Logikprogrammierung

A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have desired meaning.
Sterling und Shapiro: The Art of Prolog, Seite 1

Algorithm = Logic + Control

Robert A. Kowalski, 1979

Anmerkung

Logic Was Logische Komponente Problembeschreibung
Control Wie Steuerkomponente Programmablauf

imperativ, prozedural, befehlsorientiert
deklarativ, logikorientiert

Beispiel Liste sortieren:
Durch paarweises Vertauschen
Permutation + Ordnungstest

Spezifikation, Rapid Prototyping mittels logischer Formeln
allgemeines Wissen durch Regeln, konkrete Annahmen als Fakten; Abfragen=Problembeschreibung, mittels Folgerungstest

Beispiel für ein Logik-Programm

$\{$`kante(a,b)` $\leftarrow\ \top$,
`kante(a,c)` $\leftarrow\ \top$,
`kante(b,d)` $\leftarrow\ \top$,
`kante(c,d)` $\leftarrow\ \top$,
`kante(d,e)` $\leftarrow\ \top$,

`verbindung(Start,Ziel)` $\leftarrow$
    `kante(Start,Ziel)`,

`verbindung(Start,Ziel)` $\leftarrow$
    `kante(Start,Knoten)` $\wedge$
    `verbindung(Knoten,Ziel)`$\}$

Constraint-Logikprogrammierung: Motivation

Krypto-arithmetisches Rätsel

$$
\begin{array}{cccccc}
 & & S & E & N & D \\
+ & & M & O & R & E \\
\hline
= & M & O & N & E & Y
\end{array}
$$

Implementierung

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,R,Y]) :-
     gen_domains([S,E,N,D,M,O,R,Y],0..9),
     all_distinct([S,E,N,D,M,O,R,Y]),
                  1000*S + 100*E + 10*N + D
     +            1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y,
     S #\= 0, M #\= 0,
     labeling([],[S,E,N,D,M,O,R,Y]).
```

Lösung

`[S,E,N,D,M,O,R,Y] = [9,5,6,7,1,0,8,2]`

$$\begin{array}{cccccc} & & 9 & 5 & 6 & 7 \\ + & & 1 & 0 & 8 & 5 \\ \hline = & 1 & 0 & 6 & 5 & 2 \end{array}$$

Testbeispiele für `send` ohne Labelling

```
:- send([S,E,N,D,M,O,R,Y]).

M = 1,
O = 0,
S = 9,
E in 4..7,
N in 5..8,
D in 2..8,
R in 2..8,
Y in 2..8 ?
```

```
:- send([9,4,N,D,M,O,R,Y]).

no
```

Grund: `N = 5` und `R = 8` impliziert, daß für `D` keinen möglichen Wert mehr gibt.

```
:- send([9,5,N,D,M,O,R,Y]).

D = 7,
M = 1,
N = 6,
O = 0,
R = 8,
Y = 2
```

Anmerkung

CLP-Sprachen vereinigen die Vorteile von Logikprogrammiersprachen (deklarativ, für beliebige Prädikate, nichtdeterministisch) mit denen von Constraintlösern (deklarativ, effizient für spezielle Prädikate, deterministisch). Die Kombination von Suche mit Constraintbehandlung ist besonders nützlich für kombinatorische Probleme (exponentielle Komplexität haben, d. h. sehr schnell praktisch unlösbar). Die erhöhte Deklarativität verringert den Programmieraufwand ohne die Effizienz zu vernachlässigen.

Abstrakte CLP-Syntax

Atome:	A, B	$::=$	$p(t_1, \ldots, t_n),\ n \geq 0$
Constraints:	C, D	$::=$	$c(t_1, \ldots, t_n) \mid C \land D,\ n \geq 0$
Ziele:	G, H	$::=$	$\top \mid \bot \mid A \mid C \mid G \land H$
CL-Klauseln:	K	$::=$	$A \leftarrow G$
CL-Programme:	P, Q	$::=$	$\{K_1, \ldots, K_m\},\ m \geq 0$

Anmerkung

Wir verwenden zur Definition der abstrakten Syntax neben einer verbalen Definition auch die Darstellung in einer erweiterten Backus-Naur-Form (BNF). Eine Produktionsregel

Name: $G, H ::= A \mid B$, Bedingung

bedeutet, daß die Symbole G bzw. H (eventuell mit Indizes, z.B. G_2, oder gestrichen, z.B. G') „Name" genannt werden und von der Form A oder von der Form B sein können, falls die nachfolgende Bedingung erfüllt ist. Großbuchstaben stehen dabei für Ausdrücke, die wiederum durch Produktionsregeln definiert sind.

Abstrakte CLP-Syntax

Definition 1 Ein *Atom* ist ein Ausdruck der Form $p(t_1, \ldots, t_n)$, wobei p ein n-stelliges Symbol aus $\mathcal{P}_p$ und $t_1, \ldots, t_n$ Terme sind.

Ein *Constraintatom* ist ein Ausdruck der Form $c(t_1, \ldots, t_n)$, wobei c ein n-stelliges Symbol aus $\mathcal{P}_c$ ist und $t_1, \ldots, t_n$ Terme sind.

Ein *Constraint* ist entweder ein Constraintatom oder eine Konjunktion von Constraints.

Ein *Zielatom* ist entweder ein Atom oder ein Constraintatom.

Ein *Ziel* ist entweder $\top$ oder $\bot$ oder ein Zielatom oder eine Konjunktion von Zielen.

Abtrakte CLP-Syntax

Definition 2 Eine *(CL-)Klausel* hat die Form $A \leftarrow G$, wobei wir das Atom A als *Kopf* (engl. head) und das Ziel G als *Rumpf* (engl. body) der Klausel bezeichnen.

Klauseln von der Form $A \leftarrow \top$ werden *Fakten* genannt, alle anderen *Regeln*.

Ein *Constraint-Logikprogramm* ist eine endliche Menge von CL-Klauseln.

Logische Kalküle

Definition 3 Ein *(logischer) Kalkül* ist ein Tripel $(\Sigma, \equiv, (\mathcal{Z}, \mathcal{T}, \mapsto))$, wobei gilt:

- Σ ist eine Signatur für eine Sprache der Prädikatenlogik erster Stufe.
- $\equiv$ ist eine Kongruenz über Formeln (und Zuständen) über der Signatur Σ.
- $(\mathcal{Z}, \mathcal{T}, \mapsto)$ ist ein Zustandsübergangssystem.

Zustände

Definition 4 Ein *Zustand* ist ein Paar $\langle G, C\rangle$, wobei G ein Ziel ist und C ein Constraint. G wird Zielspeicher genannt und C Constraintspeicher.

Ein *Anfangszustand* ist ein Zustand von der Form $\langle G, \mathit{true}\rangle$.

Ein Zustand heißt *erfolgreicher Endzustand*, wenn er von der Form $\langle \top, C\rangle$ ist und C ungleich *false* ist.

Ein Zustand heißt *erfolgloser Endzustand* (gescheitert), wenn er die Form $\langle G, \mathit{false}\rangle$ hat.

Anmerkung

Intuitiv repräsentiert G die Ziele, die noch zu lösen sind, und C die bereits aufgetretenen, vereinfachten Constraints.

Kalküle - Kongruenz

Definition 5 Eine *Äquivalenzrelation* $\equiv$ ist eine Relation, die reflexiv, symmetrisch und transitiv ist:

(Reflexivität)	$A \equiv A$
(Symmetrie)	Wenn $A \equiv B$ dann auch $B \equiv A$
(Transitivität)	Wenn $A \equiv B$ und $B \equiv C$ dann auch $A \equiv C$

Definition 6 Eine *Kongruenz* $\equiv$ *über Formeln* ist eine Äquivalenzrelation, für die zusätzlich gilt: Wenn $\varphi \equiv \psi$ dann auch $\neg\varphi \equiv \neg\psi$ und wenn $\varphi \equiv \psi$ und $\varphi' \equiv \psi'$, dann auch $\varphi \rightarrow \varphi' \equiv \psi \rightarrow \psi'$.

Eine *Kongruenz* $\equiv$ *über Zuständen* ist eine Kongruenz, für die zusätzlich gilt: Wenn $s_i \equiv t_i$ für alle $i = 1, \ldots, n$, dann auch $\langle s_1, \ldots, s_n \rangle \equiv \langle t_1, \ldots, tn \rangle$.

Anmerkung

Wenn also Teilformeln zueinander kongruent sind, sind auch die daraus zusammengesetzten Formeln kongruent. Eine Kongruenz wird hier dazu verwendet, um triviale Äquivalenzen von Formeln (z.B. zwischen $A \wedge B$ und $B \wedge A$) implizit ausnützen zu können. Wir erweitern die Kongruenz über Formeln auf eine Kongruenz über Zuständen wie folgt: Zwei Zustände sind kongruent, wenn ihre Komponenten kongruent sind. Ist ein Zustandsübergang mit bestimmten Zuständen möglich, so ist er auch mit allen dazu kongruenten Zuständen erlaubt.

Kongruenz für CLP-Kalkül

Kommutativität:	$G_1 \wedge G_2$	$\equiv$	$G_2 \wedge G_1$
Assoziativität:	$G_1 \wedge (G_2 \wedge G_3)$	$\equiv$	$(G_1 \wedge G_2) \wedge G_3$
Identität:	$G \wedge \top$	$\equiv$	G
Absorption:	$G \wedge \bot$	$\equiv$	$\bot$

Anmerkung

Die Kongruenz $\equiv$ besagt, daß es gleichgültig ist, in welcher Reihenfolge und Klammerung die Atome in einer Konjunktion auftreten. Darüber hinaus ist $\top$ in einer Konjunktion überflüssig, während $\bot$ alle anderen Konjunkte überflüssig macht. Wir werden für den Rest des Buches diese Kongruenz beibehalten.

Entfalten:

$$\frac{\begin{array}{c}(B \leftarrow H) \text{ aus } P \\ CT \models \exists((B \doteq A) \wedge C)\end{array}}{\langle A \wedge G, C\rangle \mapsto_{Entfalten} \langle H \wedge G, (B \doteq A) \wedge C\rangle}$$

Scheitern:

$$\frac{\begin{array}{c}\text{Es gibt keine Klausel } B \leftarrow H \text{ aus } P \text{ mit} \\ CT \models \exists((B \doteq A) \wedge C)\end{array}}{\langle A \wedge G, C\rangle \mapsto_{Scheitern} \langle \perp, \mathit{false}\rangle}$$

Vereinfachen:

$$\frac{CT \models (C \wedge D_1) \leftrightarrow D_2}{\langle C \wedge G, D_1\rangle \mapsto_{Vereinfachen} \langle G, D_2\rangle}$$

Anmerkung

$B \doteq A$, d. h. $p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n)$, steht für $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$.

In unserem einfachen CLP-Kalkül ist die (Aus-)Wahl (Selektion) der Klausel innerhalb eines Programms P und die Auswahl des Atoms A aus dem Ziel nicht vorbestimmt (Nichtdeterminismus). Während die Klauselselektion über Erfolg oder Scheitern einer Ableitung entscheiden kann, beeinflußt die Selektion des Atoms lediglich die Länge der Ableitung (im schlimmsten Fall unendlich).

Nichtdeterminismus (Auswahl von Klauseln und Zielatomen) wird nach einem festen Schema aufgelöst: Das am weitesten links stehende Zielatom eines Zieles wird selektiert (von links nach rechts). Klauseln werden in textueller Reihenfolge ausgewählt: Scheitert die Ableitung mit einer bestimmten Klausel, versucht man Rücksetzen (engl. (chronological) backtracking), d. h. Zurücknehmen der zuletzt gemachten Klauselwahl. Diese Tiefensuche führt dazu, daß eine Ableitung vollständig berechnet wird, bevor Alternativen untersucht werden. Unvollständig: erste berechnete Ableitung unendlich, sodaß andere (erfolgreiche) Ableitungen nicht mehr gefunden werden können. Programmierer kann durch die Reihenfolge der Klauseln und Literale den Programmablauf (die Steuerung) beeinflussen.

Kalküle - Berechnung

Definition 7 Eine *Berechnung* (engl. computation) ist eine Sequenz $S_0 \mapsto S_1 \mapsto S_2 \mapsto \ldots$ von Reduktionen. Wir schreiben $S \mapsto^n T$, falls es $S_0, \ldots, S_n$ gibt mit $S = S_0$, $T = S_n$ und $S_i \mapsto S_{i+1}$ $(i = 0, \ldots, n-1)$. Wir schreiben $S \mapsto^* T$, wenn es ein $n \geq 0$ gibt mit $S \mapsto^n T$.

Eine *Ableitung* (engl. derivation) ist eine Berechnung, die entweder in einem Endzustand endet oder unendlich ist.

Ableitungen

Definition 8 Eine Ableitung ist *erfolgreich*, wenn ihr Endzustand erfolgreich ist. Eine Ableitung ist *erfolglos* (*gescheitert*), wenn ihr Endzustand erfolglos (gescheitert) ist. Eine Ableitung ist *unendlich*, wenn sie keinen Endzustand hat.

Ein Ziel G ist *erfolgreich*, wenn es eine erfolgreiche Ableitung beginnend mit $\langle G, true \rangle$ hat. Ein Ziel G ist *erfolglos (endlich gescheitert)*, wenn es nur erfolglose Ableitungen beginnend mit $\langle G, true \rangle$ hat.

Antwortconstraint

Definition 9 Jedem Zustand $\langle H, C\rangle$, der in einer Ableitung beginnend mit $\langle G, \mathit{true}\rangle$ vorkommt, wird eine Formel $\exists\bar{x}(H \wedge C)$ als *logische Beschreibung* zugeordnet, wobei $\bar{x}$ die Variablen sind, die in H oder C, aber nicht in G vorkommen. Die in G vorkommenden Variablen bleiben frei.

Ein *Antwortconstraint (kurz: eine Antwort) eines Zieles* G ist die logische Beschreibung eines Endzustandes einer Ableitung, die mit $\langle G, \mathit{true}\rangle$ beginnt.

Anmerkung

Constraints als Antworten sind sinnvoll, da sie mehrere (manchmal unendlich viele) Antworten zu einer Antwort zusammenfassen (intensionale Antworten). Das Ziel $X + Y \geq 3 \wedge X + Y \leq 3$ z. B. wird zur intensionalen Antwort $X + Y \doteq 3$ vereinfacht, Variablen werden aber nicht gebunden.

CLP-Kalkül Beispiel

$\{$`kante(a,b)` $\leftarrow \top$,
`kante(a,c)` $\leftarrow \top$,
`kante(b,d)` $\leftarrow \top$,
`kante(c,d)` $\leftarrow \top$,
`kante(d,e)` $\leftarrow \top$,

`verbindung(Start,Ziel)` $\leftarrow$
    `kante(Start,Ziel)`,

`verbindung(Start,Ziel)` $\leftarrow$
    `kante(Start,Knoten)` $\wedge$
    `verbindung(Knoten,Ziel)`$\}$

Anmerkung

Zur Illustration wollen wir die Erreichbarkeit in einem gerichteten Graphen mithilfe eines Logikprogramms untersuchen.

Die Kanten des Graphen werden durch das Prädikat `kante/2` beschrieben. Die direkten und indirekten Verbindungen zwischen den Knoten des Graphen werden durch das Prädikat `verbindung/2` definiert.

CLP-Kalkül Beispiel

$\{$`kante(a,b)` $\leftarrow \top$, `kante(a,c)` $\leftarrow \top$,
`kante(b,d)` $\leftarrow \top$, `kante(c,d)` $\leftarrow \top$,
`kante(d,e)` $\leftarrow \top$,

```
verbindung(Start,Ziel) ←
      kante(Start,Ziel),
verbindung(Start,Ziel) ←
      kante(Start,Knoten) ∧
      verbindung(Knoten,Ziel)}
```

$$\begin{array}{ll} & \langle \mathtt{verbindung}(\mathtt{b},\mathtt{Y}), \mathit{true}\rangle \\ \mapsto_{Entfalten} & \langle \mathtt{kante}(\mathtt{S},\mathtt{Z}), \mathtt{S}=\mathtt{b} \wedge \mathtt{Z}=\mathtt{Y}\rangle \\ \mapsto_{Entfalten} & \langle \top, \mathtt{S}=\mathtt{b} \wedge \mathtt{Z}=\mathtt{d} \wedge \mathtt{Y}=\mathtt{d}\rangle \end{array}$$

Anmerkung

Man kann jetzt alle vom Knoten `b` aus erreichbaren Knoten `Y` bestimmen, durch das Ziel `verbindung(b,Y)`.

Wenn man die erste Regel für das Prädikat `verbindung` und eine passende Kante auswählt, dann erhält man folgende Ableitung:

Eine Antwort für das obige Programm und das Ziel `verbindung(b,Y)` ist dann `Y` $\doteq$ `d`, d. h. `d` ist ein aus `b` erreichbarer Knoten.

CLP-Kalkül Beispiel

$\{$`kante(a,b)` $\leftarrow\ \top$, `kante(a,c)` $\leftarrow\ \top$,
`kante(b,d)` $\leftarrow\ \top$, `kante(c,d)` $\leftarrow\ \top$,
`kante(d,e)` $\leftarrow\ \top$,

```
verbindung(Start,Ziel) ←
      kante(Start,Ziel),
verbindung(Start,Ziel) ←
      kante(Start,Knoten) ∧
      verbindung(Knoten,Ziel)}
```

$$
\begin{array}{ll}
 & \langle \mathtt{verbindung}(\mathtt{b},\mathtt{Y}), \mathit{true}\rangle \\
\mapsto_{E.} & \langle \mathtt{kante}(\mathtt{S},\mathtt{K}) \wedge \mathtt{verbindung}(\mathtt{K},\mathtt{Z}), \mathtt{S} = \mathtt{b} \wedge \mathtt{Z} = \mathtt{Y}\rangle \\
\mapsto_{E.} & \langle \mathtt{verbindung}(\mathtt{K},\mathtt{Z}), \mathtt{S} = \mathtt{b} \wedge \mathtt{Z} = \mathtt{Y} \wedge \mathtt{K} = \mathtt{d}\rangle \\
\mapsto_{E.} & \langle \mathtt{kante}(\mathtt{K},\mathtt{Z}), \mathtt{S} = \mathtt{b} \wedge \mathtt{Z} = \mathtt{Y} \wedge \mathtt{K} = \mathtt{d}\rangle \\
\mapsto_{E.} & \langle \top, \mathtt{S} = \mathtt{b}, \mathtt{Z} = \mathtt{e} \wedge \mathtt{K} = \mathtt{d} \wedge \mathtt{Y} = \mathtt{e}\rangle
\end{array}
$$

Anmerkung

Wenn aber anstatt der ersten Regel für `verbindung` die zweite Regel ausgewählt wird, erhält man folgende Ableitung:

Eine zweite Antwort ist $\mathtt{Y} \doteq \mathtt{e}$.

CLP-Kalkül Beispiel

$\{$`kante(a,b)` $\leftarrow\ \top$, `kante(a,c)` $\leftarrow\ \top$,
`kante(b,d)` $\leftarrow\ \top$, `kante(c,d)` $\leftarrow\ \top$,
`kante(d,e)` $\leftarrow\ \top$,

```
verbindung(Start,Ziel) ←
      kante(Start,Ziel),
verbindung(Start,Ziel) ←
      kante(Start,Knoten) ∧
      verbindung(Knoten,Ziel)}
```

$$
\begin{array}{ll}
 & \langle \mathtt{verbindung}(\mathtt{f},\mathtt{g}), \mathit{true}\rangle \\
\mapsto_{E.} & \langle \mathtt{kante}(\mathtt{S},\mathtt{Z}), \mathtt{S}=\mathtt{f}\wedge \mathtt{Z}=\mathtt{g}\rangle \\
\mapsto_{Sch.} & \langle \bot, \mathit{false}\rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathtt{v}(\mathtt{f},\mathtt{g}), \mathit{true}\rangle \\
\mapsto_{E.} & \langle \mathtt{k}(\mathtt{S},\mathtt{K})\wedge \mathtt{v}(\mathtt{K},\mathtt{Z}), \mathtt{S}=\mathtt{f}\wedge \mathtt{Z}=\mathtt{g}\rangle \\
\mapsto_{E.} & \langle \mathtt{k}(\mathtt{S},\mathtt{K})\wedge \mathtt{k}(\mathtt{K},\mathtt{K1})\wedge \mathtt{v}(\mathtt{K1},\mathtt{Z}), \mathtt{S}=\mathtt{f}\wedge \mathtt{Z}=\mathtt{g}\rangle
\end{array}
$$

Anmerkung

Betrachten wir nun die Frage, ob zwischen den Knoten `f` und `g` eine Verbindungskante existiert, d. h. das Ziel `verbindung(f,g)`. Mit der ersten Regel erhalten wir:

Diese Ableitung ist also erfolglos. Die Auswahl der zweiten Regel führt ebenfalls zu einer erfolglosen Ableitung, wenn im zweiten Ableitungsschritt das Ziel `kante(S,K)` zur Reduktion selektiert wird. Eine andere Möglichkeit ist es, das Ziel `verbindung(K,Z)` und die zweite Regel für `verbindung` auszuwählen:

Es ist leicht ersichtlich, daß sich bei Wiederholung der gleichen Auswahl eine unendliche Ableitung ergibt.

CLP Beispiel Minimum 1

$$\{\mathtt{min(X,Y,Z)} \leftarrow \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \doteq \mathtt{Z},$$
$$\mathtt{min(X,Y,Z)} \leftarrow \mathtt{Y} \leq \mathtt{X} \wedge \mathtt{Y} \doteq \mathtt{Z}\ \}$$

$$\begin{array}{ll} & \langle \mathtt{min}(\mathtt{1},\mathtt{2},\mathtt{C}), \mathit{true}\rangle \\ \mapsto_{Entfalten} & < \mathtt{X} \leq \mathtt{Y}\wedge\mathtt{X}\doteq\mathtt{Z},\ \ \mathtt{1}\doteq\mathtt{X}\wedge\mathtt{2}\doteq\mathtt{Y}\wedge\mathtt{C}\doteq\mathtt{Z}> \\ \mapsto_{Vereinfachen} & < \top,\ \mathtt{C}\doteq\mathtt{1}> \end{array}$$

Anmerkung

Das folgende CL-Programm implementiert das Prädikat `min/3`. `min(X,Y,Z)` trifft genau dann zu, wenn `Z` das Minimum von `X` und `Y` ist:

Hierbei erfolgt zuerst eine Reduktion *Entfalten*, bei der die erste Klausel ausgewählt wird. Dann erfolgt eine Constraintvereinfachung durch Anwendung der Reduktion *Vereinfachen*.

Die Ableitung ist erfolgreich und das Antwortconstraint ist $\mathtt{C} \doteq \mathtt{1}$. In den Beispielen führen wir beim Vereinfachen auch eine Projektion durch, um ein einfaches Antwortconstraint zu erhalten.

Betrachten wir nun eine andere Ableitung ausgehend vom obigen Anfangszustand. Die Auswahl der zweiten Klausel im ersten Ableitungsschritt führt zu einem inkonsistenten Constraintspeicher, nämlich $\mathtt{2} \leq \mathtt{1} \wedge \mathtt{2} \doteq \mathtt{C}$. Damit scheitert diese Ableitung.

CLP Beispiel Minimum 2

$$
\begin{array}{ll}
 & \langle \mathtt{min}(\mathtt{A}, \mathtt{2}, \mathtt{1}), \mathit{true} \rangle \\
\mapsto_{Entfalten} & <\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \doteq \mathtt{Z},\ \mathtt{A} \doteq \mathtt{X} \wedge \mathtt{2} \doteq \mathtt{Y} \wedge \mathtt{1} \doteq \mathtt{Z}> \\
\mapsto_{Vereinfachen} & < \top,\ \mathtt{A} \doteq \mathtt{1}>
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathtt{min}(\mathtt{A}, \mathtt{A}, \mathtt{B}), \mathit{true} \rangle \\
\mapsto_{Entfalten} & <\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \doteq \mathtt{Z},\ \mathtt{A} \doteq \mathtt{X} \wedge \mathtt{A} \doteq \mathtt{Y} \wedge \mathtt{B} \doteq \mathtt{Z}> \\
\mapsto_{Vereinfachen} & < \top,\ \mathtt{A} \doteq \mathtt{B}>
\end{array}
$$

Anmerkung

Das Ziel `min(A,2,2)` führt zur Antwort $\mathtt{A} \doteq \mathtt{2}$, wobei mit jeder der beiden Klauseln eine Ableitung mit dieser Antwort möglich ist. Das Ziel `min(A,2,3)` scheitert.

In Prolog würden diese Reduktionen zu einer Fehlermeldung führen, da noch kein Wert für `A` bekannt ist und der Vergleich nur zwischen bekannten Werten möglich ist. Im CLP-Kalkül ist es aber möglich, auch hier „rückwärts“ zu rechnen.

Die intensionale Antwort sagt uns, daß das Minimum von zwei gleichen Zahlen diese Zahl selbst ist. Die Auswahl der zweiten Klausel führt zum selben Endzustand.

Stabilitätseigenschaft für CLP-Sprachen

Wenn $\langle G, C\rangle \mapsto_{Entfalten} \langle G', C'\rangle$ und D ist ein Constraint und $C' \wedge D$ ist erfüllbar, dann ist $\langle G \wedge H, C \wedge D\rangle \mapsto_{Entfalten} \langle G' \wedge H, C' \wedge D\rangle$ möglich.

Anmerkung

Wenn eine Reduktion *Entfalten* zu einem Folgezustand mit Constraintspeicher C' führt, dann kann diese Reduktion auch auf jeden Zustand mit zusätzlichem Ziel und speziellerem Constraintspeicher als C' (solange er konsistent bleibt) angewendet werden, und der Folgezustand unterscheidet sich nur durch das zusätzliche Ziel und das zusätzliche Constraint.

Diese Eigenschaft ist einer der Vorteile von CLP gegenüber herkömmlichen Programmiersprachen mit überschreibender Zuweisung (engl. destructive assignment).

Vervollständigung

Beispiel 1 Sei G das Ziel `verbindung(f,g)`. Das Ziel hat keine erfolgreiche Ableitung. Es gilt allerdings:

$P^{\rightarrow} \not\models$ `verbindung(f,g)` und

$P^{\rightarrow} \not\models \neg$`verbindung(f,g)`

Wenn man $P^{\rightarrow}$ aber so „vervollständigt“, daß es nicht nur die aus den Klauseln resultierenden notwendigen Bedingungen (Implikationen), sondern auch die entsprechenden hinreichenden Bedingungen enthält (Implikation in die andere Richtung), dann folgt daraus $\neg$`verbindung(f,g)`. $\diamond$

Anmerkung

Eine CL-Klausel $A \leftarrow G$ kann als Implikation $G \rightarrow A$ verstanden werden. G ist die Konjunktion $A_1 \wedge \ldots \wedge A_n$. Die Implikation entspricht der Klausel $A \vee \neg A_1 \vee \ldots \vee \neg A_n$, was den Namen CL-Klausel erklärt. Die logische Leseweise eines Programms P ist dann der universelle Abschluß der Konjunktion der Klauseln von P, wir schreiben dafür $P^{\rightarrow}$.

Mit dieser einfachen deklarativen Semantik können allerdings nur positive Informationen erschlossen werden, nicht aber negative.

Vervollständigung

Definition 10 *Vervollständigung* von P, $P^{\leftrightarrow}$:
Für n-stellige Prädikat p aus P mit

$$\begin{array}{ccc} p(\bar{t}_1) & \leftarrow & G_1 \\ \vdots & & \vdots \\ p(\bar{t}_m) & \leftarrow & G_m \end{array}$$

wird in $P^{\leftrightarrow}$ eingefügt

$$\begin{array}{cccc} p(\bar{x}) & \leftrightarrow & \exists \bar{y}_1\ (\bar{t}_1 \doteq \bar{x} \wedge G_1) & \vee \\ & & \vdots & \vee \\ & & \exists \bar{y}_m\ (\bar{t}_m \doteq \bar{x} \wedge G_m) & \end{array}$$

$\bar{x}$ ist eine Folge der Länge n von paarweise verschiedenen, neuen Variablen, und $\bar{y}_i$ die Folge der Variablen in G_i und $\bar{t}_i$.
Für jedes andere Prädikatensymbol p wird in $P^{\leftrightarrow}$ eingefügt:

$$\neg p(\bar{x})$$

Clarksche Gleichheitstheorie CET

Definition 11 Sei Σ eine Signatur, die unendlich viele Funktionssymbole, davon mindestens eine Konstante, enthält. Die *Clarksche Gleichheitstheorie CET* ist:

Reflexivität:
$\forall(\top \rightarrow x \doteq x)$
Symmetrie:
$\forall(x \doteq y \rightarrow y \doteq x)$
Transitivität:
$\forall(x \doteq y \land y \doteq z \rightarrow x \doteq z)$
Verträglichkeit:
$\forall(x_1 \doteq y_1 \land \ldots \land x_n \doteq y_n \rightarrow f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n))$
Zerlegung:
$\forall(f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n) \rightarrow x_1 \doteq y_1 \land \ldots \land x_n \doteq y_n)$
Widerspruch:
$\forall(f(x_1, \ldots, x_n) \doteq g(y_1, \ldots, y_m) \rightarrow \bot)$ falls $f \neq g$ oder $n \neq m$
Azyklizität:
$\forall(x \doteq t \rightarrow \bot)$ falls t keine Variable ist und x in t vorkommt

Anmerkung

Damit das Gleichheitszeichen $\doteq$ richtig behandelt wird und nicht wie ein beliebiges binäres Prädikat, werden die Sätze, die die Gleichheit beschreiben (Clark'sche Gleichheitstheorie), zu $P^{\leftrightarrow}$ hinzugenommen.

Die ersten drei Sätze sagen aus, daß $\doteq$ eine Äquivalenzrelation ist.

Clark'sche Vervollständigung

Definition 12 Sei P ein Constraint-Logikprogramm und CT eine Constrainttheorie. Die *Clark'sche Vervollständigung* $\mathcal{P}$ von P ist die Formelmenge:

$$\mathcal{P} := CT \cup P^{\leftrightarrow}$$

Deklarative Semantik

Theorem 1 Korrektheit und Vollständigkeit von erfolgreichen Ableitungen [JaffarLassez87] Sei P ein CL-Programm und G ein Ziel.

Wenn G eine erfolgreiche Ableitung mit einem Antwortconstraint C hat, dann gilt

$$P^{\leftrightarrow}, CT \models \forall C \rightarrow G$$

Wenn $P^{\leftrightarrow}, CT \models \forall(C \rightarrow G)$ gilt und C in CT erfüllbar ist, dann hat G erfolgreiche Ableitungen mit Antwortconstraints $C_1, \ldots, C_n$, so daß

$$CT \models \forall C \rightarrow (C_1 \vee \ldots \vee C_n).$$

Anmerkung

Die deklarative und die operationale Semantik einer Sprache sollen einander möglichst gleichen. Einerseits soll alles, was wir ableiten können, auch logisch aus dem Programm folgen (Korrektheit), andererseits soll alles, was folgt, auch ableitbar sein (Vollständigkeit).

Der Vollständigkeitssatz für CLP zeigt, daß man im Gegensatz zum LP-Kalkül mehrere Antworten betrachten und kombinieren muß, um die Vollständigkeit zu erhalten. Falls aber das Constraintsystem unabhängig ist, kann die Disjunktion zu einem Disjunkt reduziert werden.

Beweis siehe Buch.

Deklarative Semantik

Beispiel 2 Sei P das folgende Programm und CT die Constrainttheorie, die u.a. die Constraints $\leq$ und $\geq$ interpretiert:

$\mathtt{p(X,Y)} \leftarrow \mathtt{X} \leq \mathtt{Y.}$
$\mathtt{p(X,Y)} \leftarrow \mathtt{X} \geq \mathtt{Y.}$

Das Ziel `p(X,Y)` hat zwei erfolgreiche Ableitungen mit den Antwortconstraints $\mathtt{X} \leq \mathtt{Y}$ bzw. $\mathtt{X} \geq \mathtt{Y}$.

Es gilt $P^{\leftrightarrow}, CT \models \forall(\mathit{true} \rightarrow \mathtt{p}(\mathtt{X},\mathtt{Y}))$; nach dem Satz also auch $CT \models \forall(\mathit{true} \rightarrow \mathtt{X} \leq \mathtt{Y} \vee \mathtt{X} \geq \mathtt{Y})$.

Jedes Antwortconstraint für sich allein reicht aber nicht aus: $CT \not\models \forall(\mathit{true} \rightarrow \mathtt{X} \leq \mathtt{Y})$ und $CT \not\models \forall(\mathit{true} \rightarrow \mathtt{X} \geq \mathtt{Y})$. $\diamond$

Deklarative Semantik

Definition 13 Eine Ableitung ist *fair*, wenn sie entweder scheitert oder wenn jedes Atom, das in der Ableitung vorkommt, nach endlich vielen Reduktionen ausgewählt wird.

Theorem 2 Vollständigkeit und Korrektheit von erfolglosen Ableitungen Sei P ein CL-Programm und G ein Ziel. Es gilt $P^{\leftrightarrow}, CT \models \neg\exists G$ genau dann, wenn jede faire Ableitung beginnend mit $\langle G, \mathit{true}\rangle$ endlich scheitert.

Anmerkung

Beweis siehe Buch.

Anfänge der Constraintbasierten Programmierung

1963	Sutherland, Sketchpad, Graphiksystem für geometrisches Zeichnen
1970	U. Montanari, Constraintnetzwerke
1970	R.E. Fikes, REF-ARF, Sprache für ganzzahlige, lineare Gleichungen
1972	A. Colmerauer, Marseille, sowie R.A. Kowalski, London, Prolog
1977	A.K. Mackworth, Constraints Netzwerk Algorithmen
1978	J.-L. Lauriere, Alice, Sprache für kombinatorische Probleme
1979	A. Borning, Thinglab, interaktives Graphiksystem
1980	G. L. Steele, Constraints, erste constraintbasierte Sprache, in LISP

1982	A. Colmerauer, Prolog II, Prolog mit Gleichheitsconstraints
1987	Ait-Kaci, Austin, Life, Prologerweiterung mit Gleichheitsconstraints
1987	J. Jaffar und J.L. Lassez, CLP(X) - Schema
1987	J. Jaffar, CLP(R), Monash Univ. Melbourne, arithmetische Constraints
1988	P. v. Hentenryck, CHIP, ECRC München, endlichen Wertebereichen
1988	P. Voda, Vancouver, Trilogy, CLP-ähnlich mit Ganzzahlarithmetik
1988	Older, Ottawa, Bell-Northern Research, Intervallarithmetik
1988	A. Aiba, Tokyo, ICOT, nichtlineare Gleichungssysteme
1988	W. Leler, Termersetzungs-Sprache zum Schreiben von Constraints
1988	A. Colmerauer, Prolog III, Univ. Marseille, Constraints über Listen

	Entwicklung von Prolog
um 1964	J. A. Robinson, Resolutionsprinzip
um 1972	A. Colmerauer, Marseille, spezialisierter Thereombeweiser für natürlichsprachiges Verstehen, genannt Prolog
um 1972	R. A. Kowalski, Edingburgh, prozedurale Interpretation von Hornklauseln
1975	D. H. D. Warren, London, erste effiziente Implementierung erster Compiler, fast vollständig in Prolog, WAM (Warren Abstract Machine)
1982-1994	Fifth Generation Computing Project, Japan

Anmerkung

Colermauer, Kowalski, FCGS Projekt

Künstliche Intelligenz, Expertenssysteme

Microsoft Windows NT Netzwerkkonfiguration

Knowledgewares Application Development Workbench (60.000 Lizenezen 1995) 250.000 Zeilen prolog, in C 10* soviel.

FCGS scheitert, CLP gewinnt, Prolog Konkurse

	Entwicklung von Prolog, Fortsetzung
1984-1996	LP am European Computer-Industry Research Center (ECRC), München, mit Schwerpunkt Logikprogrammierung ECL^{i}PS^e Prolog mit etwa 300 Lizenzen weltweit
seit 1985	LP am Swedish Insitute of Computer Science (SICS) Sicstus Prolog mit etwa 500 Lizenzen weltweit
80er Jahre	EG Esprit-Projekte im Bereich LP
80er Jahre	Borlands Turbo-Prolog, Pascalähnliche, Prologartige Sprache
1996	ISO Prolog Standard.