**Proseminar**

# Logikprogrammierung mit Constraints

Eric Weber

RWTH Aachen
Aachen, Deutschland
Lehr- und Forschungsgebiet Informatik 2
Prof. Dr. Jürgen Giesl

*Betreuer*
*Jera Hensel*

## 1 Einleitung

Man stelle sich vor, man habe die letzte Ziffer seiner Banknummer vergessen. Jedoch weiß man noch, dass die Zahl definitiv größer als 5 war und außerdem war es eine Primzahl. Aus diesen beiden Informationen über die vergessene Zahl kann man eindeutig schließen, dass die gesuchte Zahl die 7 sein muss, da keine andere Ziffer beide Bedingungen erfüllt. Diese Bedingungen sind nichts anderes als *Constraints*. Ein so einfaches Problem können wir noch selber lösen, jedoch gelangen wir bei steigender Komplexität wie bei vielen Sachen auch hier schnell an unsere Grenzen. Aus diesem Grund gibt es die *Logikprogrammierung mit Constraints* (CLP). Mitte der 1980er Jahre, aus der Kombination von *Constraint Solving* und *Logikprogrammierung* entstanden, kommt sie vor allem dort zum Einsatz, wo Ressourcen effizient organisiert werden müssen. Oft wird die CLP als das bezeichnet, was dem heiligen Gral der Programmierung am nächsten kommt. Nämlich, dass man einfach nur das Problem deklariert und der Computer es von alleine löst. Nun wollen wir uns die CLP genauer ansehen.

## 2 Logikprogrammierung mit Constraints

### 2.1 Constraints

Zuerst einmal sollte der Begriff *Constraint* noch einmal näher erläutert werden.
*Constraints* sind Bedingungen oder Eigenschaften von sonst unbekannten Objekten, beziehungsweise Variablen, oder aber auch Beziehungen zwischen den Variablen.
*Beispiel*:
Seien $X$ und $Y$ zwei Variablen, deren Wert wir herausfinden wollen. Gegeben haben wir zuerst zwei *Constraints*, welche nur Eigenschaften der Variablen an sich sind, nämlich $X$ soll kleiner als 4 sein $(X < 4)$ und $Y$ soll größer als 0 sein $(Y > 0)$. Jedoch können *Constraints*, wie bereits erwähnt, auch Beziehungen zwischen den Variablen sein. Beispielweise soll $X$ drei mal so groß sein wie $Y (Y * 3 = X)$. Mit diesen Informationen kann man nun folgern, dass $X = 3$ sein muss und $Y = 1$ sein muss.

### 2.2 Syntax

Die Syntax eines Logikprogramms mit *Constraints* unterscheidet sich von einem normalen Logikprogramm nur in der Hinsicht, dass nun *Constraints* hinzugefügt werden. Die *Prädikatssymbole* werden nun also um *Constraint-Symbole* erweitert. Diese werden durch eine sogenannte *Constraint Theorie* definiert und von einem *Constraint Solver* ausgewertet.

### 2.3 Constraint-Theorie und Constraint-Solver

Die *Constraint-Theorie* ist eine Menge von geschlossenen Formeln und wenn ein *Constraint* aus dieser Formelmenge folgt, dann ist er wahr. Ob man aus der *Constraint-Theorie* einen *Constraint* folgern kann, wird von dem bereits erwähnten *Constraint-Solver* überprüft, jedoch ist nicht jede *Constraint-Theorie* und dem entsprechend auch nicht jeder *Constraint-Solver* gleich. In dieser Ausarbeitung arbeiten wir nur mit einer *Constraint-Theorie* über den ganzen Zahlen, also Integers und der „intuitiven“ Deutung der *Funktions- und Prädikatssymbole*. Das bedeutet, dass z.B. das Symbol $<$ (kleiner als) als *Constraint* die Bedeutung haben soll, wie sie in der Mathematik definiert ist, also, dass die linke Zahl kleiner als die rechte Zahl sein soll.

### 2.4 Definition

Um ein *Constraint-Programm* zu definieren, wird zuerst eine Menge von Variablen $X_1, \ldots, X_n$gegeben, dessen Auswertung unser Ziel ist.
Anschließend wird für jede Variable $X$ die Menge der Werte gegeben, welche die Variable annehmen kann, und zwar mit $X \in \{v_1, \ldots, v_n\}, n \in \mathbb{N}$.
Die Menge der Werte $v$, welche eine Variable $X$ annehmen kann, nennen wir *Definitionsbereich* von $X$, jedoch ist zu beachten, dass auch andere Eigenschaften einer Variable in dem Definitionsbereich stehen können, dies werden wir im späteren Beispiel auch sehen.
Als letztes müssen die *Constraints* definiert werden, welche für dieses Programm gelten.

### 2.5 Differenzierung

Wir wollen nun erläutern, was ein Logikprogramm mit *Constraints* von einem Logikprogramm ohne *Constraints* unterscheidet.
In einem Logikprogramm ohne *Constraints* können Variablen entweder einen festen Wert oder keinen Wert haben, während bei Logikprogrammen mit *Constraints* die Variablen zum einen, keinen festen Wert annehmen müssen, sondern einen Definitionsbereich von möglichen Werten haben und zum anderen, deren Auswertung solange verzögert werden kann, bis deren Wert sicher bestimmt werden kann.
Deswegen ist auch die Reihenfolge der *Constraints* nicht wichtig, was die Generalität des Programms erhöht, des Weiteren ist man nicht nur auf die Arbeit mit festen Werten für eine Variable beschränkt, sondern kann mit ganzen Wertebereichen arbeiten und auch mit anderen Eigenschaften der Variablen. Im späteren Beispiel (2.7) werden wir z. B. neben möglichen Werten für eine Variable auch noch mit einer Präferenz für jeden Wert arbeiten. Dies macht ein Logikprogramm mit *Constraints* deutlich flexibler.
Wenn wir also Bedingungen wie $(X < 4)$ nicht als *Constraints* definieren würden, könnte Prolog diese nicht

auswerten, beziehungsweise würde es einen Initialisierungsfehler geben, da in einem Prolog-Programm ohne *Constraints* jede Variable nur mit einem festen Wert initialisiert werden kann.
Zuletzt ist auch die Auswertung signifikant effizienter als bei einem Logikprogramm ohne *Constraints*, was wir im folgenden Abschnitt sehen werden.

Um nun *Constraints* von normalen Symbolen zu unterscheiden, wird vor ein *Constraint-Symbol* ein Indikator-Symbol geschrieben, sodass klar wird, dass es sich um einen *Constraint* handelt.
Wir werden hier das Symbol $'\#'$ verwenden, sodass z.B. $'\# <'$ den *Constraint* darstellt, welcher die kleiner-als-Relation beschreibt und $'\# \geq'$ den *Constraint* für die größer-gleich-Relation darstellt.

### 2.6 Auswertung

Um bei der CLP zu einer Lösung zu gelangen, müssen die Variablen Werte annehmen, sodass alle *Constraints* erfüllt werden. Um dies zu erreichen, könnte man die Variablen einfach solange alle möglichen Werte annehmen lassen, bis alle *Constraints* erfüllt sind. Dieses Verfahren nennt man auch die *generate-and-test* Methode, jedoch ist diese Herangehensweise meistens unpraktikabel, aber in jedem Fall sehr ineffizient.
Eine deutlich besser Auswertungsstrategie ist die so genannte *constraint-and-test* Methode. Anstatt die gegebenen *Constraints* nur passiv zur Überprüfung zu nutzen, werden bei diesem Ansatz die *Constraints* aktiv genutzt, um die Suche nach passenden Werten deutlich einzuschränken. Es wird also zuerst überprüft, welche Werte mit den *Constraints* überhaupt vereinbar sind. Ein kleines Beispiel soll dies nochmal verdeutlichen.

*Beispiel*:
Gegeben sei eine Menge $\mathcal{X}$ von Variablen mit $\mathcal{X} = \{X, Y, Z\}$,
außerdem ein Definitionsbereich $D$ mit $D = \{1, 2\}$ für alle $X_i \in \mathcal{X}$,
und schließlich noch die *Constraints*: $X\# = Y \wedge X\# \neq Z \wedge Y\# > Z$.
Nun gehen wir mit der *generate-and-test* Methode einfach alle möglichen Werte durch: (siehe Abb. 1)

Wie man sieht, finden wir die Lösung erst, nachdem wir schon fast alle möglichen Werte ausprobiert haben.
Nun die *constraint-and-test* Methode:
Aus $(Y > Z)$ wissen wir, dass $Y$ nur den Wert 2 annehmen kann und $Z$ nur den Wert 1.
Wir löschen die unpassenden Werte aus den Definitionsbereichen, woraus sich folgendes ergibt:
$D(X) = \{1, 2\}, D(Y) = \{2\}, D(Z) = \{1\}$.
Aus $(X = Y)$ wissen wir, dass auch $X$ den Wert 2 haben muss. Wir passen die Definitionsbereiche an:
$D(X) = \{2\}, D(Y) = \{2\}, D(Z) = \{1\}$.
Nun kennen wir schon alle Werte, da jeder Definitionsbereich nur noch einen Wert hat und auch der letzte *Constraint* $(X \neq Z)$ erfüllt ist. Mit dieser Methode haben wir also die Lösung erhalten ohne irgendwelche Werte auszuprobieren. Natürlich ist es nicht immer möglich den Suchbereich auf nur einen Wert zu reduzieren, meistens können wir diesen aber deutlich verkleinern. Dies macht Logikprogramme mit *Constraints* besonders effizient. Wenn wir den Suchbereich nicht auf nur einen Wert für jede Variable reduzieren können, müssen wir anschließend zwar mit der *generate-and-test* Methode die Lösung errechnen, jedoch ist die Laufzeit dafür signifikant kürzer da wir vorher den Suchbereich minimiert haben.

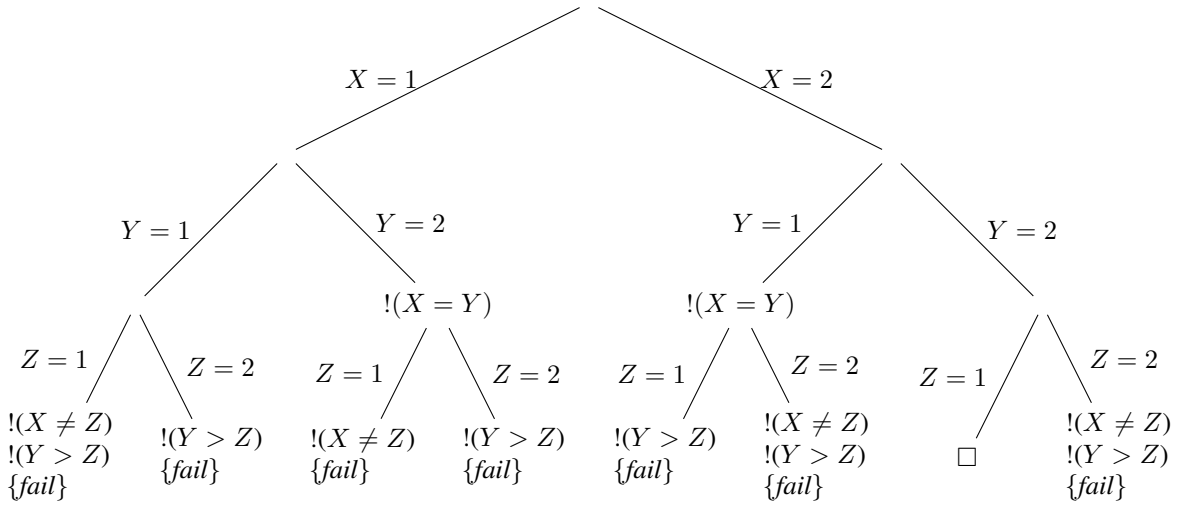


Abbildung 1:

### 2.7 Beispiel: Universitätsstundenplan

Nun wollen wir uns ein Beispiel etwas genauer ansehen. Es handelt sich um die Erstellung eines Universitätsstundenplans. Da hier Ressourcen wie Räume und Zeit möglichst effizient organisiert werden müssen, eignet sich zur Lösung dieses Problems die Logikprogrammierung mit *Constraints*.
Damit ein solcher Stundenplan geeignet ist, müssen gewisse Bedingungen, also *Constraints*, erfüllt sein, jedoch müssen wir hier eine Unterscheidung machen. Da es in der Regel keinen Stundenplan gibt, welcher wirklich alle *Constraints* konsequent erfüllt, müssen wir hier zwischen *harten Constraints* und *weichen Constraints* unterscheiden. Ein *harter Constraint* ist eine Bedingung, die immer erfüllt sein muss. Beispielsweise sollten zwei Kurse nicht zur selben Zeit im selben Raum Unterricht haben. Ein *weicher Constraint* hingegen muss nicht erfüllt sein, jedoch sollten natürlich auch *weiche Constraints* soweit es möglich ist beachtet werden. Ein Beispiel hierfür wäre die Entfernung zwischen den Räumen zweier aufeinander folgenden Veranstaltungen. Es wäre natürlich von Vorteil, wenn dieser Weg so kurz wie möglich ist, jedoch ist es eine Bedingung, dessen Verletzung man durchaus tolerieren kann.

Nun wollen wir unser Problem definieren. Zur besseren Verständlichkeit werden wir hier nur eine stark vereinfachte Version dieses Problems darlegen.
Zuerst geben wir die Menge der Variablen an, welche hier für die Veranstaltungen stehen. Wir definieren uns eine Menge von fünf Veranstaltungen mit
$X = \{ProSem, LA, FoSAP, DsAl, BuS\}$.

Des Weiteren definieren wir die Definitionsbereiche hier als Tupel von Werten, welche die Variablen annehmen können, und deren Präferenz. Wir werden im folgenden nur die Präferenzen 0 und 1 verwenden, wobei 1 eine höhere Präferenz darstellen soll als 0. Am Anfang sind noch keine Präferenzen gegeben, also initialisieren wir diese überall mit 0. Die Werte sind am Anfang auch noch für jede Veranstaltung gleich, sodass sich folgender Definitionsbereich ergibt:
$D = \{(8,0),(10,0),(12,0),(14,0),(16,0)\}$ für alle $X_i \in \mathcal{X}$.
Wir wollen also fünf Veranstaltungen auf einen Tag verteilen wobei die Werte für die Uhrzeit stehen zu

welcher die Veranstaltung beginnt.

Nun legen wir die *Constraints* fest. Zuerst definieren wir einen *harten Constraint* mit
$X_i \# \neq X_j \; \forall \; X_i, X_j \in \mathcal{X}$.
Dieser *Constraint* sagt aus, dass zwei Veranstaltungen nicht zur selben Zeit stattfinden dürfen, also dürfen die Werte zweier Variablen nicht gleich sein.
Nun ist es noch so, dass die Professoren nicht den ganzen Tag Zeit haben und so gibt jeder Professor an, zu welcher Zeit er die Veranstaltung halten kann. Da es für einen Professor unmöglich ist eine Veranstaltung abzuhalten wenn er keine Zeit hat, handelt es sich hier wieder um *harte Constraints*. Sie lauten wie folgt:
$ProSem \# > \; 8 \; \wedge \; LA \# < \; 16 \; \wedge \; DsAl \# > \; 14 \; \wedge \; FoSAP \# < \; 14 \; \wedge \; BuS \# > \; 10$
Mit diesen *Constraints* können wir schonmal den Suchbereich reduzieren, indem wir die Werte aus den Definitionsbereichen löschen, welche nicht mit den gegebenen *Constraints* übereinstimmen. Daraus ergeben sich nun folgende Definitionsbereiche:
$D(ProSem) = \{(10,0),(12,0),(14,0),(16,0)\}, D(LA) = \{(8,0),(10,0),(12,0),(14,0)\}$,
$D(DsAl) = \{(16,0)\}, D(FoSAP) = \{(8,0),(10,0),(12,0)\}, D(BuS) = \{(12,0),(14,0),(16,0)\}$.
Nun haben manche Professoren aber noch Präferenzen, zu welchen Uhrzeiten sie unterrichten wollen. Bei diesen Präferenzen handelt es sich jedoch nur um *weiche Constraints*, da deren Einhaltung nicht notwendig ist. Das bedeutet nun, dass wir Werte, welche die folgenden *Constraints* nicht erfüllen, nicht löschen, sondern nur die Präferenz aller anderen Werte in dem Definitionsbereich um 1 erhöhen. Die Professoren haben folgende Präferenzen angegeben:
$ProSem \# > \; 12 \; \wedge \; LA \# > \; 10 \; \wedge \; FoSAP \# < \; 12 \; \wedge \; BuS \# > \; 12$
Daraus ergeben sich nun folgende Definitionsbereiche:
$D(ProSem) = \{(10,0),(12,0),(14,1),(16,1)\}, D(LA) = \{(8,0),(10,0),(12,1),(14,1)\}, D(DsAl) = \{(16,0)\}, D(FoSAP) = \{(8,1),(10,1),(12,0)\}, D(BuS) = \{(12,0),(14,1),(16,1)\}$.
Wenn wir uns den Definitionsbereich $D(DsAl)$ anschauen, sehen wir dort nur noch einen Wert stehen. Dies bedeutet, dass der Wert von $DsAl$ schon feststeht $(16)$, da die Variable offensichtlich keinen anderen Wert mehr annehmen kann. Aufgrund unseres ersten *Constraints*, welcher aussagt, dass zwei Werte einer Veranstaltung nicht gleich seien können, können wir die 16 aus allen anderen Definitionsbereichen löschen und wir erhalten folgende Definitionsbereiche:
$D(ProSem) = \{(10,0),(12,0),(14,1)\}, D(LA) = \{(8,0),(10,0),(12,1),(14,1)\}, D(DsAl) = \{(16,0)\}, D(FoSAP) = \{(8,1),(10,1),(12,0)\}, D(BuS) = \{(12,0),(14,1)\}$.

Nun haben wir mit der *constraint-and-test* Methode den Suchbereich so weit wie möglich verkleinert, also müssen wir nun mit der *generate-and-test* Methode weitermachen. Zur Vereinfachung lassen wir alle Pfade, welche zu einem *fail* führen, direkt weg. Sei außerdem $P$ die Anzahl wie oft eine Präferenz auf einem Pfad erfüllt wurde. Es ergibt sich der Baum in Abbildung 2.

Hier haben sich sieben Lösungen ergeben, welche alle *harten Constraints* erfüllen. Aus diesen wählen wir nun die Lösung aus, welche auch die meistens *weichen Constraints* erfüllt, wo also $P$ den höchsten Wert hat und es ergibt sich eine eindeutige Lösung. Unser Stundenplan lautet also wie folgt:

| 08 Uhr | 10 Uhr | 12 Uhr | 14 Uhr | 16 Uhr |
|---|---|---|---|---|
| FoSAP | Proseminar | LA | BuS | DsAl |

Zwar konnte die Präferenz vom Proseminar nicht berücksichtigt werden, nichtsdestotrotz ist dies für uns die optimale Lösung.

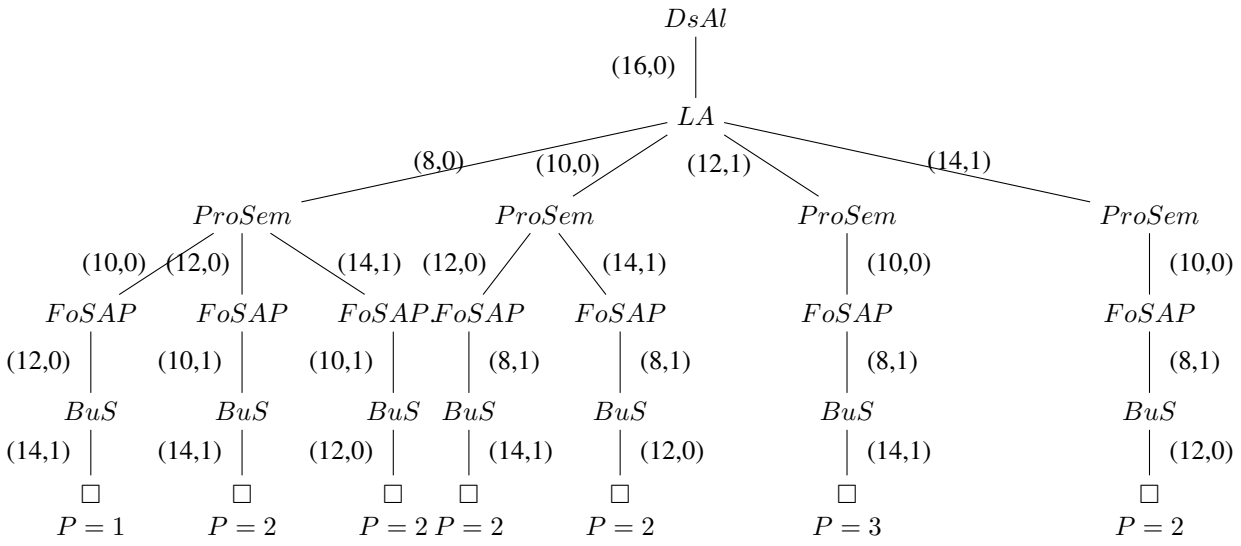


Abbildung 2:

## 3 Zusammenfassung

Wie wir gesehen haben, ist nicht viel Aufwand nötig um das Universitätsstundenplan-Problem zu lösen. Wir müssen uns nur die Variablen, deren Definitionsbereiche, die *Constraints* und wie die *Constraints* verarbeitet werden definieren. Das ist auch der Grund, wieso CLP oft als das bezeichnet wird, was dem heiligen Gral der Programmierung am nächsten kommt. Die Vorteile sind offensichtlich, wir müssen uns keine Gedanken machen wie wir ein Problem angehen, denn das entscheidet der Computer für uns. Die Erweiterung der Logikprogrammierung um *Constraints* macht die Logikprogrammierung außerdem deutlich flexibler, da Variablen keinen festen Wert annehmen müssen, ausdrucksstärker, da wir mit verschiedenen Eigenschaften von Variablen gleichzeitig arbeiten können und effizienter, da wir mit der *constraint-and-test* Methode den Suchbereich deutlich reduzieren können. Beispielweise ist der *Code*, welcher den Stundenplan für die Universität in München errechnet, gerade einmal 20 Zeilen lang und liefert ein Ergebnis in wenigen Minuten, während man damals per Hand noch mehrere Tage bis sogar Wochen für die Erarbeitung eines Stundenplans gebraucht hat.
Weiterhin ist die CLP vor allem für kombinatorische Probleme besonders geeignet. Teilweise können damit sogar kombinatorische Probleme gelöst werden, welche mit steigender Komplexität sonst unlösbar wären. Deswegen wird CLP vor allem dort angewendet, wo Ressourcen besonders effizient organisiert werden müssen, was heutzutage eine nicht selten zu bewältigende Aufgabe ist.
Leider konnte in dieser Ausarbeitung nicht jedes Detail der CLP beleuchtet werden, denn es gibt durchaus noch weitere Anwendungsmöglichkeiten und Variationen von CLP, jedoch kann man insgesamt festhalten, dass CLP eine geeignete Herangehensweise für viele Probleme ist und vor allem auch tatsächlich in der realen Welt Anwendung findet.

## Literatur


[Gie15] J. Giesl. Vorlesungsskript Logikprogrammierung. Bericht, Lehr- und Forschungsgebiet Informatik 2, RWTH Aachen, 2015.

[NM95] Ulf Nilsson und Jan Maluszyński. *Logic, programming, and Prolog*. John Wiley, West Sussex, 2.. Auflage, 1995.

[TS13] Frühwirth Thom und Abdennadher Slim. *Essentials of Constraint Programming*. Springer Science and Business Media, Berlin Heidelberg, 2013.


[TS13] [Gie15] [NM95]