Constraint Logic Programming

Lucas Bockhorst 368335
Philipp Ludwig 367344

7. Juli 2017

1 Einleitung

Im Folgenden möchten wir einen Eindruck des `Constraint Logic Programming` (oder auch kurz `CLP`) als Teil der Logikprogrammierung vermitteln. Dazu wollen wir zunächst im Kapitel `Constraints` darauf eingehen und darlegen, was diese sind und die Art und Weise erklären, wie sie funktionieren; insbesondere möchten wir zeigen, wie Constraints eine Änderung der Beweisbäume bewirken. Des Weiteren möchten wir anhand eines Anwendungsbeispiels die Funktionsweise der Constraints genauer analysieren, um dieses Wissen später auf andere Probleme anwenden zu können. Zuletzt wollen wir alle Erkenntnisse und die Vor- und Nachteile der Constraint Logic Programming gegenüber der einfachen Logikprogrammierung im Fazit vorstellen und einen Ausblick darüber geben, was mit der Logikprogrammierung mit Constraints realisiert werden kann.

2 Constraints

Constraints sind, wie der Name schon sagt, Einschränkungen oder Bedingungen. Mit ihnen lässt sich die Logikprogrammierung erweitern, was zu neuen Lösungsmöglichkeiten für diverse Probleme führt.
Man kann vorgefertigte Module in Prolog per Direktive importieren. Eine Direktive ist eine Anfrage im Programmcode, die man wie eine Regel, nur ohne Kopf schreibt. Die Zeile beginnt also direkt mit `:-`. Sie wird beim Laden des Programms ausgeführt.
Mit dem Importieren der Bibliothek `clpfd` (Constraint Logic Programming over Finite Domains) in das Prolog-Programm per Direktive

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

erhalten wir folgende Constraint-Operatoren:

```
#>=, #=<, #=, #\=, #>, #<, ins.
```

Hierbei entsprechen die Symbole mit # davor grundsätzlich den bekannten arithmetischen Operatoren. Der Operator `ins` ist die Constraint Variante des Operators `in`, der angibt, ob etwas in einem Wertebereich liegt. So meint `X in 1..3` alle `X`, die im Intervall von 1 bis 3 liegen.
Außerdem darf das Logikprogramm mit Benutzung dieses Moduls auch Aussagen über die Theorie der ganzen Zahlen enthalten.

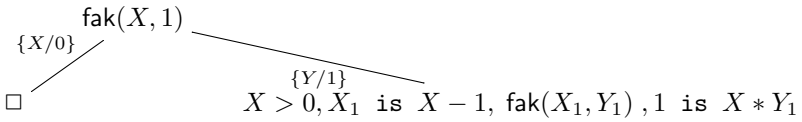
So erhält man zum Beispiel folgende Constraints:

```
X+Y #> Z*5
max(X,Y) #= X mod 3
```

Constraints sind also atomare Formeln bzw. Einschränkungen.
Mit dem Wechsel von der Logikprogrammierung zum `Constraint` Logic Programming verändern sich auch die Beweisbäume aufgrund der atomaren Bedingungen, den Constraints. Um diese Veränderungen zu veranschaulichen, nutzen wir ein Programm zur Berechnung der Fakultät:

```
fak(0,1).
fak(X,Y) :- X > 0, X1 is X - 1, fak(X1,Y1), Y is X*Y1.
```

Der klassische Beweisbaum zu der Anfrage `?- fak(X,1)` sieht wie folgt aus:

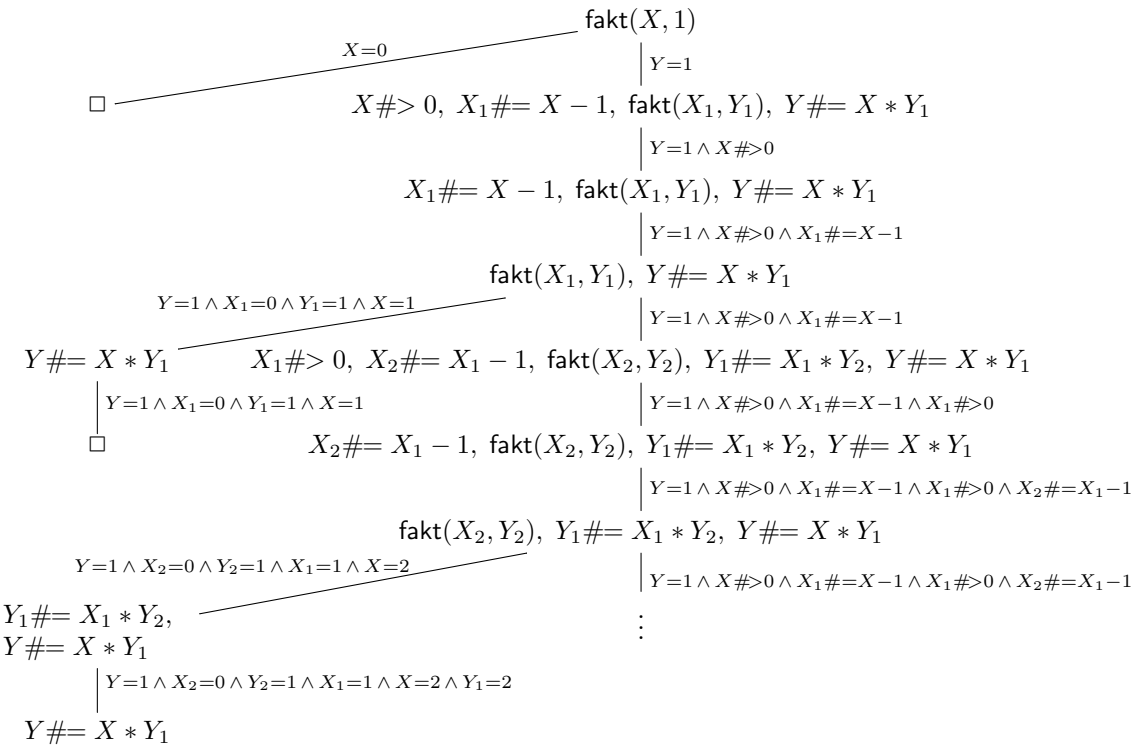


Die erste Antwort, die gefunden wird, ist also $X = 0$. Wenn man nun `;` eingibt, führt dies jedoch zu einem Programmabbruch, da wir das Beweisziel $X > 0$ in Prolog nicht stellen dürfen. Beide Argumente von $>$ müssten voll instanziiert sein, wohingegen die Variable hier nicht instanziiert ist.

Das Programm `fak` ist also nicht bidirektional, d.h. man kann es sehr gut dafür benutzen, zu einer Zahl die Fakultät zu berechnen, jedoch nicht zu einer gegebenen Fakultät, bspw. 1, ein X zu finden, das genau die Fakultät 1 hat.
Hier zeigt sich nun einer der Vorteile von Constraint Logic Programming. Im Gegensatz zu $>$ ist das Symbol $\#>$ tatsächlich bidirektional. Wir programmieren nun noch einmal unser Fakultät-Programm, allerdings diesmal mit Constraints:

```
fakt(0,1).
fakt(X,Y) :- X #> 0, X1 #= X-1, fakt(X1,Y1), Y #= X*Y1.
```

Wir ersetzen also unsere bekannten arithmetischen Symbole durch ihre entsprechenden Gegenstücke für Constraints. Obwohl das `fakt`-Programm sich vom Aussehen her kaum vom `fak`-Programm unterscheidet, verändert sich der Beweisbaum zu der eben gestellten Anfrage `?- fakt(X,1)` nun signifikant:



Was zunächst auffällt: Der Beweisbaum ist deutlich größer als vorher (aktuell ist der rechte Pfad unendlich). Zum anderen verändert sich aber auch das

Verfahren der Beweisführung im Bezug auf die Ersetzungen, die wir vornehmen.
Im Gegensatz zu den Ersetzungen von Variablen, die wir in den normalen Beweisbäumen vornehmen, wie in unserem Baum zum `fak`-Programm, schreiben wir bei Beweisbäumen die Konjunktionen von Constraints auf und fügen jeweils weitere Constraints zu unserer Konjunktion hinzu, bis wir daraus entweder eine Antwortsubstitution herleiten können oder die Konjunktion durch einen hinzugefügten Constraint nicht mehr erfüllbar ist. Das muss durch einen im CLP-System vorhandenen Constraint Solver überprüft werden.
Beim Bilden der Konjunktionen werden zudem nicht nur Constraints der Constraint-Theorie aufgenommen, sondern auch Constraints, die aus der normalen Unifikation von Anfragen und Programmklauseln stammen.
Im linken Teilbaum fügen wir also im ersten Schritt $X = 0$ zu unserer Konjunktion hinzu und sind fertig, da die erste Programmklausel angewendet werden kann.
Im rechten Teilbaum betrachten wir die zweite Programmklausel und fügen $Y = 1$ zu unserer Konjunktion hinzu. Jetzt betrachten wir den Rumpf der Regel von links nach rechts. Wir finden nun $X\# > 0$. Dieser Constraint wird somit in unsere Konjunktion mit aufgenommen.
Wir schreiben also $Y = 1 \wedge X\# < 0$. Da in unserer Konjunktion noch kein Widerspruch zu finden ist, gehen wir zum nächsten Constraint und führen dies fort.
Das CLP-System vereinfacht die Konjunktionen der Constraints zudem. Anstelle von $Y = 1 \wedge X_1 = 0 \wedge Y_1 = 1 \wedge X = 1$ müsste eigentlich $Y = 1 \wedge X\# > 0 \wedge X1\# = X - 1 \wedge X1 = 0 \wedge Y1 = 1$ stehen. Das wird zu $Y = 1 \wedge X_1 = 0 \wedge Y_1 = 1 \wedge X = 1$ vereinfacht, sodass man nachher die 2. Antwortsubstitution $X = 1$ bekommt.
Wie schon erwähnt, ist unser rechter Pfad unendlich. Das liegt daran, dass $Y\# = X * Y1$ nie in unsere Konjunktion aufgenommen wird, da das `fakt(...)`-Constraint links davon steht. Wenn wir $Y\# = X * Y1$ mit aufnehmen würden, könnten wir den Pfad sofort erfolglos abschließen, da aus

---

$Y = 1 \wedge X\# > 0 \wedge X1\# = X - 1 \wedge X1\# > 0$

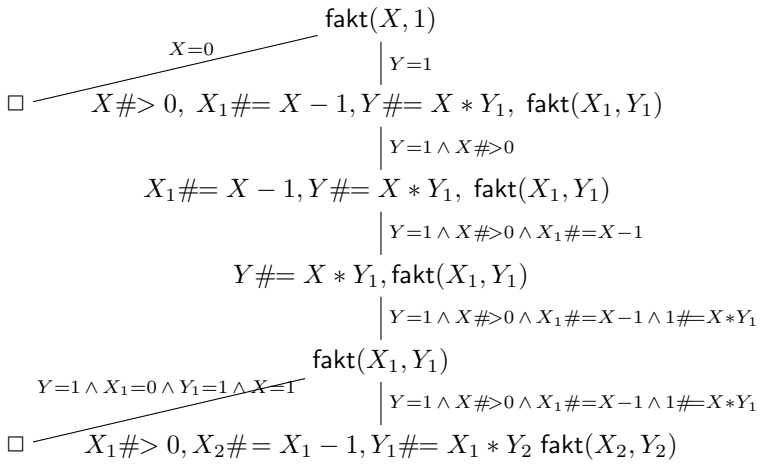
---

folgt, dass $Y = 1$ und $X\# > 1$ sein müssen, was allerdings dem Constraint $Y\# = X * Y1$ widerspricht. Wir erhalten also unsere Antworten $X = 0$ und $X = 1$, bei einer erneuten Eingabe von `;` terminiert unser Programm jedoch nicht.
Damit unser Programm terminiert, können wir im `fakt`-Programm die letz-

ten beiden Formeln unserer zweiten Regel vertauschen. Das Programm sieht nun folgendermaßen aus:

```
fakt(0,1).
fakt(X,Y) :- X #> 0, X1 #= X-1, Y #= X*Y1, fakt(X1,Y1).
```

Unsere Anfrage `?- fakt(X,1)` ergibt nun folgenden Beweisbaum:



Der rechte Pfad ist nun zu Ende, da der Constraint

$$Y = 1 \wedge X\# > 0 \wedge X1\# = X - 1 \wedge 1\# = X * Y1 \wedge X1\# > 0$$

der sich ergibt, wenn man weitermachen würde, widersprüchlich ist.
Wir sehen, unser Baum terminiert nun mit den Antworten $X = 0$, $X = 1$ und `false`, falls ein weiteres mal `;` eingegeben wird.

3 Anwendung

Im Folgenden möchten wir das Constraint Logic Programming anhand einer Anwendung, dem Lösen eines Sudokus, etwas näher erläutern.

Bei einem Sudoku hat man grundsätzlich 9x9 Felder, die wiederum in jeweils 3x3 Felder unterteilt sind. Dabei gilt, dass jede Ziffer von Eins bis Neun nur genau einmal in jeder Spalte, Zeile und jedem Kästchen von 3x3 Feldern vorkommen darf. Einige dieser Zahlen sind vorgegeben, was das Lösen eines solchen Sudokus gerade schwer macht. Das gesamte Sudoku werden wir als Liste $[x_1, \ldots, x_n]$ repräsentieren, wobei die Zahl x_i für die i-te Zeile steht,

welche ebenfalls eine Liste $[s_1, \ldots, s_n]$ mit den Ziffern enthält. Im Code wird später x_i durch einen repräsentativen Buchstaben für die bessere Übersicht ersetzt. Um nun ein Sudoku Problem lösen zu können, muss dieses zuvor definiert werden. Dazu definiert man ein Faktum mit dem gewünschten Sudoku Problem:

```
problem([[_,_,_,_,_,_,_,_,_],
        [_,_,_,_,_,3,_,8,5],
        [_,_,1,_,2,_,_,_,_],
        [_,_,_,5,_,7,_,_,_],
        [_,_,4,_,_,_,1,_,_],
        [_,9,_,_,_,_,_,_,_],
        [5,_,_,_,_,_,_,7,3],
        [_,_,2,_,1,_,_,_,_],
        [_,_,_,_,4,_,_,_,9]]).
```

Nun kann das Rätsel mit

```
?- problem(Rows), sudoku(Rows), maplist(portray_clause, Rows).
```

gelöst werden. Das Prädikat `maplist` wendet sein erstes Argument – in diesem Fall das Prädikat `portray_clause` – auf jedes Element der Liste Rows an. Hierbei ist `portray_clause` ein Pretty Printer für Klauseln und Listen, welches als Seiteneffekt diese auf dem Bildschirm ausgibt:

```
? - portray_clause([1,2]).
[1,2]
```

In der Klausel für das Prädikat `sudoku` wird `append` aufgerufen, welches eine Liste von Listen in eine einzige Liste überführt. Insbesondere wird hier die Liste von Listen Rows zu einer einzigen Liste `Vs` verkettet. Mit Hilfe des Prädikats `ins` aus der Bibliothek *clpfd*, welches ähnlich ist wie `in`, wird sicher gestellt, dass alle Elemente der Liste `Vs` aus dem vorgegebenen Bereich stammen. Es gilt also $[x_1, \ldots, x_{81}]$ `ins` $1 \ldots 9$ falls x_i in $1 \ldots 9$ für alle $1 \leq i \leq 81$ gilt. Also werden alle 81 Einträge des Sudokus überprüft, ob sie in dem Intervall $[1, 9]$ liegen. Das vordefinierte Prädikat `all_distinct` aus der Bibliothek `clpfd`, welches mit `maplist` aufgerufen wird, stellt sicher, dass alle Elemente in jeder Zeile verschieden sind. Hierbei stehen also die

`ins`– und `maplist(all_distinct, ...)`–Formeln wieder für Abkürzungen für (naheliegende) Konjunktionen von Constraints. Nun werden die Spalten mit den Reihen transponiert (`transpose`). Da wir nun auch die Spalten auf verschiedene Ziffern überprüfen müssen, wird unter Zuhilfenahme des Prädikats `maplist` und des Prädikats `all_distinct` jede Spalte auf verschiedene Ziffern überprüft.

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

sudoku(Rows) :-
        append(Rows, Vs),
        Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).

problem( [[_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,3,_,8,5],
          [_,_,1,_,2,_,_,_,_],
          [_,_,_,5,_,7,_,_,_],
          [_,_,4,_,_,_,1,_,_],
          [_,9,_,_,_,_,_,_,_],
          [5,_,_,_,_,_,_,7,3],
          [_,_,2,_,1,_,_,_,_],
          [_,_,_,_,4,_,_,_,9]]).
```

Um aber die ganzen Constraints nutzen zu können, müssen wir zuerst die `clpfd` Bibliothek konsultieren. Damit man das Programm später direkt nutzen kann, ohne immer die Bibliothek konsultieren zu müssen, schreiben wir den Import direkt als Befehl `:-use_module(library(clpfd)).` mit in die Datei. Dann folgt die Einteilung der Zeilen nach vordefinierter Liste durch Buchstaben von `As` bis `Is`. Nun können die einzelnen Zeilen in drei Blöcke

aufgeteilt werden, die rekursiv wieder in drei Blöcke aufgeteilt werden, sodass nachher insgesamt neun 3x3 Felder gespeichert sind – in jedem Block ein 3x3 Feld. Das Prädikat `blocks` sorgt dafür, dass – wie zuvor genannt – alle Reihen und Spalten rekursiv in 3x3 Blöcke aufgeteilt werden. Mit Hilfe des Prädikates `all_distinct` wird dann überprüft, dass alle Ziffern innerhalb eines Blockes verschieden sind. Danach wird das Prädikat `blocks` erneut aufgerufen, um auch den nächsten Block mit 3x3 Feldern auf Unterschiedlichkeit zu überprüfen.
Schlussendlich erhalten wir obiges Programm. Ruft man nun also

```
? - problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
```

auf, so erhält man

```
[9, 8, 7, 6, 5, 4, 3, 2, 1].
[2, 4, 6, 1, 7, 3, 9, 8, 5].
[3, 5, 1, 9, 2, 8, 7, 4, 6].
[1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7].
[7, 9, 5, 4, 6, 1, 8, 3, 2].
[5, 1, 9, 2, 8, 6, 4, 7, 3].
[4, 7, 2, 3, 1, 9, 5, 6, 8].
[8, 6, 3, 7, 4, 5, 2, 1, 9].
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

innerhalb kürzester Zeit als Ergebnis.
Das letzte Prädikat `maplist` mit dem Prädikat `portray_clause` lässt die Einträge der Liste Rows mittels Pretty Print darstellen (hier also durch eine Matrix).
Natürlich gibt es viele weitere Anwendungen, die mittels Constraint Logic Programming gelöst werden können. Ein häufiges Beispiel ist die Berechnung eines Raumbelegungsplans, Pathfinder oder auch das N-Dame Problem und viele weitere interessante Anwendungen .

4 Fazit

Zusammenfassend haben wir also dargelegt, wie das Constraint Logic Programming funktioniert. Wir haben gezeigt, dass Constraints Einschränkungen auf atomarer Ebene sind und wie diese die Abfrage beeinflussen. Ins-

besondere haben wir auch gesehen, wie die Constraints die Beweisbäume abändern und so viele neue Möglichkeiten eröffnen. Weiterhin können wir Constraints nun anwenden und aufgrund des Anwendungsbeispiels konnten wir Schritt für Schritt nachvollziehen, wie genau man ein Programm mit der Constraint Logic Programming aufzubauen hat. So können wir nun auch andere Probleme, wie das N-Dame Problem, Pathfinder oder Raumbelegungspläne in Zukunft lösen. Schussendlich können wir schnell sehen, dass das Constraint Logic Programming als Erweiterung der Logikprogrammierung wesentlich mehr Vorteile mit sich bringt als Nachteile. Zum einen erhalten wir viele nun bidirektionale atomare Bedingungen (wie am Beispiel der Fakultät gezeigt), zum anderen ist das Constraint Logic Programming durch den abgeänderten Beweisbaum sehr viel effizienter als die reine Logikprogrammierung, was sie deswegen auch gerade so für den Anwendungsbereich sehr interessant macht. Zum Beispiel werden Raumbelegungspläne oder auch kürzeste Wege wie bei Navigationssystemen mittels CLP realisiert.
Ein Nachteil allerdings ist, dass man sich erst einmal mit Constraints wirklich beschäftigen muss, bevor man diese Erweiterung der Logikprogrammierung richtig und auch vernünftig anwenden kann.

Als kleinen Ausblick wollen wir hier noch ein paar weitere Probleme nennen, die mit der Constraint Logic Programming sehr gut lösbar sind:
Five Houses Puzzle, Mastermind, Warehouse Location Problem, a Crossword Puzzle, Constraint Reasoning, Crypto-Arithmetic Problem (Send + More = Money) und noch viele andere.

Literatur

[1] Jürgen Giesl. *Logikprogrammierung*. RWTH Aachen, 2015
`http://verify.rwth-aachen.de/lp15/skript.pdf`

[2] Thom Frühwirth. *Constraint Programming*. Universität Ulm, 2007
`http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/pisa/slides-studium-generale.pdf`

[3] SWI Prolog *Example: Sudoku*
`http://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku`

[4] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*
The MIT Press Cambridge, Massachusetts
London, England
`http://tocs.ulb.tu-darmstadt.de/34099174.pdf`