Constraint Logic Programming

Justin Graß, Atakan Cubukcu, Pau Azpeitia Bergós

June 21, 2023

Abstract

Constraint Logic Programming (CLP) hilft der Logikprogrammierung ihre Möglichkeiten zu erweitern, da die Logikprogrammierung in Teilbereichen oft sehr eingeschränkt ist, wie zum Beispiel in der Arithmetik. Constraints können dort aushelfen, da sie die Möglichkeit bieten weiter Bedingungen beziehungsweise Einschränkungen an das Programm zu stellen, welche ebenfalls überprüft werden und auch gelten müssen damit das Programm eine passende Antwortsubstitution liefert. Deshalb ermöglichen Constraints der Logikprogrammierung komplexe Probleme einfacher zu lösen beziehungsweise sie lassen sich einfacher programmieren oder auch effizienter zu lösen, also mit weniger Ressourcen oder in schnellerer Zeit. Eine sehr wichtige Erweiterung der Programmiersprache "Prolog" ist "Constraint Logic Programming over Finite Domains" (CLPFD), diese erlaubt den Programmierer wie bekannt aus anderen Programmiersprachen arithmetische Ausdrücke auf den ganzen Zahlen auswerten zulassen, da man für so etwas in Prolog meist auf ein weiteres Prädikat zugreifen muss oder nicht der komplette Ausdruck ausgewertet wird, dies hilft dabei die Effizienz und Komplexität eines Programms in der Logikprogrammierung zu verbessern und dies hilft dann wiederum dem Programmierer seine Ideen einfacher zu implementieren.

1 Einführung

Bevor wir uns mit dem "Constraint Logic Programming" auseinandersetzen, schauen wir uns erst nochmal Probleme in Prolog an. Prolog hat oft Schwierigkeiten mit der Auswertung von arithmetischen Ausdrücken. Betrachten wir folgendes Beispiel:

```
?- 2=1+1.
```

Diese Anfrage gibt "false" zurück, da der Gleichheitsoperator "=" in Prolog nur überprüft, ob man die beiden Ausdrücke auf der linken und rechten Seite unifizieren kann und dies ist hier nicht der Fall, da es keine passende Antwortsubstitution gibt, damit diese beiden Ausdrücke gleich sind. Prolog hat aber, falls man mal einen Ausdruck auswerten möchte das "is"-Prädikat eingeführt. Betrachen wir nun das vorherige Beispiel, aber diesmal mit "is" anstatt "=":

```
?- 2 is 1+1.
```

Diese Anfrage wertet nun zu "true" aus. Betrachten wir aber noch folgendes analoges Beispiel merkt man wieder wie schwer die Arithmetik in Prolog sein kann.

```
?- 1+1 is 1+1.
```

Diese Anfrage wertet nun zu "false" aus, da das "is"-Prädikat nur die rechte Seite des Ausdruckes auswertet und danach versucht die beiden Ausdrücke miteinander zu unifizieren. Also lässt sich sagen, dass Prolog Probleme aufweist mit Zahlen beziehungsweise arithmetischen Ausdrücken zuarbeiten und da können Constraints helfen, denn betrachten wir die genutzten Beispiele mit der Nutzung von Constraints, lassen sich diese einfacher und einheitlicher lösen. Hier nun die Anfragen mit der Nutzung von Constraints:

```
?- 1+1 #= 1+1.
?- 2 #= 1+1.
```

Beide Anfragen werten nun zu "true" aus und das mit der Nutzung des selben Operators "#=", was hilfreich ist, da man sich dann nicht immer fragen muss welches Prädikat oder welchen Operator nun nutzen muss. Was genau Constraints sind und was ihr Nutzen ist, erfahren wir nächstes Kapitel.

2 Constraints

Das Wort "Constraint" bedeutet so viel wie "Einschränkung", dies reflektiert auch seinen Nutzen in Programmiersprachen. In Programmiersprachen sind Constraints als Bedingungen oder auch Einschränkungen bekannt, die meist in mathematischen oder logischen Kontext verwendet werden. Constraints werden im Normalfall als atomare Formeln angegeben. Sie werden meist genutzt um Variablen einzuschränken oder auch um Bedingungen an die Variablen zu stellen, aber sie können Variablen auch direkt einen bestimmen Wertebereich zuordnen. Wir betrachten im folgenden lediglich Constraints in der Logikprogrammierung, obwohl man sie ebenfalls in anderen Programmiersprachen nutzen kann. Ein Constraint in der Logikprogrammierung sieht zum Beispiel so aus:

```
X#<3
```

Dies ist ein kleines Beispiel für ein Constraint in Prolog mit "CLPFD" und in diesem Beispiel für der Wert der Variable X auf ganze Zahlen kleiner als 3 eingeschränkt, also mathematisch formuliert:

```
X<3
```

aber mehr zur "CLPFD" folgt nächstes Kapitel.
Constraints sind also meist Gleichung beziehungsweise Ungleichungen, die ebenfalls erfüllt sein müssen, damit das Programm eine Antwortsubstitution ausgibt beziehungsweise "true" ausgibt.
Damit man Constraints nutzen kann, braucht man eine Constraint-Theorie (CT), diese bildet die Grundlage für die Programmierung mit Constraints, da die "CT" aus einer Menge von Formeln besteht und wenn man einen Constraint aus diesen Formeln ableiten kann, kann man sagen dass der Constraint erfüllbar ist. Als Beispiel für eine Constraint-Theorie betrachten wir die "CLPFD" im nächsten Kapitel genauer.
Die "CT" liefert außerdem noch einen "constraint solver" der versucht die gegeben Constraints zu lösen. Um die Constraints zu lösen nutzt er verschiedene Lösungsverfahren beziehungsweise Algorithmen, die von der Constraint-Theorie abhängig sind, also jede Constraint-Theorie hat ihre eigenen Lösungsverfahren. Dann existiert noch ein weiteres zentrales Element bei der Arbeit mit Constraints und das ist der "constraint store". Die Aufgabe von ihm ist es in einem Programm die vorkommenden Constraints einzusammeln und die Korrektheit sicherzustellen, bedeutet wenn ein neuer Constraint zu dem constraint store hinzugefügt wird, muss überprüft werden, ob man den constraint store

noch erfüllen kann und wenn nicht den neu hinzugefügten Constraint wieder entfernen und "backtracking" zu betreiben, ob man eine andere Klausel nehmen kann. [Did] [Pie12] [And+07]

3 CLPFD

Nachdem Constraints formal eingeführt wurden, nutzen wir nun dieses Wissen um es auf die Logikprogrammierung zu übertragen, dazu führen wir zunächst "CLPFD" ein. Das Wort "CLPFD" steht ausgeschrieben für das Wort "Constraint Logic Programming over finite domains" und ist eine Constraint-Theorie die es ermöglicht die Arithmetik in der Programmiersprache "Prolog" einfacher und effizienter darzustellen, da es in Prolog oft sehr schwer ist mit Zahlen zuarbeiten, wie man im Kapitel "Einführung" festgestellt hat. Die "CLPFD" arbeitet nur mit Integern, also ganzen Zahlen, dies ermöglicht einfachere Auswertung der Constraints und reicht im wesentlichen für den Anwendungsbereich der "CLPFD". Um die Library der "CLPFD" in Prolog nutzen zu können, muss man folgende Direktive am Anfang des Programms einfügen:

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

Nachdem wir nun die Library der "CLPFD" importiert haben, stehen uns nun alle Eigenschaften der Library zur Verfügung. Nun folgt einen kleines Programm, was darstellt wie ein Programm mit Constraints in der "CLPFD" aussehen kann.

```
findValues(X,Y):-
X in 1..5,
Y in 1..5,
X + Y #= 5,
X #> Y,
label([X]),
label([Y]).
```

Dieses Beispielprogramm sucht mögliche Ergebnisse, so dass X+Y = 5 ist und gleichzeitig X größer als Y ist und ebenfalls X und Y beides Werte zwischen 1 und 5 sind. Aber wir betrachten nun erst einmal genau was die einzelnen Operationen in diesem Programm bewirken. Betrachten wir nun die ersten beiden Zeilen des vorher gegeben Programms:

```
X in 1..5,
Y in 1..5,
```

Diese beide Zeilen sind Constraints aus der "CLPFD", die den Wertebereich der beiden Variablen einschränkt. Dieser Constraint bewirkt, dass X und Y zwischen den Zahlen 1 und 5 liegen muss und dient daher als eine Art der Initialisierung der Variablen, da man ihnen schon die genannten Werte zuweist die danach weiter im Programm überprüft werden, ob die Variablen die weiter gegebenen Bedingungen ebenfalls erfüllen. Eine äquivalente mathematische Bezeichnung für die Constraints ist:

```
1<=X<=5
1<=Y<=5
```

Nun betrachten wir die folgenden zwei Zeilen:

```
X + Y #= 5,
X #> Y
```

Hier sieht man nun ein "#", dies bedeutet, dass es sich um einen Constraint der "CLPFD" handelt und hier die Domänen von X und Y eingeschränkt werden. Also bedeutet

```
X + Y #= 5,
```

nichts anderes als X + Y = 5. Analog für die zweite Zeile des Codeausschnitts, die sicherstellen soll, dass X größer als Y ist.
Nun betrachten wir noch die letzten beiden Zeilen unseres Beispielprogramm:

```
label([X]),
label([Y]).
```

Das "label" Prädikat ist ebenfalls ein Teil der "CLPFD" und schaut nun am Ende des Programms welche Lösungen für die gegeben Constraints beziehungsweise mit den von uns gegeben Bedingungen in Frage kommen, um somit auch eine konkrete Lösung für unser gegebenes Programm finden zu können. Mit diesen gegebenen Bedienungen versucht das "CLPFD" Modul nun mögliche Lösungen des Programms zu finden, wie genau man dort vorgeht oder auch vorgehen kann folgt im Laufe der Ausarbeitung.
Analog zu dem Beispielprogramm gibt es natürlich viele weitere arithmetische Ausdrücke die in der Constraint Logic Programming over finite domains ("CLPFD") vorhanden sind, diese aber analog zu dem Beispiel funktionieren und die gleiche Syntax nutzen.

Zusammenfassend lässt sich über die "Constraint Logic Programming over finite domains" (clpfd) sagen, dass sie eine sehr wichtige Constraint-Theorie für Prolog ist, da sie Prolog ermöglicht leichter arithmetische Operationen darzustellen und ebenfalls die Effizienz des Programms steigert, da man nun nicht extra Prädikate implementieren muss um gewisse Dinge in Programme implementieren zu können.[Jür22] [07a]

4 Lösungsverfahren in Prolog mit Constraints

Um nun in einem Prolog Programm mit Constraints eine mögliche Lösung zu finden gibt es verschiedene Möglichkeiten, das Ziel ist wie vorher schon erwähnt, dass man alle möglichen Antwortsubstitutionen für das gegebene Programm finden möchte. Wichtig zu erwähnen ist es auch, dass diese Lösungsverfahren nicht von Prolog selbst angewendet werden, sondern es dafür einen sogenannten "constraint solver" gibt der diese Techniken und Algorithmen dann anwendet, um an eine exakte Lösung der Constraints zu gelangen. Um alle möglichen Lösungen zu finden gibt es verschiedene Lösungsverfahren beziehungsweise Algorithmen, die dann sicherstellen, dass am Ende alle möglichen Lösungen gefunden werden. Daher stellen wir nun folgende Beispiele für einige gängige Lösungsverfahren vor:

1. Constraint-Propagation
Dies ist das grundlegendste Lösungsverfahren, da es die Wertebereiche konkret auf die gegebenen Constraints einschränkt. Da "CLPFD" nur auf ganzen Zahlen arbeitet, geht "CLPFD" bevor es überhaupt ein Constraint überprüft hat aus, dass die Lösungsmenge die ganzen Zahlen sind. Nun wird der erste Constraint überprüft (z.B. X #>=0), dann weiß man sofort, dass X größer gleich 0 sein muss, in dem gegeben Beispiel, somit kann man sofort den Wertebereich einschränken und alle Zahlen die kleiner als 0 sind dann aus dem Wertebereich für X streichen in dem gegebenen Beispiel. Dies wird dann auch für die anderen gegeben Constraints gemacht, bis man am Ende alle überprüft hat und die Wertebereiche der Variablen dementsprechend angepasst hat und somit die möglichen

Lösungen dort zu finden sind. Genaue Verfahren betrachten wir im Kapitel "Pfadkonsistenz". [Jür22]

Constraint-Propagation Beispiel

Betrachte folgendes Programm:

```
beispiel1(X,Y):-
X #>0,
Y #<0.
```

Dies ist ein Beispiel, wo gefordert wird, dass X echt größer als 0 ist und Y echt kleiner als 0. Am Anfang der Ausführung des Programms sind beide Domänen der Variablen noch auf den ganzen Zahlen gesetzt. Nun werden die beiden Constraints ausgewertet und bei X alle Zahlen die kleiner oder gleich 0 sind aus der Domäne verworfen, analog für die Domäne von Y. Stellt man die Anfrage:

```
?- beispiel1(X,Y).
```

Liefert Prolog die Antwortsubstitution:

```
X in 1..sup,
Y in inf.. -1.
```

Die Antwortsubstitution bedeutet, dass X von 1 bis "supremum" gehen darf, wobei "supremum" der größtmögliche Wert der Domäne ist. Analog für Y, wobei "inf" für "infimum" steht, was der kleinstmögliche Wert der Domäne ist.

2. Reduktionsregeln

Dieses Lösungsverfahren nutzt die mathematischen Eigenschaften der Constraints, um die Lösungsfindung zu erleichtern. Zum Beispiel wenn eine Variable einen Wert zugewiesen bekommt, dann wird der Wertebereich trivialerweise auf diesen Wert eingeschränkt. Dies scheint auf den ersten Blick nicht sehr gewinnbringend, jedoch wenn es zwei Variablen gibt, die beide einen eingeschränkten Wertebereich haben und diese nun miteinander addiert werden, so kann man den Wertebereich des Ergebnis der Addition ebenfalls auf die Addition dieser beiden Wertebereich der Variablen einschränken, so dass man nicht beim Wertebereich vom Ergebnis wieder die ganzen Zahlen als Wertebereich betrachten muss, sondern sofort einen präziseren zur Verfügung hat.Analog funktioniert dies mit der Multiplikation. [07b] [RBW06]

Reduktionsregel Beispiel

Betrachte Programm:

```
beispiel(Z):- X in 1..4,Y in 0..3, X+Y #= Z.
```

Wir stellen nun die Anfrage:

```
?- beispiel(Z).
```

dann liefert Prolog die Antwortsubstitution:

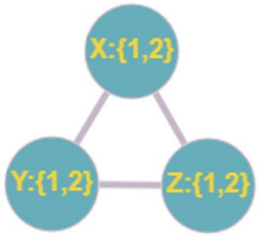
```
Z in 1..7.
```

Da er sich wie soeben erklärt, die Wertebereiche von X und Y anschaut und sofort den Wertebereich für Z einschränken kann.

3. Suche
Wenn keins der Lösungsverfahren der jeweiligen Constraint-Theorien zu einer Lösung führt, wird noch versucht eine Suche durchzuführen. In der Suche versucht man mit den jeweiligen Domänen die man hat eine passende Lösung zu finden, indem man mögliche Kombination der Variablen ausprobiert auf den gegebenen Domänen. Dies basiert mehr auf Glück und ist deswegen auch nicht effizient, weswegen sie nur als letztes probiert wird, wenn man vorher keine Lösung finden konnte.[RBW06]

5 Kantenkonsistenz

5.1 Kantenkonsistenz Beispiel



Der Graph hat Folgende Domänen für X,Y,Z: X[1,2], Y[1,2], Z[1,2]

Mit den globalen Constraints Y ¡ Z,X != Z, X = Y

Hier untersuchen wir immer nur zwei Knoten und die Kante dazwischen. Zuerst wird ein Wert für Y gewählt, damit Y > Z gilt, der gewählte Wert wäre 2. Dadurch können wir Z sofort einschränken zu 1, da gilt Y > Z, dadurch bekommen wir Z 1. Als Letztes können wir X einschränken, da gilt X ! = Z oder X = Y, dadurch bekommen wir X 2.

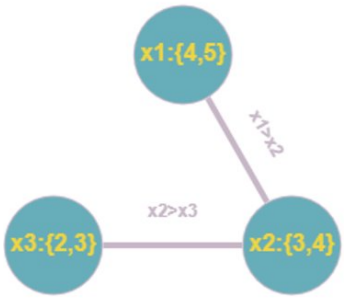
5.2 Kantenkonsistenz

Die Kantenkonsistenz ist genau wie die Pfadkonsistenz eine Technik einen Constraint-Graphen zu durchsuchen. Die Kantenkonsistenz ist eine Erweiterung der Knotenkonsistenz, die Knotenkonsistenz ist eine sehr einfach Technik die Knotenkonsistenz bezieht sich auf unäre Constraints. Die Definition wäre, wenn eine Variabel x und D der Wertebereich sei, sei C(x) ein auf x definierter unärer Constraint, das heißt die Variabel x ist knotenkonsisent Falls gilt: $\forall$d $\in D : d \in C(x)$, dass heißt ein Constraint-Graph ist Knotenkonsistenz, wenn falls jede Variabel des Constraint-Graphen knotenkonsistent ist[SS94]. Die Kantenkonsistenz bezieht sich auf binäre Constraints. Die Definition wäre, Wenn x_i und x_j Variablen wären mit dem Wertebereich D_i und D_j, dann sei C(x_i, x_j) eine auf x_i und x_j definierter Constraint.[SS94]

6 Pfadkonsistenz

6.1 Pfadkonsistenz Beispiele

Im Folgenden betrachten wir Beispiele, um die Pfadkonsistenz zu veranschaulichen.
In dem Beispiel haben wir die Domänen der Variablen x1,x2,x3, diese sind wie folgt definiert:



1. x1: [4,5]
2. x2: [3,4]
3. x3: [2,3]

Ihre Beziehungen, also die globalen Constraints sind:

1. x1 > x2, x1 muss immer größer x2 sein
2. x2 > x3, x2 muss immer größer x3 sein

Bevor die Pfadkonsistenz angewandt wird, kann man sagen das der Baum noch nicht Pfadkonsistent ist. Wir nutzen nun die Pfadkonsistenz und überprüfen die Kanten und Knoten des Constraint-Graphen. Es gibt nur einen Pfad der Länge Zwei: x1-x2-x3. Wir gehen den Pfad x1-x2-x3 entlang und prüfen die möglichen Wertekombinationen für x1, x2, x3, falls die Wertekombinationen Pfadkonsistent sind können wir ein Constraint zwischen x1 und x3 bilden. Zu erst bilden wir das kartesische Produkt aller Relationen zwischen x1 und x3.

1. [4,2]
2. [4,3]
3. [5,2]
4. [5,3]

Nun gehen wir mit allen Relation den Pfad x1-x2-x3 lang, wir beachten dabei dir Constraints. Die übrig gebliebenen Relationen sind:

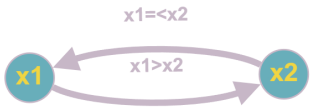
1. [4,2]
2. [5,2]
3. [5,3]

Durch die Anwendung der Constraints haben wir die Domänen eingeschränkt und dadurch haben wir die Pfadkonsistenz für die drei übrig gebliebenen Relationen gezeigt. Als letzten Schritt führen wir einen "Constraint" zwischen x1-x3 ein damit die Relationen schneller und effizienter nach x3 kommen.
Die Pfadkonsistenz ist gut um Ergebnisse zu filtern" und mögliche Inkonsistenzen in deinem Constraint-Graphen zu identifizieren und die Wertedomänen der Variablen einzuschränken, dadurch können wir Ergebnisse die wir wollen effizienter und schneller bekommen. Man

Hier wurde zwischen x1 und x3 ein Constraint eingefügt

kann die Pfadkonsistenz auch bei komplexeren Bereichen anwenden, solange man die Bedingungen der Pfadkonsistenz nicht bricht, ist es ein Werkzeug, welches die Logikprogrammierung vereinfacht und erweitert, sodass man komplexere Programme effizienter lösen kann. Dies war ein Beispiel, wo die Pfadkonsistenz zu einem korrekten Ergebnis gekommen ist, das heißt nicht das es immer zu einem korrekten Ergebnis kommen muss, die Pfadkonsistenz ist mit Vorsicht zu genießen. Man sollte immer prüfen, ob das Ergebnis auch stimmt, da es auch zu falschen Einschränkungen kommen kann, da die Constraints widersprüchlich sind.

Hier ist ein Beispiel wo die Pfadkonsistenz den Graphen falsch einschränkt.



Mit den globalen Constraints x1 > x2, x1 =< x2

Hier kann man sehen, dass die Constraints, eigentlich sich gegenseitig aufheben sollten. Das Problem ist durch die Constraints finden wir hier immer Werte für wo die Constraints gelten. Hier haben wir einen Constraint-Graphen der Pfadkonsistenz ist, aber die Constraints einander widersprechen, weshalb man die Constraints mit Vorsicht wählen sollte.

6.2 Pfadkonsistenz Theroie

Es gibt verschiedene Techniken, um Domänen mit der Constraint Programmierung einzuschränken. Die Pfadkonsistenz wurde aus der Kantenkonsistenz entwickelt, da die Kantenkonsistenz nicht alle Inkonsistenzen aus Domänen eingeschränkt hat und ein Kantenkonsistenz Constraint-Netz auch keine Lösung haben kann, wurde aus ihr die Pfadkonsistenz entwickelt, die als Erweiterung der Kantenkonsistenz dient. Bei der Pfadkonsistenz werden Kanten und Knoten betrachtet, diese bilden ein Constraint-Netz oder anders genannt Constraint-Graphen, diese Kanten und Knoten haben Variablen, die sie miteinander verbinden, die Kanten und Knoten kann man durchlaufen, die Strecke die durchlaufen wurde, würde als Pfad bezeichnet werden, ein Pfad kann mehrmals denselben Knoten entlang und auch hintereinander zum selben Knoten. Eine allgemeine Formulierung wäre, dass ein Pfad P=$(v_1, \ldots, v_i, \ldots, v_m)$ pfadkonsistent ist, wenn sie binär Constraint für alle Wertekombinationen für$(v_1, \ldots, v_m)$, sodass man für v_i Werte finden kann, sodass die Constraints $C_{(1,1)}, \ldots .., C_{(i,i+1)}, \ldots .., C_{(m-1,m)}$. Zusammenfassend kann man sagen, dass die

Pfadkonsistenz als Erweiterung der Kantenkonsistenz entwickelt wurde, um Inkonsistenzen zu beheben, wie im Text genannt. Indem sie die Beziehungen der Constraint entlang der Pfade betrachtet, dies ermöglicht eine präzisere Einschränkung der Domänen innerhalb des Constraint-Graphen.[Jür22][Wol06]

Ein Constraint ist pfadkonsistent, wenn die Pfade im Constraint-Graph pfadkonsistent sind. Man muss beachten, dass die Pfadkonsistenz für den Pfad P, wie oben definiert, nicht vorschreibt, dass die Wertzuweisungen für diese Variablen auch mit Constraints übereinstimmen müssen, die nicht in dem Pfad sind. Es ist also nur eine lokale Konsistenz. Die lokale Konsistenz hat auch gute Eigenschaften, die Effizienz gesteigert wird, da nur die unmittelbar mit dem Pfad verbundenen Constraints berücksichtigt werden. Durch die kleineren Domänen kann man sich auf spezielle Bereiche eines Netzwerkes fokussieren, um lokale Konsistenzen zu sichern. Darüber hinaus ermöglichen lokale Konsistenzen auch eine modulare Herangehensweise, da wir nur bestimmte Pfade prüfen, können Änderungen oder Erweiterungen in einem Teil des Graphen gemacht werden, ohne den Rest des Graphen zu beeinflussen. Es gibt eine negative Eigenschaft von lokale Konsistenzen, wenn man viele lokale Konsistenten zusammenbringt, müssen diese keine globale Konsistenz bilden, diese Fehler“ wurde widerlegt, es wurde bewiesen von Ugo Montanari, in Montanari, im Jahr 1974, dass man Pfadkonsistenz für vollständige Constraint-Graphen erreichen kann, es müssen lediglich alle Pfade der Länge 2 pfadkonsistent sein. Eine verkürzte Form von seiner Definition wäre[Wol06]:

$$\begin{aligned} \forall v_i, v_j \quad &: \quad D_i \subseteq R_i^u \wedge D_j \subseteq R_j^u \\ &\wedge \quad (d_i, d_k) \in R_{i,k}^b \wedge (d_k, d_j) \in R_{k,j}^b \end{aligned}$$

[Wol06]
Diese Formel sagt aus, dass die Variablen v_i und v_j spezifische Bedingungen erfüllen müssen, damit sie "pfadkonsistent" sind. Die Bedingungen sind, dass die Domänen der Variablen in ihren jeweiligen oberen Relationen enthalten sein müssen und es müssen spezifische Beziehungen zwischen den Werten in den Unterrelationen existieren.

7 Constraint Vereinfachung

Um Constraints zu vereinfachen, bedienen wir uns an einen von vielen Techniken, die Technik, die wir verwenden werden sind, die CHR-Regeln. Diese Regeln sind deklarativ und werden im Regelformalismus geschrieben. Im Folgenden können sie die 3 CHR-Regeln finden[TS97]:

Reflexivität @ X =< Y <=> X = Y | true.
Antisymmetrie @ X =< Y, Y =< X <=> X = Y.
Transitivität @ X =< Y, Y =< Z ==> X =< Z.
[TS97]
Diese Regeln sind schemenhaft anwendbar. Die erste Regel besagt, dass X =< Y ist, wenn X=Y ist. Diese Vorbedingung muss erfüllt sein, um die Anwendung einer Regel zu gewährleisten, das wird "Wächter“ genannt. Durch die Reflexivitäts-Regel kann für alle B =< B, *true* eingesetzt werden. Durch diese Regel kann geprüft werden, ob ein Constraint erfüllbar ist. Die zweite Regel besagt, dass man die Constraints, die so aussehen wie "Y =< X, X =< Y" zu X=Y vereinfachen kann. Also, dass zwei Ungleichheitsconstraints zu einem Gleichheitsconstraint umgewandelt werden können. Die zweite Regel hat keine

Vorbedingung, also "Wächter", also kann man diese immer anwenden.
Die dritte Regel besagt, dass "X =< Y, Y =< Z" zu dem Constraint "X =< Z" umgewandelt werden darf. Die ersten beiden Regeln werden genutzt, um Constraints zu vereinfachen oder zu lösen, die dritte Regel wird verwendet, um logische Konsequenzen als redundante Constraints einzufügen.[TS97]

7.1 Constraint Vereinfachung Beispiel

Wir fangen an mit den Constraints A =< B, C =< A, B =< C. Hier können wir erkennen, dass wir die Transitivität benutzen können, um hier einzuschränken, hier kann man wegen C =< A, A =< B die Transitivität benutzen, weshalb wir C =< B einfügen dürfen. Als zweiten Schritt benutzen wir die Antisymmetrie, hier fügen wir B = C ein, die Antisymmetrie dürfen wir benutzen, weil C =< B, B =< C ist. Als dritten und letzten Schritt können wir wieder die Antisymmetrie nutzen, weil wir B =< A haben und A =< B, weshalb wir wie in Schritt 2 A=B einfügen können. Nachdem wir alles vereinfacht haben, haben wir das Ergebnis A = B, B = C. Die Vereinfachung sollte man benutzen, um einen komplexen Constraint so weit wie möglich zu vereinfachen, damit man ihn so einfach wie möglich implementieren und nutzen kann. In dem Abschnitt 5.1 haben wir gezeigt, dass widersprüchliche Constraints nicht als Fehler erkannt werden, weshalb die Fehlersuche bei einem komplexen Constraint-Graphen lange dauern könnte, falls sie die Constraints, aber vereinfacht haben, ist die Wahrscheinlichkeit geringer, dass ein Fehler auftaucht.

8 SLD-Baum

Ein SLD-Baum (Selective Linear Definite Clause Resolution Tree) ist ein Baumdiagramm, das verwendet wird, um die Ausführung von logischen Programmierungen zu visualisieren. Es stellt eine Methode dar, um die Erfüllbarkeit einer Anfrage durch eine Reihe von Resolutionschritten zu prüfen. Der Baum zeigt die verschiedenen Kombinationen von Regeln und Fakten, die angewendet werden, um eine Lösung zu finden oder zu widerlegen.

Der SLD-Baum bezieht sich auf den Ableitungsbaum, der durch den SLD-Resolver-Algorithmus generiert wird. Der SLD-Baum-Algorithmus basiert auf der Idee, einen Suchprozess in einem Ableitungsbaum zu verwenden, um eine logische Anfrage zu lösen. Dieser Baum zeigt die potenziellen Inferenzpfade, die zur Lösung der Anfrage führen können. Jeder Baumknoten zeigt eine Programmklausel und ihren Unifikationszustand mit der Anfrage. Die Konstruktion des Ableitungsbaums beginnt mit der ersten Anfrage als Wurzel des Baums. Daraufhin werden die logischen Programmregeln angewendet, um den Baum zu erweitern, indem neue Klauseln hinzugefügt werden, die die bereits vorhandenen Klauseln unifizieren. Bis eine Lösung gefunden wird oder es keine weiteren Zweige zur Erkundung gibt, wird dieser Prozess wiederholt. Der SLD-Baum hilft beim Verständnis und Debuggen des logischen Inferenzprozesses, indem er die SLD-Resolution visuell darstellt. Jeder Zweig des Baums repräsentiert eine Sequenz von Resolutionsschritten, die entweder zu einer Lösung führen oder zu einem Scheitern der Ableitung. Bei der SLD-Resolution wird selektiv vorgegangen, um eine effiziente Suche zu ermöglichen. Dadurch wird der Suchraum reduziert und die Effizienz des Algorithmus verbessert.
Es ist wichtig zu beachten, dass der SLD-Baum aufgrund der nichtdeterministischen Natur der logischen Programmierung potenziell unendlich sein kann. Dies liegt daran, dass es in der logischen Programmierung möglicherweise mehrere Lösungen für eine Anfrage gibt. Daher müssen Strategien angewendet werden, um eine endlose Exploration zu vermeiden,

wie zum Beispiel die Begrenzung der Anzahl von Resolutionsschritten oder die Verwendung von Cut-Strategien (wie dem Cut nach Scheitern).

Zusammenfassend ist der SLD-Baum ein Baumdiagramm, das die Ausführung von logischen Programmierungen visualisiert. Er zeigt die Resolutionschritte und Kombinationen von Regeln und Fakten zur Lösungsfindung. Der Baum basiert auf dem SLD-Resolver-Algorithmus und hilft beim Verständnis und Debuggen des Inferenzprozesses. Durch den selektiven Ansatz der SLD-Resolution wird der Suchraum reduziert und die Effizienz verbessert. Es ist jedoch wichtig, Strategien anzuwenden, um endlose Exploration zu vermeiden.[SS94]

8.1 SLD-Baum Beispiel

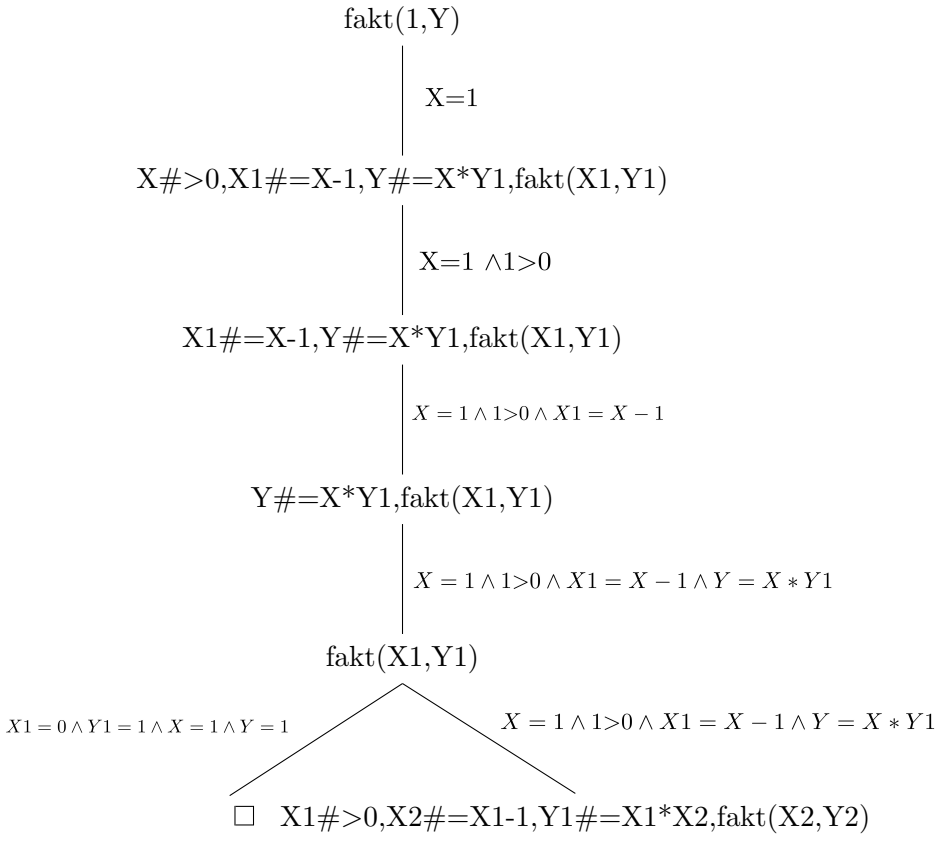
Nachdem wir nun den SLD Baum formal eingeführt haben, folgt nun ein Beispiel mit Erläuterung. Betrachten wir nun folgendes Beispielprogramm in Prolog mit "CLPFD":

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

Dies ist ein Programm, welches die Fakultät einer Zahl ausrechnet oder da Logikprogrammierung bidirektional auswertbar ist, lässt sich von einer gegebenen Zahl bestimmen ob, sie die Fakultät von einer Zahl ist. Des weiteren betrachten wir die folgende Anfrage:

```
?- fakt(1,Y).
```

Diese Anfrage soll nun prüfen, was '1!' ergibt. Aus der gegebenen Anfrage lässt sich nun folgender SLD-Baum ableiten:



Der Baum funktioniert so, dass er erst schaut ob es eine passende Substitution für den Fakt "fakt(0,1)" gibt, dies ist nicht der Fall, also nimmt er die Regel für "fakt(X,Y)", dann ersetzt man den Term durch die gegebene rechte Seite der Regel. Dann nimmt man schrittweise alle gegeben Constraints in den "constraint store" (Kanten des SLD-Baums) auf als Konjunktionen, so dass dann der "constraint solver" gegeben falls Lösungen für die vorliegenden Constraints finden kann. Dann bleibt schließlich "fakt(X1,Y1)" übrig, dies kann man nun mit "fakt(0,1)" unifizieren, wenn man X1 mit 0 substituiert und Y1 mit 1 substituiert, damit findet man nun eine Lösung für die gegebene Anfrage. Dieser Baum liefert dann die Antwortsubstitution:

```
Y = 1.
```

Da sich im ganz rechten Pfad die Constraints "X1#>0, X2 #= X1-1, Y1#= X1*Y1" nicht mehr lösen lassen, gibt Prolog danach "false" aus und stoppt somit die weitere Auswertung. [Jür22]

9 Zusammenfassung

Zusammenfassend kann man zur Constraint Logic Programming (CLP) sagen, dass es der Logikprogrammierung sehr hilft, da man nicht nur mit der aus der Logikprogrammierung bekannten Prädikatenlogik und mit der etwas komplizierten Arithmetik arbeiten muss, sondern auch noch die Möglichkeit bekommt, andere Constraint-Theorien zu nutzen, wie in der Ausarbeitung vorgestellten "CLPFD" die einem die Möglichkeit gibt, weitere Bedingungen beziehungsweise Einschränkungen als Form von Constraints an das Programm zustellen, ohne sich dabei Sorgen um die Implementation zumachen, da man möglicherweise für diese Bedingungen ohne eine Constraint-Theorie wieder ein extra Prädikat erstellen müsste, was zum einem auf die Effizient des Programms geht und zum anderen den Code nicht sehr übersichtlich gestaltet. Ein weiterer Vorteil ist auch, dass man wenn man ohne Constraint-Theorie arbeitet und sich extra Prädikate für seine Operationen erstellt, dass es fehleranfälliger ist, da es natürlich öfters zu Fehler in der Implementation eines Programms führt, vor allem wenn man dort sehr viele verschiedene Prädikate nutzen muss, dies spricht dann natürlich für die Nutzung einer Constraint-Theorie, wo man beispielsweise mit einem Ausdruck einer Variable sofort einen Wertebereich zuweisen kann und nicht ein Prädikat dafür anfertigen muss. Daher kann man sagen, dass es sich lohnt wenn man in der Logik programmiert, sich mit Constraint-Theorien auseinanderzusetzen, da sie einem die Arbeit sehr erleichtern können.

References

[07a] "Constraints über endlichen Wertebereichen — Finite-Domain-Constraints". In: *Einführung in die Constraint-Programmierung: Grundlagen, Methoden, Sprachen, Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 71–96. ISBN: 978-3-540-68194-6. DOI: 10.1007/978-3-540-68194-6_4. URL: https://doi.org/10.1007/978-3-540-68194-6_4.

[07b] "Constraints und Constraint-Löser". In: *Einführung in die Constraint-Programmierung: Grundlagen, Methoden, Sprachen, Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 53–70. ISBN: 978-3-540-68194-6. DOI: 10.1007/978-3-540-68194-6_3. URL: https://doi.org/10.1007/978-3-540-68194-6_3.

[And+07] H. R. Andersen et al. "A Constraint Store Based on Multivalued Decision Diagrams". In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Christian Bessière. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 118–132. ISBN: 978-3-540-74970-7.

[Did] Didia. *Constraint*. URL: https://de.wikipedia.org/wiki/Constraint. (accessed: 31.05.2023).

[Jür22] Giesl Jürgen. "Logikprogrammierung". Script. Aachen: Rheinisch-Westfaelische Technische Hochschule Aachen, Sommersemester 2022.

[Pie12] Thomas Piecha. "Theoretische Grundlagen der Logikprogrammierung". Script. Tübingen: Universitat Tübingen ¨ Wilhelm-Schickard-Institut für Informatik, Sommersemester 2012.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006. ISBN: 978-0-444-52726-4. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

[SS94] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[TS97] Fruhwirth Thom and Abdennadher Slim. "Der Mietspiegel im Internet Ein Fall für Constraint-Logikprogrammierung". Diplomarbeit. Oettingenstraße 67, 80538 München: Institut für Informatik, Ludwig-Maximilians-Universität, 1997.

[Wol06] Runte Wolfgang. "Ein hybrides Framework für Constraint-Solver". Diplomarbeit. Bremen: Universität Bremen, 27. Januar 2006.