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 [FA03, 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 (Σ, Δ) um Constraints über einer Teilsignatur (Σ', Δ') zu erweitern. Hierbei erweist es sich als sinnvoll, das Gleichheits-Prädikatssymbol "=" sowie die speziellen Prädikatssymbole true und fail gesondert zu behandeln.

Definition 6.1.1 (Constraint-Signatur und Constraints) *Sei (Σ, Δ) 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 Δ' keines der Prädikatssymbole true, 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 Σ' der Funktionssymbole und eine Teilmenge Δ' der Prädikatssymbole, aus denen die Constraints gebildet werden. Hierbei dürfen Prädikate aus Δ' nur auf Terme angewendet werden, die keine Funktionssymbole außer denjenigen aus Σ' 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 Contraint-Signatur* $(\Sigma, \Delta, \Sigma', \Delta')$*, bei der* Σ' *und* Δ' *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 Σ' *und* Δ' *auch als* Σ_{FD} *und* Δ_{FD}*. Hierbei steht "FD" für Finite Domains. Falls* $\mathsf{f} \in \Sigma_1$*, so ergeben sich unter anderem folgende Constraints:*

$$\begin{aligned}
X + Y &\ \#> \ Z * 3 \\
\mathsf{max}(X, Y) &\ \#= \ X \bmod 2 \\
\mathsf{f}(X) + 2 &\ = \ Y + Z
\end{aligned}$$

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* α_+ *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:*

$$\varphi \in CT_{FD} \quad gdw. \quad S_{FD} \models \varphi$$

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* Σ_{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 true und $=$ enthalten und es keine Klausel zur Definition von fail gibt. Auf den linken Seiten anderer Regeln dürfen die Prädikatssymbole der Constraints nicht auftreten. Außerdem dürfen die Prädikatssymbole aus Δ' nur auf Terme mit Funktionssymbolen aus Σ' 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, σ) 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*

$$\begin{aligned} & (\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U),\ \varnothing) \\ \vdash_{\mathcal{P}}\ & (\neg\mathsf{add}(\mathsf{s}(0), 0, Z),\ \{X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(Z)\}) \\ \vdash_{\mathcal{P}}\ & (\square,\ \underbrace{\{X'/\mathsf{s}(0),\ Z/\mathsf{s}(0)\} \circ \{X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(Z)\}}_{\{X'/\mathsf{s}(0),\ Z/\mathsf{s}(0),\ X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(\mathsf{s}(0))\}}) \end{aligned}$$

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 true*, die stets wahr ist.*

$$\begin{aligned} & (\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U),\ \mathsf{true}) \\ \vdash_{\mathcal{P}}\ & (\neg\mathsf{add}(X, Y, Z),\ \overline{\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U) = \mathsf{add}(X, \mathsf{s}(Y), \mathsf{s}(Z))}) \\ \vdash_{\mathcal{P}}\ & (\square,\ \overline{\mathsf{add}(X, Y, Z) = \mathsf{add}(X', 0, X')} \wedge \overline{\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U) = \mathsf{add}(X, \mathsf{s}(Y), \mathsf{s}(Z))}) \end{aligned}$$

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, σ) betrachten, bei denen σ 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* fail*, falls $A = p(\ldots)$, $B = q(\ldots)$ mit $p \neq q$*
- $\overline{A = B}$ *ist die Formel* 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:*

$$\begin{aligned}
& (\neg\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U),\ \mathsf{true}) \\
\vdash_{\mathcal{P}}\ & (\neg\mathsf{add}(X,Y,Z),\ \underbrace{\overline{\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U) = \mathsf{add}(X,\mathsf{s}(Y),\mathsf{s}(Z))}}_{\mathsf{s}(0)=X\,\wedge\,\mathsf{s}(0)=\mathsf{s}(Y)\,\wedge\,U=\mathsf{s}(Z)}) \\
\vdash_{\mathcal{P}}\ & (\square,\ \underbrace{\overline{\mathsf{add}(X,Y,Z) = \mathsf{add}(X',0,X')}}_{X=X'\,\wedge\,Y=0\,\wedge\,Z=X'} \wedge\ \mathsf{s}(0) = X\ \wedge\ \mathsf{s}(0) = \mathsf{s}(Y)\ \wedge\ U = \mathsf{s}(Z))
\end{aligned}$$

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 true, fail und $=$. Man sollte bei der Vereinfachung von CO daher die Axiome über true, 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 φ mit den Variablen $X_1, \ldots, X_n$ bezeichnet $\forall\,\varphi$ den *Allabschluss* von φ, d.h. die Formel $\forall X_1, \ldots, X_n\,\varphi$. Analog dazu ist $\exists\,\varphi$ der *Existenzabschluss* von φ, 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 σ 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 $\alpha_f = f$, $\alpha_= = \{(r, r) \mid r \in \mathcal{T}(\Sigma, \mathcal{V})\}$ und $\beta(X) = X$. Hier gilt also $I(r) = r$ für alle Terme r und $I \models r_1 = r_2$ gilt gdw. 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 σ, 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 σ 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 true, fail und $=$ enthalten, sondern es sind nun auch Constraints möglich, die mit Prädikatssymbolen aus Δ' gebildet werden. Entsprechend muss man nun auch die Axiome CT zusätzlich zu den Axiomen $\forall X\ X = X$ und 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* ν *mit* $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ *und* $n \geq 0$*, so dass*
    - G_1 *und* $\nu(K)$ *keine gemeinsamen Variablen haben*
    - $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO_1 \wedge \overline{A_i = B}$
  - $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}$
  - $CO_2 = CO_1 \wedge \overline{A_i = B}$

(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, 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 Contraint-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 ν 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 true, fail und $=$ nicht enthält. Nach Lemma 6.1.13 (b) folgt also, dass σ 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 Indukionsschritt $l > 1$. 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 $\mathcal{P} \models C_1 \wedge \ldots \wedge C_n \rightarrow B$ 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 Δ' 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 σ 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

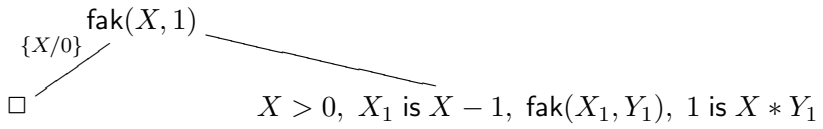
$$(\{\neg A_{j+1}, \ldots, \neg A_k\}, CO_1) \vdash_{\mathcal{P}}^{+} (\Box, CO_1 \wedge CO_2),$$

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 $CT \cup \cup\{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO_1 \wedge CO_2)$. 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.

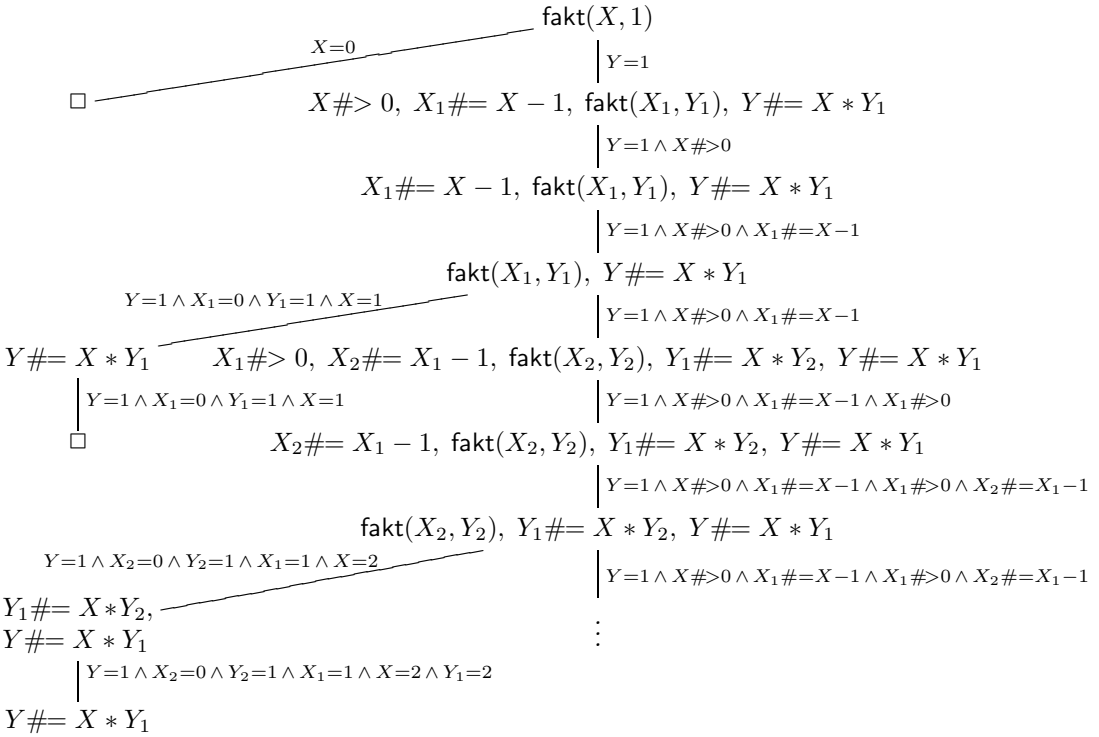
Um die dadurch entstehenden SLD–Bäume zu illustrieren, betrachten wir noch einmal das `fak-` und das `fakt`-Programm aus Bsp. 6.1.6.

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 enstehende 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*

$$Y = 1 \wedge X_2 = 0 \wedge Y_2 = 1 \wedge X_1 = 1 \wedge X = 2 \wedge Y_1 = 2$$

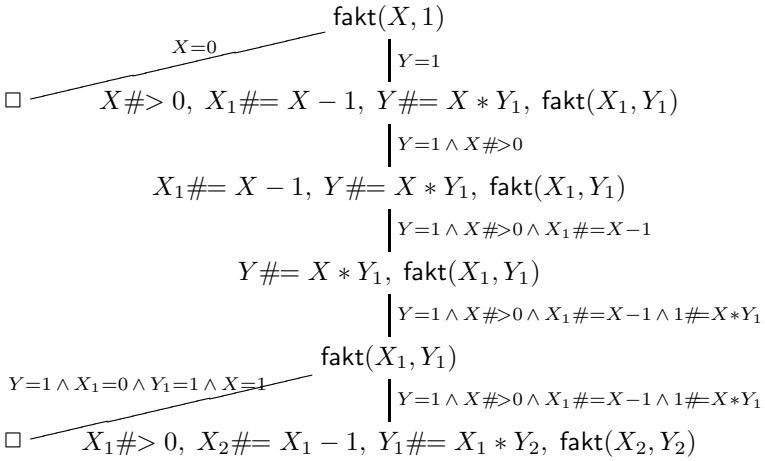
widerspricht.

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 "*`No`*", falls der Benutzer anschließend noch einmal "*`;`*" eingibt.*

6.2 Logikprogrammierung mit Constraints in Prolog

Nun zeigen wir, wie die Logikprogrammierung mit Constraint 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 Σ' und Δ' enthalten sein sollen.

Wie andere Programmiersprachen auch bieten Prolog-Implementierungen meist ein Modul-System an. Einzeln 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 Σ_{FD} und Δ_{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

$$CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO \tag{6.5}$$

gilt. Selbst bei anderen Constraint-Theorien, bei denen diese Frage entscheidbar ist, kann das dazugehörige Entscheidungsverfahren so aufwendig 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 φ_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 $CT_{FD} \models CO[X_1/a_1, \ldots, X_n/a_n]$. 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. Als simplify(CO) wird dann üblicherweise eine zu

$$X_1 \in D_1 \wedge \ldots \wedge X_n \in D_n \tag{6.6}$$

äquivalente Formel ausgegeben.

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, X2 in 7 .. 8
```

Diese Schreibweise mit dem vordefinierten Prädikat `in` *ist gleichbedeutend zu*

```
6 #=< X1, X1 #=< 7, 7 #=< X2, X2 #=< 8
```

Das Prädikat `in` *kann auch vom Programmierer in Programmen und Anfragen verwendet werden. Hierbei steht* `inf` *und* `sup` *für* $-\infty$ *und* ∞. *Wir könnten unsere Anfrage also auch wie folgt umformulieren:*

```
?- X1 in 6 .. sup, X1 #< X2, X2 in inf .. 8.
```

Man erkennt, dass im Allgemeinen die Formel (6.6) nicht äquivalent zu CO ist, d.h. simplify *ist nun nicht mehr äquivalenzerhaltend. In der Tat würde in unserem Beispiel nun auch die Lösung* `X1 = 7, X2 = 7` *zugelassen werden, obwohl diese die ursprüngliche Anfrage nicht erfüllt. Es gibt allerdings 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 nicht nur Beispiele, bei denen die Simplifikation (6.6) "inkorrekt" arbeitet. Es gibt sogar 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:

```
X1 in inf .. sup, X2 in inf .. sup
```

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 Program 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* Σ' *und* Δ' *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* Δ' *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 [FA03]. 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 zurück 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.