$ äquivalent zu $\mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_j)$ und $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_{j+1} \wedge \ldots \wedge A_k)$. Wie im Vollständigkeitsbeweis der prozeduralen Semantik für normale Logikprogramme (Satz 4.1.8) zeigt man, dass aus $\mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_j)$ folgt, dass es die Berechnung

$$(\{\neg A_1, \ldots, \neg A_j\}, \mathsf{true}) \vdash_{\mathcal{P}}^{+} (\Box, CO_1)$$

gibt. Somit existiert also auch die Berechnung

$$(\{\neg A_1, \ldots, \neg A_k\}, \mathsf{true}) \vdash_{\mathcal{P}}^{+} (\{\neg A_{j+1}, \ldots, \neg A_k\}, CO_1).$$

Hierbei enthält $CO_1$ die Gleichungen zwischen den zu unifizierenden Atomen. Da $\sigma$ ein Unifikator ist, folgt mit Lemma 6.1.13 (b), dass auch $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO_1)$ gilt. Weiter gilt auch

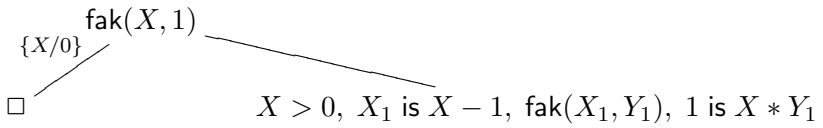
$$(\{\neg A_{j+1}, \ldots, \neg A_k\}, CO_1) \vdash_{\mathcal{P}}^{+} (\Box, CO_1 \wedge CO_2),$$

wobei $CO_2 = A_{j+1} \wedge \ldots \wedge A_k$. Wegen $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_{j+1} \wedge \ldots \wedge A_k)$ haben wir nun also $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO_1 \wedge CO_2)$. Damit folgt $\sigma(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, CT, G]\!]$. □

Um den Indeterminismus 2. Art (d.h. den Indeterminismus bei der Auswahl des Literals aus der Anfrage) aufzulösen, geht man wieder genau wie bei der normalen Logikprogrammierung vor. Auch bei der Logikprogrammierung mit Constraints beschränkt man sich also auf kanonische Berechnungen, d.h. auf Berechnungen, bei denen stets das linkeste Literal der Anfrage gewählt wird. In Def. 6.1.14 wäre somit stets $i = 1$. Der Indeterminismus 1. Art wird ebenfalls wieder dadurch aufgelöst, dass der SLD–Baum in Tiefensuche von links nach rechts durchsucht wird. Dies bedeutet, dass Programmklauseln wieder in der Reihenfolge von oben nach unten betrachtet werden.

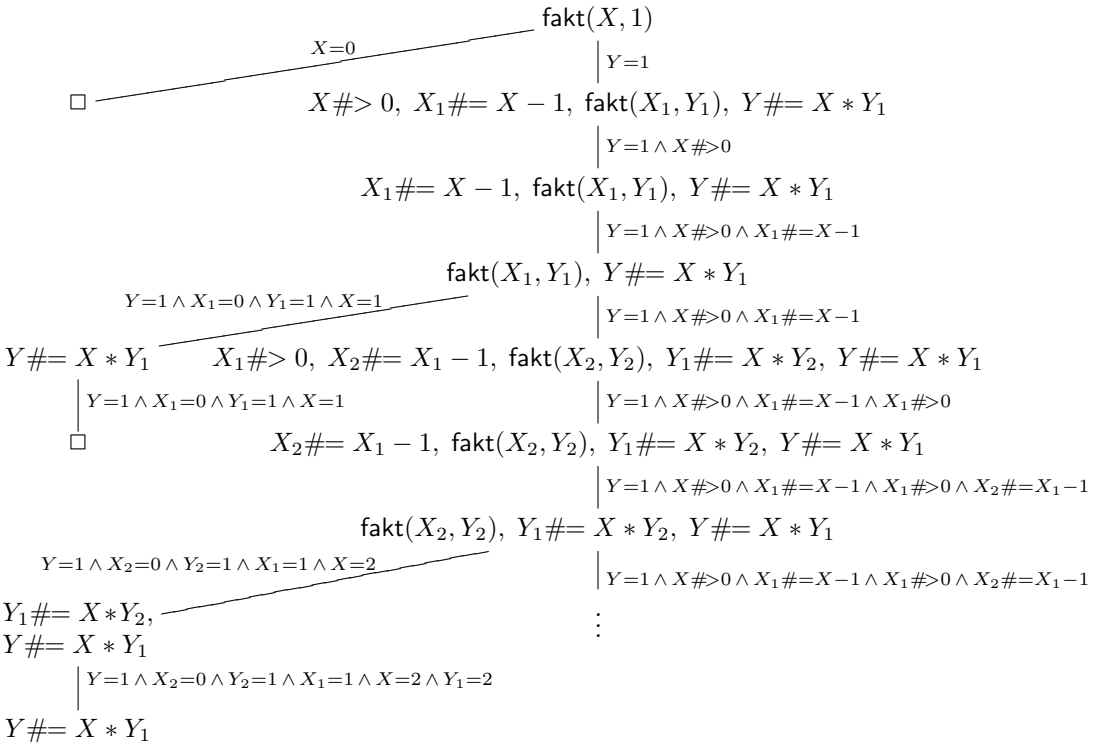
Um die dadurch entstehenden SLD–Bäume zu illustrieren, betrachten wir noch einmal das `fak`- und das `fakt`-Programm aus Bsp. 6.1.6.

**Beispiel 6.1.17** *Für das* `fak`*-Programm aus Bsp. 6.1.6 und die Anfrage "*`?- fak(X,1)`*" ergibt sich der folgende SLD–Baum.*




*Die erste Antwortsubstitution ist also* `X = 0`*. Wenn man anschließend "*`;`*" eingibt, so führt dies jedoch zu einem Programmabbruch, da man das Beweisziel "*`X > 0`*" in* Prolog *nicht stellen darf. Das Problem ist, dass das Prädikat* `>` *zwei voll instantiierte arithmetische Ausdrücke als Argument erwartet. Das Programm* `fak` *ist also nicht bidirektional. Es eignet sich zur Berechnung der Fakultät von vorgegebenen Zahlen (d.h. für Anfragen wie "*`?- fak(1,Z)`*"), aber nicht dazu, zu einer vorgegebenen Zahl (wie* `1`*) herauszufinden, ob es eine Zahl* `X` *gibt, so dass die Fakultät von* `X` *gerade* `1` *ist.*

*Hier zeigt sich einer der Vorteile der Logikprogrammierung mit Constraints. Im Unterschied zu "*`>`*" ist das Prädikatssymbol "*`#>`*" tatsächlich bidirektional. Wir stellen daher nun für das* `fakt`*-Programm aus Bsp. 6.1.6 die Anfrage "*`?- fakt(X,1)`*". Bei den SLD-Bäumen werden wir jetzt die Kanten jeweils mit der aufgesammelten Konjunktion CO von Constraints beschriften, wobei wir diese (mit "*simplify*") vereinfachen. Falls für die entstehende Konjunktion CO nicht* $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO$ *gilt, so wird der entsprechende Kind–Knoten im SLD–Baum nicht gebildet. Es ergibt sich daher folgender SLD–Baum:*




*Der rechte Pfad in dem Baum ist unendlich. Das Problem ist hier, dass das Constraint* $Y\#= X*Y_1$ *aus der Anfrage nie in die Konjunktion CO von Constraints aufgenommen wird, da immer das* $\mathsf{fakt}(\ldots)$*-Constraint links davon steht. Würde man stattdessen* $Y\#= X * Y_1$

*mit in CO aufnehmen, so würde man den Pfad sofort erfolglos abschließen können, da aus*

$$Y = 1 \wedge X\#> 0 \wedge X_1\#= X - 1 \wedge X_1\#> 0$$

*folgt, dass* $Y = 1$ *und* $X\#> 1$ *sein müssen, was jedoch dem Constraint* $Y\#= X * Y_1$ *widerspricht.*

*Aus einem ähnlichen Grund lässt sich der unterste Pfad mit dem Knoten "*$Y\#= X * Y_1$*" nicht mehr fortsetzen, da "*$Y\#= X * Y_1$*" den bisherigen Constraints*

$$Y = 1 \wedge X_2 = 0 \wedge Y_2 = 1 \wedge X_1 = 1 \wedge X = 2 \wedge Y_1 = 2$$

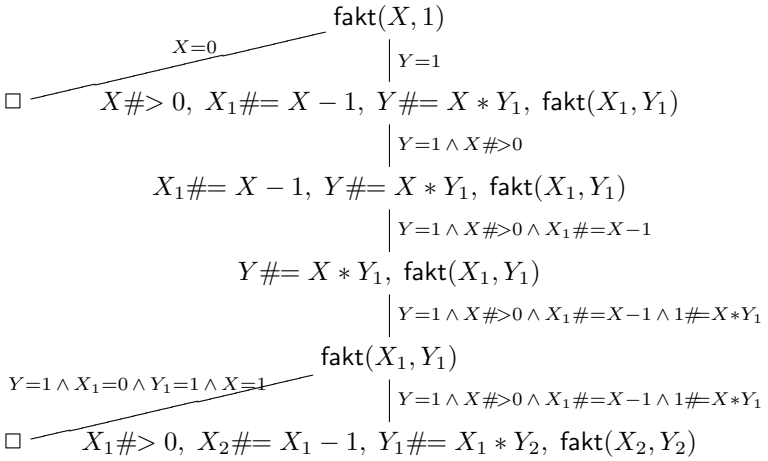
*widerspricht.*

*Stellt man dem Programm also die Anfrage "*`?- fakt(X,1)`*", so erhält man zunächst die Antworten* `X = 0` *und* `X = 1`*, da die Antwortconstraints auf diejenigen eingeschränkt werden, die die Variable* `X` *der Anfrage enthalten. Gibt der Benutzer danach noch einmal "*`;`*" ein, so terminiert das Programm nicht mehr.*

*Um dies zu verbessern, empfiehlt es sich, die beiden letzten atomaren Formeln im Rumpf der zweiten* `fakt`*-Regel zu vertauschen. Man erhält dann das Programm*

```
fakt(0,1).
fakt(X,Y) :- X #> 0, X1 #= X-1,  Y #= X*Y1, fakt(X1,Y1).
```

*Die Anfrage "*`?- fakt(X,1)`*" führt nun zu folgendem SLD–Baum:*




*Der rechte Pfad lässt sich nun nicht weiter fortsetzen, da die dann entstehende Konjunktion von Constraints*

$$Y = 1 \wedge X\#> 0 \wedge X_1\#= X - 1 \wedge 1\#= X * Y_1 \wedge X_1\#> 0$$

*widersprüchlich ist. Somit ist der SLD–Baum nun endlich und die Anfrage "*`?- fakt(X,1)`*" führt zu den Antworten* `X = 0` *und* `X = 1` *und terminiert mit "*`false`*", falls der Benutzer anschließend noch einmal "*`;`*" eingibt.*

## 6.2 Logikprogrammierung mit Constraints in Prolog

Nun zeigen wir, wie die Logikprogrammierung mit Constraints in der Programmiersprache Prolog realisiert ist. Zunächst erkennt man, dass die Prädikate "true" und "=" tatsächlich mit den gleichen Klauseln wie in Def. 6.1.5 vordefiniert sind. In einem Prolog-Programm muss aber natürlich zunächst einmal angegeben werden, welche Constraint-Theorie $CT$ verwendet werden soll. Hierdurch wird implizit auch festgelegt, welche Funktions- und Prädikatssymbole in den Mengen $\Sigma'$ und $\Delta'$ enthalten sein sollen.

Wie andere Programmiersprachen auch bieten Prolog-Implementierungen meist ein Modul-System an. Einzelne Programm-Bibliotheken sind dann in unterschiedlichen Modulen abgelegt. Das Prädikat `use_module` dient dazu, die Prädikate eines Moduls zu importieren. Um beispielsweise die Bibliothek mit den Prädikatssymbolen der Constraint-Theorie $CT_{FD}$ aus Bsp. 6.1.2 zu importieren, muss das Prolog-Programm die folgende Direktive enthalten:

```
:- use_module(library(clpfd)).
```

Danach stehen die Symbole aus $\Sigma_{FD}$ und $\Delta_{FD}$ zur Verfügung (vgl. Bsp. 6.1.2) und die zugrundeliegende Constraint-Theorie ist $CT_{FD}$. Man kann nun also in der Tat das `fakt`-Programm aus Bsp. 6.1.6 schreiben und die Antworten auf die Anfragen "`?- fakt(1,Z).`" und "`?- fakt(X,1).`" sind `Z = 1` bzw. `X = 0` und `X = 1`.

Ein Problem ist jedoch, dass die Folgerbarkeit aus den Axiomen der Constraint-Theorie oftmals unentscheidbar ist. So ist es z.B. bei der Theorie $CT_{FD}$ nicht entscheidbar, ob für eine Konjunktion von Constraints $CO$

$$CT_{FD} \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\, CO \tag{6.5}$$

gilt. Selbst bei anderen Constraint-Theorien, bei denen diese Frage entscheidbar ist, kann das dazugehörige Entscheidungsverfahren so aufwändig sein, dass man es nicht sinnvoll in der Implementierung der Logikprogrammierung mit Constraints einsetzen kann.

Aus diesem Grund werden in realen Implementierungen der Logikprogrammierung mit Constraints meist Techniken verwendet, die die Frage (6.5) *approximieren*. Hierbei kann es sein, dass die Approximation behauptet, dass (6.5) gilt, obwohl dies nicht der Fall ist. Im Fall der Constraint-Theorie $CT_{FD}$ wird meist die sogenannte *Pfadkonsistenz* verwendet.

**Definition 6.2.1 (Pfadkonsistenz)** *Sei $CO = \varphi_1 \wedge \ldots \wedge \varphi_m$ eine Konjunktion von Constraints mit $\varphi_i \in \mathcal{A}t(\Sigma_{FD}, \Delta_{FD}, \mathcal{V})$. Seien $X_1, \ldots, X_n$ die Variablen in $CO$ und seien $D_1, \ldots, D_n$ jeweils Teilmengen von $\mathbb{Z}$. Wir sagen, dass $D_1, \ldots, D_n$* zulässige Domains *für die Variablen $X_1, \ldots, X_n$ bzgl. $CO$ sind, falls für alle Constraints $\varphi_i$ mit $1 \leq i \leq m$ und alle Variablen $X_j$ mit $1 \leq j \leq n$ gilt: Für alle $a_j \in D_j$ existieren $a_1 \in D_1, \ldots, a_{j-1} \in D_{j-1}, a_{j+1} \in D_{j+1}, \ldots, a_n \in D_n$ so dass $CT_{FD} \models \varphi_i[X_1/a_1, \ldots, X_n/a_n]$. Falls es zulässige Domains $D_1, \ldots, D_n$ gibt, die alle nicht-leer sind, so heißt $CO$* pfadkonsistent.

Der wesentliche Unterschied zwischen der Pfadkonsistenz von $CO$ und der "wirklichen Konsistenz" (d.h. der Frage, ob $CT_{FD} \models \exists\, CO$ gilt), ist, dass bei der Pfadkonsistenz die Constraints separat voneinander betrachtet werden. In Prolog-Implementierungen geht man dabei meist so vor, dass man zunächst alle $D_j$ auf $\mathbb{Z}$ setzt. Anschließend durchläuft man sukzessive alle Constraints und alle Variablen und reduziert die jeweiligen Domains. Dies wird solange durchgeführt, bis sich die Domains $D_j$ nicht mehr ändern.

**Beispiel 6.2.2** *Betrachten wir die folgende Formel CO*

$$X_1 \#> 5 \ \wedge \ X_1 \#< X_2 \ \wedge \ X_2 \#< 9$$

*Am Anfang ist* $D_1 = \mathbb{Z}$ *und* $D_2 = \mathbb{Z}$*. Wir betrachten zunächst den ersten Constraint* $X_1 \#> 5$*. Wir müssen nun alle Elemente aus* $D_1$ *löschen, bei denen dieser Constraint nicht erfüllbar ist. Dies führt zu* $D_1 = \{6, 7, \ldots\}$*.*

*Jetzt betrachten wir den zweiten Constraint. Wir beginnen mit der Variable* $X_1$ *und löschen alle Elemente* $a_1 \in D_1$*, für die es kein* $a_2 \in D_2$ *gibt, so dass* $a_1 < a_2$ *ist. Solche Elemente gibt es allerdings nicht. Nun nehmen wir die Variable* $X_2$ *und löschen alle Elemente* $a_2 \in D_2$*, für die es kein* $a_1 \in D_1$ *gibt, so dass* $a_1 < a_2$ *gilt. Dies führt zu* $D_2 = \{7, 8, \ldots\}$*.*

*Jetzt untersuchen wir den dritten Constraint. Dieser führt zu* $D_2 = \{7, 8\}$*. Man wiederholt nun die Untersuchungen der Constraints, bis sich nichts mehr ändert. Die Betrachtung des zweiten Constraints für die Variable* $X_1$ *ergibt noch* $D_1 = \{6, 7\}$*. In der Tat erhält man bei der Anfrage*

```
?- X1 #> 5, X1 #< X2, X2 #< 9.
```

*die Antwortconstraints*

```
X1 in 6 .. 7, X1 #< X2, X2 in 7 .. 8
```

*Diese Schreibweise mit dem vordefinierten Prädikat* `in` *ist gleichbedeutend zu*

```
6 #=< X1, X1 #=< 7, X1 #< X2, 7 #=< X2, X2 #=< 8
```

*Das Prädikat* `in` *kann auch vom Programmierer in Programmen und Anfragen verwendet werden. Hierbei stehen* `inf` *und* `sup` *für* $-\infty$ *und* $\infty$*. Wir könnten unsere Anfrage also auch wie folgt umformulieren:*

```
?- X1 in 6 .. sup, X1 #< X2, X2 in inf .. 8.
```

*Es gibt auch ein Prädikat* `label`*, mit dem man erzwingen kann, dass* Prolog *die möglichen Lösungen nach und nach aufzählt. Hierbei muss* `label` *als Argument die Liste der Variablen bekommen, für deren Werte man sich interessiert. Die Anfrage*

```
?- X1 #> 5, X1 #< X2, X2 #< 9, label([X1,X2]).
```

*liefert daher*

```
X1 = 6, X2 = 7 ;
X1 = 6, X2 = 8 ;
X1 = 7, X2 = 8
```

*Hierbei müssen aber die Domains für die Variablen im Argument von* `label` *tatsächlich endlich sein. Ansonsten führt die Anfrage mit* `label` *zu einem Programmfehler. Dies würde also z.B. bei der folgenden Anfrage auftreten:*

```
?- X1 #> 5, X1 #< X2, label([X1,X2]).
```

Es existieren aber auch Beispiele, bei denen die Constraints pfadkonsistent sind, obwohl sie widersprüchlich sind.

**Beispiel 6.2.3** *Das einfachste solche Beispiel ist*

```
?- X1 #> X2, X1 #=< X2.
```

*Die Überprüfung der Pfadkonsistenz zeigt, dass es für jeden Wert $a_1 \in D_1 = \mathbb{Z}$ einen Wert $a_2 \in D_2 = \mathbb{Z}$ gibt, so dass der erste Constraint erfüllt ist und ebenso gibt es für jeden Wert $a_2 \in D_2 = \mathbb{Z}$ einen Wert $a_1 \in D_1 = \mathbb{Z}$ gibt, so dass der erste Constraint erfüllt ist. Dasselbe gilt für den zweiten Constraint. Somit ergibt sich als Antwort wieder*

```
X1 #> X2, X1 #=< X2.
```

Zum Abschluss wollen wir zwei typische Beispiele für Programme betrachten, die sich sehr gut als Logikprogramm mit Constraints formulieren lassen. Das erste Beispiel benutzt wieder die Constraint-Theorie $CT_{FD}$.

**Beispiel 6.2.4** *Wir programmieren das n-Damen-Problem (unser Programm stammt im wesentlichen von [NM96]). Hierbei hat man ein Schachbrett der Größe $n \times n$ und das Ziel ist, auf dem Schachbrett $n$ Damen zu platzieren, die sich gegenseitig nicht schlagen können. Dies bedeutet, dass es keine Zeile, keine Spalte und keine Diagonale geben darf, in der mehr als eine Dame steht.*

*Wir repräsentieren die Positionen der Damen als Liste $[x_1, \ldots, x_n]$, wobei die Zahl $x_i$ die Nummer der Zeile ist, in der die Dame der $i$-ten Spalte steht. Die Positionen der Damen sind also $(x_1, 1), \ldots, (x_n, n)$. Die Liste $[x_1, \ldots, x_n]$ ist also eine Permutation der Zahlen $[1, \ldots, n]$. Um herauszufinden, wie man $n$ Damen auf einem $n \times n$ Brett platzieren kann, ruft man*

```
?- queens(n,L).
```

*auf. Die atomare Formel mit dem vordefinierten Prädikat* `length` *stellt sicher, dass die Ergebnisliste* `L` *die Länge $n$ hat. Das Prädikat* `ins` *aus der Bibliothek* `clpfd` *ist ähnlich wie* `in`*, aber es stellt sicher, dass* alle Elemente *der Liste* `L` *aus dem vorgegebenen Bereich stammen. Es gilt also "$[x_1, \ldots, x_n]$* `ins 1 .. N`*" falls "$x_i$* `in 1 .. N`*" für alle $1 \leq i \leq n$ gilt. Das vordefinierte Prädikat* `all_different` *aus der Bibliothek* `clpfd` *stellt sicher, dass alle Elemente von* `L` *paarweise verschieden sind. Hierbei stehen also die* `ins`*- und* `all_different`*-Formeln wieder für Abkürzungen für (naheliegende) Konjunktionen von Constraints.*

```
queens(N,L) :- length(L, N),
               L ins 1 .. N,
               all_different(L),
               safe(L),
               label(L).

safe([]).
safe([X|Xs]) :- safe_between(X, Xs, 1),
                safe(Xs).

safe_between(X, [], M).
```

```
safe_between(X, [Y|Ys], M) :- no_attack(X, Y, M),
                              M1 #= M + 1,
                              safe_between(X, Ys, M1).

no_attack(X, Y, N) :- X+N #\= Y, X-N #\= Y.
```

*Das Prädikat* `safe` *dient dazu, sicherzustellen, dass die Damen in der Liste* `L` *nicht auf gleichen Diagonalen sind. Wir werden es gleich anschließend genauer erklären. Schließlich dient* `label(L)` *dazu, die einzelnen Elemente von* `L` *auszugeben.*

*Die atomare Formel* `safe(L)` *stellt sicher, dass keine Dame in* `L` *eine Dame schlagen kann, die* rechts von ihr *steht. Man benutzt hier das Hilfsprädikat* `safe_between`*, wobei* `safe_between(X, L, M)` *wahr ist, falls die Dame in Zeile* `X` *keine Dame in den Spalten aus* `L` *schlagen kann, sofern* `M` *der Abstand der Spalte von* `X` *zu der ersten Spalte aus* `L` *ist. Daher trifft* `no_attack(X, Y, N)` *zu, falls die Dame in Zeile* `X` *und die Dame in Zeile* `Y`*, die* `N` *Spalten weiter rechts ist, nicht auf derselben Diagonale sind. Beispielsweise erhält man*

```
?- queens(4,L).

L = [2, 4, 1, 3] ;
L = [3, 1, 4, 2]
```

Als nächstes Beispiel betrachten wir eine andere Constraint-Theorie für Constraints über reellen Zahlen.

**Beispiel 6.2.5** *Wir behandeln nun eine Constraint-Signatur* $(\Sigma, \Delta, \Sigma', \Delta')$*, bei der* $\Sigma'$ *und* $\Delta'$ *Funktions- und Prädikatssymbole zur Verarbeitung reeller Zahlen enthalten. Um diese Symbole von den (z.T. gleich lautenden) Symbolen aus* $\Sigma \setminus \Sigma'$ *und* $\Delta \setminus \Delta'$ *zu unterscheiden, nehmen wir diesmal nicht neue Namen wie "*$\#>$*", sondern wir vereinbaren, dass alle Constraints mit Prädikaten aus* $\Delta'$ *in geschweifte Klammern geschrieben werden. Die Constraint-Theorie* $CT_R$ *enthält dann alle entsprechenden wahren Formeln über* $\mathbb{R}$*. Um die entsprechende Bibliothek zu importieren, benötigt man die Direktive*

```
:- use_module(library(clpr)).
```

*Als Beispiel wollen wir nun ein Programm schreiben, das zur Zins- und Rückzahlungsberechnung bei Krediten verwendet werden kann [FA10]. Hierbei ist* `mortgage(D,T,I,R,S)` *wahr, falls*

- `D` *der Betrag ist, den man als Kredit aufgenommen hat (Debt)*
- `T` *die Dauer (in Monaten) ist, seit der man den Kredit schon aufgenommen hat*
- `I` *der Zinssatz ist, den man pro Monat zahlen muss (Interest)*
- `R` *die Rückzahlung ist, die man pro Monat leisten muss*
- `S` *der Betrag an Schulden ist, den man nach* `T` *Monaten noch hat*

*Das Programm lautet dann wie folgt:*

```
mortgage(D, T, I, R, S) :- {T = 0, D = S}.
mortgage(D, T, I, R, S) :- {T > 0, T1 = T - 1, D1 = D + D * I - R},
                           mortgage(D1, T1, I, R, S).
```

*Falls noch kein Monat vergangen ist, so sind die verbleibenden Schulden* `S` *gerade der Betrag* `D`*, den man als Kredit aufgenommen hat. Ansonsten erhöhen sich aufgrund der Zinsen pro Monat die Schulden um* `D * I` *und jeden Monat verringern sie sich um die Rückzahlung* `R`*.*

*Man kann also nun fragen, wie groß die Schulden nach 30 Jahren (d.h. 360 Monaten) noch sind, wenn man ursprünglich 100000 Euro Kredit aufgenommen hatte, der Zinssatz pro Monat 1 % beträgt und man jeden Monat 1025 Euro zurück zahlt.*

```
?- mortgage(100000, 360, 0.01, 1025, S).

S = 12625.9
```

*Man sieht, dass man inzwischen schon 360 * 1025 = 369000 Euro zurückgezahlt hat und dennoch immer noch eine Restschuld von 12625.9 Euro hat. Dies demonstriert den Effekt der Zinsen.*

*Aufgrund der Bidirektionalität der Logikprogrammierung (und der Tatsache, dass die Prädikate in den Constraints (im Unterschied zu den eingebauten arithmetischen Prädikaten) tatsächlich bidirektional arbeiten), kann man in den Anfragen beliebige Argumente vorgeben und andere berechnen lassen. So kann man z.B. fragen, wie hoch der Kredit denn sein dürfte, damit man ihn bei diesen Konditionen nach 30 Jahren abbezahlt hat.*

```
?- mortgage(D, 360, 0.01, 1025, 0).

D = 99648.8
```

*Um zu fragen, wie lange wir bei unserem ursprünglichen Kredit von 100000 Euro noch zurückzahlen müssen, könnte man folgende Anfrage stellen:*

```
?- {S =< 0}, mortgage(100000, T, 0.01, 1025, S).

S = -807.964, T = 374.0 ;
S = -1841.04, T = 375.0 ;
S = -2884.45, T = 376.0 ;
S = -3938.3,  T = 377.0 ;
...
```

*Dies bedeutet, dass man also nach 374 Monaten eine "Schuld" von -807,964 Euro hätte. Im letzten Monat muss man also keine volle Rate von 1025 Euro mehr zurückzahlen. (Die weiteren Antworten sind zwar korrekt, aber nicht wirklich beabsichtigt.)*

*Wenn man wissen will, wie der Zusammenhang zwischen dem ursprünglichen Kredit* `D` *und der monatlichen Rückzahlung* `R` *ist (falls man nach 30 Jahren seine Schuld abbezahlt haben will), dann kann man folgende Anfrage stellen:*

```
?- mortgage(D,360,0.01,R,0).

{R=0.0102861*D}
```

Diese Beispiele illustrieren, dass die Ergänzung der Logikprogrammierung um Constraints tatsächlich eine sehr sinnvolle und nützliche Erweiterung der Logikprogrammierung darstellt.

# Literaturverzeichnis


[Apt97] K. R. Apt. *From Logic Programming to Prolog*, Prentice Hall, 1997.

[Bra11] I. Bratko. *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 2011.

[CM13] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*, Springer, 2013.

[FA10] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*, Springer, 2010.

[Gie11] J. Giesl. *Termersetzungssysteme*, Skript zur Vorlesung, RWTH Aachen, 4. Auflage, 2011.

[Gie14] J. Giesl. *Funktionale Programmierung*, Skript zur Vorlesung, RWTH Aachen, 7. Auflage, 2014.

[Gra11] E. Grädel. *Mathematische Logik*, Skript zur Vorlesung, RWTH Aachen, 2011.

[Han87] M. Hanus. *Problemlösen mit Prolog*, Teubner, 1987.

[Kle52] S. C. Kleene. *Introduction to metamathematics*. North-Holland, 1952.

[Llo13] J. W. Lloyd. *Foundations of Logic Programming*, Springer, 2013.

[MS98] K. Marriott and P. J. Stuckey. *Programming with Constraints*, MIT Press, 1998.

[NM96] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog*, Wiley, 1996.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23-41, 1965.

[Sch92] P. H. Schmitt. *Theorie der logischen Programmierung*, Springer, 1992.

[Sch00] U. Schöning. *Logik für Informatiker*, Spektrum Akademischer Verlag, 2000.

[Sch08] U. Schöning. *Theoretische Informatik – kurzgefasst*, Spektrum Akademischer Verlag, 2008.

[SS00] L. Sterling and E. Shapiro. *The Art of Prolog*, MIT Press, 2000.

# Index