INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians-
Universität
München

Analyse von regelbasierten Constraintlösern

Slim Abdennadher

http://www.pms.informatik.uni-muenchen.de/publikationen
Dissertation/PhD Thesis PMS-DISS-1998-1, Mai 1998

A mes parents Hassiba et Tijani
A mon épouse Nabila

Analyse von regelbasierten Constraintlösern

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik und Informatik
der Ludwig-Maximilians-Universität München

von

Slim Abdennadher

Tag der Einreichung:	21.04.1998
Tag der mündlichen Prüfung:	18.05.1998
Berichterstatter:	Prof. Dr. François Bry
	Prof. Dr. Martin Wirsing

Danksagung

An erster Stelle möchte ich meinen Doktorvater François Bry danken. Er hat mich in spannende Forschungsgebiete eingeführt und hat mir stets freundschaftlich mit Rat und Tat zur Seite gestanden. Für die Begutachtung der Arbeit schulde ich auch Martin Wirsing meinen Dank.

Meinem Freund und Kollege Thom Frühwirth verdanke ich unzählige Stunden mit und ohne Wissenschaft. Er hat einen großen Anteil am Gelingen dieser Dissertation, denn er hat in mir die Freude an wissenschaftlicher Arbeit geweckt.

Darüberhinaus gilt mein Dank insbesondere meine Studenten Holger Meuss und Michael Marte für die Diskussionen und Anregungen zum Thema dieser Arbeit. Auf ihre Unterstützung konnte ich mich stets verlassen.

Norbert Eisinger gebührt besonderen Dank für sein sorgfältiges Korrekturlesen der allerersten Version dieser Arbeit, seine konstruktiven Kommentare und nicht zuletzt für seine moralische Unterstützung. Aber auch François Bry, Thom Frühwirth, Holger Meuss und Martin Wirsing haben frühere Versionen ganz oder teilweise gelesen und durch ihre Anmerkungen zur Konsistenz beigetragen.

Bei allen Mitarbeitern des Lehrstuhls Programmier- und Modellierungssprachen am Institut für Informatik an der Ludwig-Maximilians-Universität München möchte ich mich für eine angenehme und engagierte Arbeitsatmosphäre bedanken. An dieser Stelle möchte ich Thomas Brüggemann, Tim Geisler, Martin Josko, Ellen Lilge, Sven Panne, Heribert Schütz und Sunna Torge nennen.

Besonderer Dank gebührt meinen Eltern Hassiba und Tijani, die mich auf meinem Weg begleitet haben und mir den Zugang zur akademischen Laufbahn ermöglichten. Ein dickes Dankeschön haben sich auch meine Brüder Salem und Lassad und seine Frau Dorra verdient, mitdenen ich schöne Zeiten während meiner Urlaube in Tunesien verbracht habe, um Energie und Motivation zu tanken.

Vor allem aber verdienen meine wunderbare Frau Nabila und meine Tochter Shirine ein riesiges Dankeschön, daß sie in den vergangenen Monaten so viel Verständnis für mein akademisches Grundbedürfnis aufgebracht haben.

Abstract

Constraint Handling Rules (CHR) [Frü95] is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers.

In this work we refine the operational semantics presented in [Frü95]. The new operational semantics is more faithful to the actual implementations of CHR. Then we give soundness and completeness results for CHR programs.

Typically, more than one CHR rule is applicable to a conjunction of constraints. It is obviously desirable that the result of a computation in a solver will always be the same, semantically and syntactically, no matter which of the applicable CHR rules is applied. This essential syntactical property of any constraint solver will be called confluence and will be investigated in this work. Confluence ensures that the solver will always compute the same result for a given set of constraints independent of which rules are applied. It also means that it does not matter for the result in which order the constraints arrive at the constraint solver. We will introduce a decidable, sufficient and necessary syntactic condition for confluence of terminating CHR programs. This condition adopts the notion of critical pairs as known from term rewriting systems. A straightforward translation of the results in this field was not possible, because the CHR formalism gives rise to phenomena not appearing in term rewriting systems. Moreover, as will be shown in this work, confluence of a program implies consistency of its logical meaning (under a mild restriction).

After introducing the notion of confluence for CHR, we investigate completion methods to make a non-confluent CHR program confluent (if possible).

Finally, we present practical applications of constraint programming implemented with CHR. The focus of this work is to show that simplicity, flexibility, efficiency and rapid prototyping are advantages of CHR.

Résumé

Constraint Handling Rules (CHR) [Frü95] est un langage de haut niveau permettant le developpement de solveur de contraintes ainsi que la modification de solveurs existants.

Dans ce mémoire nous présentons une version précisée de la sémantique operationnelle proposée dans [Frü95]. Cette nouvelle définition est plus conforme aux implémentations actuelles de CHR. Nous présentons également des résultats de correction et de complétude de programmes CHR.

En général, plus d'une CHR règle est applicable à une conjonction de contraintes. Il est donc préférable d'avoir le même résultat, en termes syntaxtiques et sémantiques, indépendamment de la règle choisie lors de l'execution. Cette propriété syntaxique essentielle pour tout solveur de contraintes, qu'on appelera confluence, est au cœur de ce travail. La confluence garantit que le solveur calcule un résultat unique pour un ensemble de contraintes donné. Cela implique également que l'ordre de prise en compte des contraintes par le solveur n'a pas d'effet sur le résultat. Nous définissons une condition syntaxique décidable, nécessaire et suffisante pour la confluence de programmes CHR dont l'exécution termine. Cette condition s'appuie sur la notion de paires critiques utilisée pour les systèmes de réécriture de termes. Bien que semblable à la confluence en réécriture, la confluence de programmes CHR ne se réduit pas à celle-ci. Il est aussi prouvé dans ce mémoire que la confluence d'un programme implique (sous une faible restriction) la consistance de sa signification logique.

Après l'étude de la confluence, nous nous intéressons aux méthodes permettant de rendre confluent un programme CHR qui ne l'est pas (si cela est possible).

Finalement, nous présentons des applications pratiques de la programmation par contraintes utilisant le langage CHR. Le but est de montrer que la simplicité, la flexibilité, l'efficacité et le prototypage rapide sont des avantages de CHR.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Begriff „Constraint-Logikprogrammierung“ (CLP) [vH91, FHK+92, JM94, FA97b] bezeichnet eine Familie von Programmiersprachen, die in der zweiten Hälfte der achtziger Jahre als eine natürliche Fusion zweier deklarativer Paradigmen entstand, nämlich Logikprogrammierung und Constraintlösen.

Logikprogrammierung

Die Idee der Logikprogrammierung (LP) ist, Probleme logisch zu beschreiben, also in einem deklarativen Programmierstil [SS86, Llo87]. Relationen zwischen Werten werden durch Prädikate beschrieben. In diesen Programmiersprachen werden das zum Problem gehörige allgemeine Wissen und die konkreten Annahmen durch eine eingeschränkte Klasse von logischen Formeln, den Hornklauseln, angegeben. Zu den Hornklauseln gehören Regeln und Fakten. Das Problem selbst wird ebenfalls als Klausel ausgedrückt, die sogenannte Anfrage. Diese Formeln bilden dann die Spezifikation des Problems. Ein festgelegter Algorithmus, der auf dem Resolutionsprinzip [Rob65] basiert, liefert dann selbsttätig einen Beweis für die Anfrage aus den zugrundeliegenden Regeln und Fakten. Durch ihre abstrakte deklarative Natur eignen sich LP-Sprachen gut für das „Rapid Prototyping“, d.h. die schnelle Erst- und Weiterentwicklung von Prototypen auf der Basis unvollständiger Spezifikationen.

Beispiel 1.1 Wir wollen folgenden Sachverhalt – in Abwandlung eines bekannten Syllogismus – ausdrücken: Alle griechischen Philosophen sind berühmt. Diese Aussage wird durch eine Regel modelliert. Zusätzlich haben wir Tabellen mit Griechen und Philosophen gegeben. In Prolog werden nur Variablennamen mit großem Anfangsbuchstaben geschrieben. Zeilen, die mit einem Prozentzeichen beginnen, sind Kommentare.

```
% Programm

% Regeln fuer Zusammenhaenge (allgemeines Wissen)

beruehmt(Mensch) :- grieche(Mensch), philosoph(Mensch).

%  Fakten fuer die Tabelle (konkrete Annahmen)

grieche(platon).
grieche(plexousakis).
grieche(sokrates).

philosph(platon).
philosoph(sokrates).
```

```
philosoph(wittgenstein).
```

Die Regel liest man als: „Ein Mensch ist dann berühmt, wenn dieser Mensch Grieche ist und er Philosoph ist“. Das Symbol „`:-`“ entspricht der logischen Implikation („$\Leftarrow$“) und „,“ der Konjunktion. Die Regel ist allquantifiziert, wobei die Allquantoren nicht explizit hingeschrieben werden. Nun fragen wir uns, ob Sokrates berühmt ist:

```
% Anfrage

:- beruehmt(sokrates).
   ja.                        % Antwort
```

Die Antwort ist erwartungsgemäß positiv. In LP-Sprachen ist es aber auch möglich, Parameter ungebunden (ohne bestimmten Wert) zu lassen. Wir können uns also auch fragen, wer berühmt ist.

```
:- beruehmt(Wer).
   Wer = sokrates;
   Wer = platon.
```

Constraint-Logikprogrammierung

Unter Constraintlösen versteht man das Lösen von Problemen, indem man „Constraints“ angibt, die von einer Lösung erfüllt werden müssen. Constraints sind Nebenbedingungen, die in deklarativer Weise formuliert werden und die die Suche nach geeigneten Lösungen im Vergleich zu den von herkömlichen Programmiersprachen gebotenen Mitteln beschleunigen. Constraints eignen sich zur Darstellung unvollständiger Information, also zur Beschreibung der Eigenschaften und Beziehungen von teilweise unbekannten Objekten.

Constraints wurden in den siebziger Jahren im Bereich der Künstlichen Intelligenz erforscht, um Probleme in der Bilderkennung und Zeitplanung lösen zu können. Dies geschah in Form von Constraint-Netzwerken, die auf Konsistenz und mögliche Lösungen untersucht werden sollen [Kum92, Mac92]. Um nicht für jede Instanz eines Problems ein neues Lösungsverfahren angeben zu müssen, gab es Ende der siebziger Jahre erste Versuche, gewisse Klassen von Constraints in Werkzeugen und Programmiersprachen verfügbar zu machen. Damit Constraintlösen in Programmiersprachen integriert werden kann und damit Constraints und ihre Lösungen einen Einfluß auf den Ablauf von Programmen haben können, muß es ein Programmstück geben, das die Constraints verwaltet und löst. Das ist der Constraintlöser.

Insbesondere hat die Einbettung von Constraintlösern in logische Programmiersprachen zu neuen Sprachen geführt, den sogenannten Constraint-Logikprogrammiersprachen (CLP-Sprachen). Mit dieser Kombination wurde es möglich, schnell und elegant komplexe kombinatorische Probleme durch eine Verbindung aus Constraintlösen und Suche zu lösen. Anwendungsgebiete sind Produktions- und Ressourcenplanung (insbesondere Zeit- und Kapazitätswirtschaft), Transportoptimierung, Layoutgenerierung, CAD-Systeme, Personalplanung und andere.

Die Idee der Constraint-Logikprogrammierung ist also, daß gewisse Prädikate als Constraints deklarativ formuliert und effizient durch spezielle Algorithmen behandelt werden können. Das heißt, daß die allgemeine Methode der Resolution und spezielle Methoden, für die die Constraintlöser zuständig sind, flexibel kombiniert werden können. Während mit der Resolution nach Lösungen gesucht wird, berechnet ein Constraintlöser die Lösung deterministisch nach einem speziellen, effizienten Verfahren. Bei der Implementierung von Constraints finden komplexe algorithmische Techniken aus dem Operations Research Verwendung. Die Suche nach Lösungen wird durch heuristikbasierte Verfahren unterstützt, die spezifische Information aus Anwendungsgebieten ausnutzen. Wartung und Erweiterung von Anwendungen wird durch die Trennung von Problemrepräsentation und Lösungssuche erleichtert. Die Constraints werden im Zuge der Abarbeitung eines Programmes inkrementell dem sogenannten Constraintspeicher hinzugefügt. Dem Constraintlöser obliegen u.a. die Aufgaben, den Constraintspeicher bei Hinzukommen neuer Constraints zu aktualisieren und zu vereinfachen und u.a. zu prüfen, ob die im Speicher vorliegenden Constraints schon ausreichen, den Wert einer Variable eindeutig zu bestimmen.

Beispiel 1.2 Wir definieren eine Maximum-Relation in einer CLP-Sprache mit Gleichheits- und Ungleichheitsconstraints. Der Einfachheitshalber habe diese Sprache die gleiche Syntax wie die LP-Sprache im vorigen Beispiel. Das Prädikat `max(X,Y,Z)` trifft genau dann zu, wenn `Z` das Maximum von `X` und `Y` ist.

```
max(X,Y,Z) :- X ≤ Y, Y = Z.
max(X,Y,Z) :- X > Y, X = Z.
```

Das Prädikat `max(X,Y,Z)` wird durch zwei Regeln, die implizit allquantifizert sind, definiert: Entweder gilt `X ≤ Y` und `Y = Z` oder `X > Y` und `X = Z`. Die Anfrage `max(3,5,M)` liefert das erwartete Ergebnis `M = 5` durch Anwendung der ersten Regel. Die Anwendung der zweiten Regel liefert das inkonsistente Constraint `3 > 5, 3 = Z` und scheitert daher. In Prolog und herkömmlichen Programmiersprachen kann $\leq$ nur getestet werden, wenn beide Argumente bekannt sind. Die Anfrage `max(N,3,5)` liefert daher in Prolog eine Fehlermeldung, obwohl man im allgemeinen durchaus „rückwärts“ rechnen kann. Mit Constraints ist die Deklarativität wieder hergestellt: Die Anfrage läßt sich mit der zweiten Regel zu `N > 3, N = 5` reduzieren. Diese Constraints vereinfachen sich zur Antwort `N = 5`. Die Anfrage `max(5,N,3)` scheitert, da keine der Regeln erfüllbare Constraints liefert. Sogar die Anfrage `max(M,N,N)` liefert bei der Anwendung der ersten Regel eine Antwort, nämlich eine intensionale, das Constraint `M ≤ N`.

Wie ausdrucksstark eine CLP-Sprache ist, wird an der Funktionalität und Qualität des Constraintlösers gemessen. Meistens wird der Constraintlöser als „Blackbox“ zur Verfügung gestellt. Die Benutzer haben in die Implementierung dieses Lösers keinen Einblick. Dies erschwert seine Modifizierung und seine Erweiterung. Die Erfahrung mit kommerziellen Anwendungen hat aber gezeigt, daß die meisten Probleme durch eine heterogene Kombination von Constraints charakterisiert werden, die noch dazu adaptiert bzw. ergänzt werden müssen, um den speziellen Anforderungen der Anwendung optimal Rechnung zu tragen.

Im CLP-Bereich gibt es einen Ansatz, Constraintlöser explizit über sogenannte Constraint Handling Rules (CHR) [Frü95] zu implementieren. Mit CHR kann ein Benutzer selbst Constraintlöser für einen beliebigen Constraintbereich entwickeln und in ein vorhandenes System integrieren. Ein in CHR geschriebener Constraintlöser (CHR-Programm) repräsentiert dann gleichzeitig die Spezifikation (die Theorie) und ihre Implementierung. CHR-Programme bestehen aus Regeln, die beschreiben, wie Constraints sich zu neuen Constraints vereinfachen (Simplifikationsregeln) und wie Constraints neue Constraints propagieren (Propagationsregeln). Fortgesetzte Vereinfachung und Propagierung von Constraints führt schließlich zur Lösung der Constraints.

Da CHR-Programme auf einem deklarativen Regelformalismus basieren, sind sie sehr leicht erweiterbar und modifizierbar. Die Constraint-Sprache CHR erleichtert dann die Constraint-Programmierung erheblich. Sie ermöglicht die einfache Implementierung und Modifizierung von Constraintlösern. Aus dem Grund ist es besonders wichtig, selbst geschriebene CHR-Programme zu analysieren und zu verifizieren. Im Gegensatz zu klassischen Constraintlösern (Blackbox-Ansatz), wo die Analyse und die Verifikation meistens unmöglich ist, sollen dem CHR-Benutzer zusätzlich Werkzeuge zu Verfügung gestellt werden, mit denen er testen kann, ob bestimmte Eigenschaften von einem selbst geschriebenen Constraintlöser erfüllt sind. Die Deklarativität der Sprache CHR erleichtert diese Analyse.

Eine wichtige Eigenschaft eines Constraintlösers ist die Konfluenz. CHR ist eine „committed choice“-Sprache, d.h. wenn es mehrere passende Regeln gibt, nimmt man eine, die alternativen Regeln werden nicht mehr berücksichtigt. Dies zieht jedoch auch Probleme nach sich: Manche CHR-Programme können bei identischen Eingaben je nach Regelauswahl unterschiedliche Antworten liefern. Wenn das der Fall ist, handelt es sich im allgemeinen entweder um einen Programmierfehler oder um einen unvollständigen Constraintlöser. Wenn der Löser also bei einer Anfrage in zwei Fällen zwei unterschiedliche Antworten ausgibt, so kann man davon ausgehen, daß eine Antwort

„einfacher“ ist als die andere. In Extremfällen kann der Constraintlöser mit einer Berechnungssequenz eine Inkonsistenz von Constraints erkennen, während eine andere Berechnungssequenz eine Konjunktion von Constraints als Antwort liefert. Auch wenn beide Antworten korrekt sind (in dem Sinne, daß sie beide logisch äquivalent zur Anfrage sind), ist dies nicht erwünscht, da eine Inkonsistenz natürlich erkannt werden soll. Man wird also von einem Constraintlöser verlangen, daß er auf eine Anfrage syntaktisch gleiche Antworten liefert, das heißt, daß er konfluent ist. Konfluenz garantiert also, daß dasselbe Ergebnis berechnet wird, unabhängig davon, in welcher Reihenfolge die Constraints eintreffen und in welcher Reihenfolge die Regeln darauf angewandt werden, um sie abzuarbeiten.

Themen dieser Arbeit

In dieser Arbeit wird diese wichtige Eigenschaft der Konfluenz genauer untersucht. Das Hauptergebnis ist eine entscheidbare Charakterisierung der konfluenten terminierenden CHR-Programme mit Hilfe der kritischen Paare des Programmes. Dieser Begriff ergab sich durch Übertragung des entsprechenden Begriffes aus dem Bereich der Termersetzungssysteme, genauer gesagt der Bedingten Termersetzungssysteme (Conditional Term Rewrite Systems) [DOS88] auf CHR. Ein kritisches Paar bezeichnet die zwei „allgemeinsten“ Folgezustände, die durch die Anwendung zweier CHR-Regeln bzw. zweier Varianten einer Regel auf einen Ausgangszustand entstehen können. Das Anwenden zweier Regeln auf denselben Ausgangszustand kommt formal durch die Überlappung ihrer Kopfatome durch Unifikation zum Ausdruck.

Außerdem untersucht die Arbeit den Nutzen des Konfluenzbegriffs für zwei wichtige Aufgaben: die operationale und die semantische Analyse von CHR-Programmen.

Bei der operationalen Analyse von CHR-Programmen stellt die Konfluenz sicher, daß bei jeder möglichen Auswahlstrategie dieselbe Antwort berechnet wird, auch wenn sich die Berechnungen selber voneinander unterscheiden. Insbesondere ist die Antwort auch unabhängig von der Reihenfolge, in der die Constraints dem Löser übergeben werden. Dies ist gerade bei nebenläufigen Anwendungen von Interesse. Außerdem erlaubt das in dieser Arbeit entwickelte Kriterium der kritischen Paare nicht nur die Konfluenz von Programmen zu überprüfen, sondern auch explizit die Regeln zu identifizieren, die verantwortlich sind, wenn ein CHR-Programm nicht konfluent ist.

Bei der semantischen Analyse garantiert die Konfluenz, daß die deklarative Semantik eines CHR-Programmes gemeinsam mit einer (gewisse vordefinierte Constraints beschreibenden) Theorie eine konsistente Theorie ist. Aus den Korrektheits- und Vollständigkeitssätzen folgt auch, daß nur in einer konsistenten Theorie „sinnvolle“ Ableitungen garantiert sind. Außerdem bietet die Konfluenz eine einfache Möglichkeit, Programme und sogar einzelne Constraints aus Programmen hinsichtlich Kompatibilität zu vergleichen. Dies findet eine praktische Anwendung bei der Kombination von Constraintlösern, die sich in einigen Constraintdefinitionen überlappen. Hier ist es ja für den Benutzer von entscheidender Bedeutung, daß diese Constraints dieselben oder wenigstens nicht widersprüchliche semantische Eigenschaften haben. Es ist ja denkbar, daß ein Constraint von dem einen Löser nur partiell definiert wird, daß also die semantischen Eigenschaften der Constraintdefinition des einen Lösers eine Teilmenge der Eigenschaften der anderen Constraintdefinition sind.

Ein weiteres Ergebnis der Arbeit ist ein Verfahren, das in vielen Fällen ein terminierendes nicht-konfluentes CHR-Programm zu einem terminierenden konfluenten transformiert, das äquivalent zum ursprünglichen ist, wenn das CHR-Programm bereichsbeschränkt ist. Dieses Verfahren ist eine Variante der Vervollständigung von Knuth und Bendix. Das Vervollständigungsverfahren kann aber auch scheitern, wenn man es mit Constraintsymbolen zu tun hat, die permutativ sind (Permutativität ist eine Verallgemeinerung der Symmetrie auf mehrstelligen Constraintsymbolen). Die Arbeit präsentiert einen Ansatz zur Behebung dieser Schwierigkeit, der auf einer Erweiterung der operationalen Semantik und einer Verallgemeinerung der Konzepte auf Konzepte modulo Permutativität beruht. Dieser Ansatz ist ebenfalls von Arbeiten im Bereich der Termersetzung inspiriert.

Diese Arbeit gliedert sich wie folgt: Kapitel 2 gibt einen Überblick über die wichtigsten Ideen der Constraint-Logikprogrammierung. Kapitel 3 stellt die Syntax und die deklarative Semantik von CHR-Programmen vor und führt eine neue operationale Semantik ein. Für diese werden die erforderlichen Zusammenhänge zur deklarativen Semantik nachgewiesen.

Kapitel 4 ist dem Hauptthema Konfluenz gewidmet. Dieses Kapitel ist in zwei Abschnitte unterteilt. Zuerst wird die Konfluenz von Programmen bestehend aus nur Simplifikationsregeln untersucht. Wenn die Propagationsregel in die Betrachtungen miteinbezogen wird, verkomplizieren sich die Verhältnisse beträchtlich. Nach einer genauen Analyse dieser Probleme folgt eine angepaßte operationale Semantik, die eine sehr gute Annäherung an die tatsächliche Implementierung ist. Diese zweite operationale Semantik führt schließlich zu einer entscheidbaren Charakterisierung der konfluenten terminierenden CHR-Programme.

Kapiel 5 behandelt dann die Vervollständigung von CHR-Programmen, Kapitel 6 die Anwendung von der Konfluenz und Vervollständigung und Kapitel 7 die Verallgemeinerung für die Permutativität.

Die Kapitel 8 und 9 stellen Bezüge zu anderen Gebieten her und ordnen diese Arbeit in breitere Zusammenhänge ein.

Diese Arbeit setzt Grundkenntnisse der Logik und der Logikprogrammierung voraus. Die Definitionen grundlegender Begriffe, so wie sie in dieser Arbeit verwendet werden, werden in einem Anhang aufgelistet.

Kapitel 2

Constraint-Logikprogrammierung

Dieses Kapitel bietet eine formale Einführung in die Constraint-Logikprogrammierung (CLP), die bekannt gewordene Ausweitung des Paradigmas der Logikprogrammierung. Eine ausführliche Einführung in die Constraint-Programmierung findet sich in [FA97b], an dem wir uns in diesem Kapitel orientieren. CLP-Sprachen vereinigen die Vorteile von LP-Sprachen mit denen von Constraintlösen. Die Idee ist dann, gewisse syntaktisch ausgezeichnete Relationen, die Constraints genannt werden, nicht mittels Resolution, sondern durch spezielle Algorithmen zu lösen. Dadurch braucht man sich nicht damit zu behelfen, nicht-logische Konstrukte in LP-Sprachen aufzunehmen, um z.B. zwei natürliche Zahlen zu addieren. Eine saubere, deklarative Lösung für die Behandlung von arithmetischen Ausdrücken in LP-Sprachen ist die Einführung von Gleichungen zwischen linearen arithmetischen Ausdrücken, die von einer speziellen Programmbibliothek effizient behandelt werden. Wir wollen etwa `X-Y=3, X+Y=5` schreiben und damit die Lösung `X=4, Y=1` bekommen. Dazu erweitert man die Unifikation so, daß sie, wenn immer ein arithmetisches Funktionsymbol in einer Gleichung vorkommt, diese Teilgleichung an das Spezialprogramm weitergibt. Damit lassen wir nicht nur uninterpetierte Funktionsymbole zu, sondern auch interpretierte, die von einem speziellen Programm behandelt werden. Analog will man auch, daß Konjunktionen von speziellen Relationen durch den arithmetischen Gleichungslöser und nicht durch Resolution behandelt werden, zum Beispiel erwartet man, daß der Gleichungslöser die Bindungen `X=0,Y=1` (die einzige Lösung über den natürlichen Zahlen) für das Ziel `X<Y, X+Y=1` liefert. Im allgemeinen muß man auch hier damit rechnen, „unvollständige“ (intensionale) Antworten zu bekommen, z.B. ein Ziel wie `X+Y>1, X+Y<3` wird zu `X+Y=2` vereinfacht.

Constraint-Logikprogrammierung wurde mit Prolog II [CKvC83] aus der Taufe gehoben. Dabei sind Constraints prädikatenlogische Formeln erster Stufe, die zur abstrakten Beschreibung von logischen Datenstrukturen dienen, typischerweise von arithmetischen Werten oder von verschiedenartigen Bäume. Aus diesen Entwicklungen heraus entstanden dann in der zweiten Hälfte der achtziger Jahre die ersten CLP-Sprachen, nämlich CLP(R) [HJM+87], CHIP [DVS+88, DHS+90] und Prolog III [Col90].

Formal betrachtet besteht ein Constraintproblem aus einer Menge von Variablen und Eigenschaften dieser Variablen sowie Beziehungen zwischen diesen Variablen (Constraints), die erfüllt werden müssen. Die Idee ist, daß man mit Hilfe von Constraints partielle Information über die möglichen Belegungen der Variablen ansammelt und aus dieser partiellen Information Schlußfolgerungen ziehen kann. Die Art der Relationen und der Wertebereiche bestimmt das Constraintsystem, das zur Lösung des Problems benutzt werden kann. Ein Beispiel sind lineare Gleichungssysteme über reellen Zahlen, Boolesche Algebra, Rechnen mit endlichen Mengen (Aufzählungen) oder Zeichenketten. Ein Constraintsystem legt dann die Syntax und die Semantik einer Menge von Constraints fest. Die zwei wichtigsten Beschreibungsformalismen zur Definition von Constraintssystemen sind das CLP-Schema von Jaffar und Lassez [JL87] und das allgemeinere Schema von Höhfeld und Smolka [HS88]. Der Hauptunterschied zwischen diesen beiden Schemata liegt in der Vielfältigkeit der Constraints. Während das Höhefeld-Smolka Schema Constraints über mehreren Wertebereichen (Constraint-Bereiche) zuläßt, werden beim CLP-Schema nur Constraints über einem einzigen Wertebereich erlaubt.

2.1 Constraintsystem

Im folgenden wird das allgemeinere Höhefeld-Smolka Schema logisch beschrieben und folgen dabei [Abd96, FA97b], da das CLP-Schema ein Spezialfall davon ist. Hier wird in einem Constraintsystem angegeben, um welche Constraintsymbole es sich handelt, wie die Constraintsymbole definiert sind und welche Constraints erlaubt sind.

Definition 2.1 Ein *Constraintsystem* ist ein Tripel $(\Sigma, CT, \mathcal{C})$. Dabei sind

- Σ eine Signatur, die mindestens die nullstelligen Constraintsymbole *true* und *false* enthält,
- die *Constrainttheorie* CT eine nichtleere, konsistente Theorie über Σ und
- die Menge der Constraints $\mathcal{C}$ über Σ, so daß
  - *true* $\in \mathcal{C}$ und *false* $\in \mathcal{C}$;
  - $\mathcal{C}$ ist abgeschlossen unter Konjunktion, d.h. wenn $C_1, C_2 \in \mathcal{C}$, dann $C_1 \wedge C_2 \in \mathcal{C}$;
  - $\mathcal{C}$ ist abgeschlossen unter Variablenumbenennung (Variantenbildung), d.h. wenn $C_1 \in \mathcal{C}$ und C_2 eine Variante von C_1 ist, dann ist $C_2 \in \mathcal{C}$.

Die Constrainttheorie schränkt die Menge der möglichen Interpretationen für Constraintsymbole ein, indem ihre Eigenschaften durch Axiome beschrieben werden. Außerdem ist die Menge der Constraints $\mathcal{C}$ gegenüber der Menger aller Formeln über der Constraintsignatur stark eingeschränkt, um die gewünschten Berechnungsdienste mit angemessenem Aufwand erzielen zu können. $\mathcal{C}$ wird auch die Menge der *erlaubten Constraints* genannt.

Logisch gesehen haben $\top$ und *true* die Bedeutung „wahr" und $\perp$ und *false* die Bedeutung „falsch". Aber damit Constraints von prädikatenlogischen Formeln unterschieden werden können, wird diese zusätzliche Notation eingeführt.

Beispiel 2.2 Das *Herbrand*-Constraintsystem H [FA97b] stellt Constraints für die Gleichheit von Termen zur Verfügung. Es hängt eng mit der Unifikation von Termen zusammen. Die Signatur von H enthält u.a. das zweistellige Constraintsymbol $\doteq$ und die Constraintsymbole *true* und *false*. Als Theorie von H wählen wir die Clarksche Gleichheitstheorie CET

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

Die erlaubten Constraints von H haben die Form

$$C \quad ::= \mathit{true} \mid \mathit{false} \mid s \doteq t \mid C \wedge C,$$

wobei s und t beliebige Terme über der Signatur von H sind.

Im folgenden werden wichtige Eigenschaften von Constraintsystemen vorgestellt. Man bevorzugt natürlich eine vollständige Constrainttheorie.

Definition 2.3 Gegeben sei ein Constraintsystem $(\Sigma, CT, \mathcal{C})$. Eine Constrainttheorie CT heißt *vollständig* genau dann, wenn $CT \models \forall C$ oder $CT \models \forall \neg C$ für jedes erlaubte Constraint $C \in \mathcal{C}$ gilt.

Dies ist aber eine starke Eigenschaft, die von wenigen Constrainttheorien erfüllt wird. Meistens reicht eine schwächere Vollständigkeitsbedingung aus, da man nur mit existentiell quantifizierten Variablen zu tun hat.

Definition 2.4 Gegeben sei ein Constraintsystem $(\Sigma, CT, \mathcal{C})$. Eine Constrainttheorie CT heißt *erfüllbarkeitsvollständig* genau dann, wenn entweder $CT \models \exists C$ oder $CT \models \neg\exists C$ für jedes erlaubte Constraint $C \in \mathcal{C}$ gilt.

Die folgende Eigenschaft, *Grundvollständigkeit*, ist ein Spezialfall der Erfüllbarkeitsvollständigkeit und wird von fast allen bekannten Constraintsystemen erfüllt. Diese Eigenschaft wird für einige Ergebnisse in Abschnitt 6.1.1 (Konfluenz und Konsistenz von CHR-Programmen) gebraucht.

Definition 2.5 Gegeben sei ein Constraintsystem $(\Sigma, CT, \mathcal{C})$. Eine Constrainttheorie CT heißt *grundvollständig* genau dann, wenn entweder $CT \models C$ oder $CT \models \neg C$ für jedes erlaubte Grundconstraint $C \in \mathcal{C}$ gilt.

Eine weitere wichtige Eigenschaft von Constraintsystemen ist die *Unabhängigkeit*. In einem unabhängigen Constraintsystem ist eine Konjunktion von positiven Constraints und negierten Constraints genau dann erfüllbar, wenn jedes negierte Constraint für sich mit jedem positiven Constraint erfüllbar ist.

Definition 2.6 Ein Constraintsystem $(\Sigma, CT, \mathcal{C})$ heißt *unabhängig* genau dann, wenn für beliebige erlaubte Constraints $C, C_1, \ldots, C_n \in \mathcal{C}$ gilt:

$$CT \models \exists(C \wedge \neg C_1 \wedge \ldots \wedge \neg C_n) \text{ gdw. } CT \models \exists(C \wedge \neg C_i) \text{ für } i = 1, \ldots, n.$$

Beispiel 2.7 Das Constraintsystem *INT* [FA97b], das Constraints für ganze Zahlen zur Verfügung stellt, ist nicht unabhängig. Die Signatur von *INT* enthält die Konstanten $0, -1, 1, -2, 2, \ldots$, die zweistelligen Funktionssymbole $+$ und $*$ sowie das zweistellige Constraintsymbol $<$. Als Theorie CT von *INT* kann man die Sätze wählen, die für ganze Zahlen gültig sind. Die erlaubten Constraints von *INT* haben die Form:

$C ::= s \doteq t \;\mid\; \mathit{true} \;\mid\; \mathit{false} \;\mid\; s < t \;\mid\; C \wedge C$

wobei s und t beliebige Terme über der Signatur von *INT* sind.
Das Constraintsystem *INT* ist nicht unabhängig, da z.B. für $C = \mathit{true}, C_1 = X < Y, C_2 = X \doteq Y$ und $C_3 = Y < X$ folgendes gilt:

$CT \models \exists X, Y\ \neg(X < Y)$,
$CT \models \exists X, Y\ \neg(X \doteq Y)$ und
$CT \models \exists X, Y\ \neg(Y < X)$ aber
$CT \not\models \exists X, Y\ (\neg(X < Y) \wedge \neg(X \doteq Y) \wedge \neg(Y < X))$.

2.2 Constraintlöser

Ein Constraintsystem legt unter anderem fest, wie Constraints definiert sind, aber sagt nichts über ihre Behandlung. Der *Constraintlöser* implementiert einen speziellen Algorithmus zum Lösen der erlaubten Constraints eines Constraintsystems im Einklang mit der Constrainttheorie. Die Constraints werden im Zuge der Abarbeitung eines Programmes inkrementell dem sogenannten Constraintspeicher hinzugefügt. Der Constraintlöser vereinfacht dann den Constraintspeicher bei Hinzukommen neuer Constraints:

Vereinfachung: Wenn $CT \models \forall(C \leftrightarrow C')$ und C' „einfacher" ist als C, dann ersetze C durch C'. In der Regel wird C' eine Normalform von C sein.

Außerdem hat ein Constraintlöser zusätzliche Aufgaben, die als Varianten bzw. Erweiterungen der Vereinfachung implementiert werden können:

Konsistenztest : Ist C in der Constrainttheorie erfüllbar, $CT \models \exists C$?

Inkrementalität: Ein Constraintlöser kann beim Hinzufügen eines neuen Constraints C in eine Menge M von Constraints die Menge $M \cup \{C\}$ vereinfachen, ohne die Vereinfachung der bisherigen Constraints M zu wiederholen.

Determination: Man will prüfen, ob die im Speicher vorliegenden Constraints schon ausreichen, um den Wert einer Variable eindeutig zu bestimmen.

2.3 CLP-Kalkül

In diesem Abschnitt geben wir einen Kalkül an, der mit Constraints rechnen kann. Der CLP-Kalkül ist eine Verallgemeinerung des LP-Kalküls (siehe Abschnitt A.2). Die rein syntaktische Gleichheit (Unifikation) wird zur Behandlung von Constraints verallgemeinert. Statt Substitutionen gibt es nun Gleichungen (Gleichheitsconstraints). Im folgenden werden Substitutionen von der Form $\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$ mit dem Gleichungssystem $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ identifiziert.

Ein Constraintlöser zur Behandlung von Gleichheitsconstraints soll dann die Theorie CET implementieren, d.h. der Constraintlöser soll folgendes leisten: Wenn $s \doteq t$ in CET erfüllbar ist (d.h. s und t sind unifizierbar), dann wird $s \doteq t$ in die (Normal-)Form $X_1 \doteq t_1 \wedge \ldots \wedge X_n \doteq t_n$ gebracht, wobei die Variablen $X_1, \ldots, X_n$ paarweise verschieden sind und nicht in den Termen $t_1, \ldots, t_n$ vorkommen, ansonsten wird $s \doteq t$ zu *false* reduziert.

Die CLP-Syntax (Abb. 2.1) unterscheidet sich von der LP-Syntax nur dadurch, daß Constraints im Rumpf einer Klausel auftreten können. In der Signatur des CLP-Kalküls ist also die Menge $\mathcal{P}$ der Prädikatensymbole in zwei disjunkte Mengen unterteilt: $\mathcal{P}_c$ enthält die Constraintsymbole, die durch eine Constrainttheorie CT beschrieben sind und durch einen Constraintlöser behandelt werden, und $\mathcal{P}_p$ die Prädikatensymbole, die durch CLP-Programme definiert werden.

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

Abbildung 2.1: CLP-Syntax

Definition 2.8 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 $\mathcal{P}_c$ und $t_1, \ldots, t_n$ Terme sind. Ein *Constraint* ist ein Constraintatom oder eine Konjunktion von Constraints.
Ein *Zielatom* ist ein Atom oder ein Constraintatom. Ein *Ziel* ist entweder $\top$ oder $\bot$ oder ein Zielatom oder eine Konjunktion von Zielen. Eine *CLP-Klausel* hat die Form $A \leftarrow G$, wobei A ein Atom ist und G ein Ziel.
Ein *CLP-Programm* ist eine endliche Menge von CLP-Klauseln.

Definition 2.9 Ein *Zustand* ist ein Paar $\langle G, C\rangle$, wobei G ein Ziel ist und C eine Konjunktion von Constraints. G wird auch Zielspeicher genannt und C Constraintspeicher. $\top$ stellt den leeren Zielspeicher dar.

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

Definition 2.10 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 (gescheiterter) Endzustand, wenn er die Form $\langle G, \mathit{false}\rangle$ hat.

Definition 2.11 Eine Ableitung ist *erfolgreich*, wenn sie endlich ist und ihr Endzustand erfolgreich ist. Eine Ableitung ist *erfolglos* (*gescheitert*), wenn sie endlich ist und ihr Endzustand erfolglos (gescheitert) ist.
Ein Ziel G ist *erfolgreich*, wenn es eine erfolgreiche Ableitung beginnend mit $\langle G, \mathit{true}\rangle$ hat. Ein Ziel G ist *erfolglos (endlich gescheitert)*, wenn es nur erfolglose Ableitungen beginnend mit $\langle G, \mathit{true}\rangle$ hat.

Die Definition des Begriffs Ableitung findet sich im Anhang (Definition A.19).

Definition 2.12 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 (lokalen) Variablen sind, die in H oder C aber nicht in G vorkommen. Die in G vorkommenden Variablen bleiben frei.

Im folgenden steht $p(t_1, \ldots, t_n) \doteq q(s_1, \ldots, s_n)$ als Abkürzung für $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$, falls p und q die gleichen Prädikatensymbole sind und für *false* sonst.

Unfold

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

Clash

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

Simplify

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

Abbildung 2.2: CLP-Reduktionsregeln

Die Reduktionsregeln **Unfold** und **Clash** (Abb. 2.2) bearbeiten die Atome und die Reduktionsregel **Simplify** bearbeitet die Constraints. Diese Reduktionsregel wird durch den Constraintlöser durchgeführt. Die Kongruenz zwischen Konjunktionen von Zielen ist dieselbe wie im LP-Kalkül (Abschnitt A.2). Nachdem ein Ziel aus dem Zielspeicher ausgewählt wird, wird der Folgezustand aus dem aktuellen Zustand wie folgt bestimmt:

- Ist das ausgewählte Ziel ein Atom und wird eine frische Variante einer Klausel gefunden, derart, daß die Gleichheitsconstraints, die aus dem Atom und dem Kopf der CLP-Klausel entstehen, mit dem Constraintspeicher konsistent sind, dann wird das ausgewählte Atom mit Hilfe der Reduktionsregel **Unfold** durch den Rumpf der CLP-Klausel ersetzt. Anderfalls scheitert die Ableitung für das Atom durch Anwendung der Reduktionsregel **Clash**. Mit einer frischen Variante einer Klausel (siehe Definition A.9) ist hier eine Variante einer Klausel gemeint, die keine gemeinsamen Variablen mit dem CLP-Programm und den Komponenten des betrachteten Zustandes hat.

- Ist das ausgewählte Ziel ein Constraint, dann wird es aus dem Zielspeicher mit Hilfe der Reduktionsregel **Simplify** entfernt, zusammen mit dem Constraintspeicher durch den Constraintlöser vereinfacht. Die vereinfachten Constraints ergeben den neuen Constraintspeicher.

Die Auswahl der Ziele innerhalb eines Zustandes wird wieder durch eine Selektionsfunktion bestimmt. Die meisten CLP-Sprachen benutzen wie Prolog die Tiefensuche mit chronologischem Backtracking.

Im CLP-Kalkül werden Antworten als Constraints dargestellt. Constraints als Antworten sind sinnvoll, da sie eine ganze Menge von Antworten (im Sinne von LP) zu einer Antwort (im Sinne von CLP) zusammenfassen (intensionale Antworten).

Definition 2.13 Ein *Antwortconstraint* eines Zieles G ist die logische Beschreibung eines Endzustandes einer Ableitung, die mit $\langle G, true\rangle$ beginnt.

Eine Beschränkung der berechneten Antwort auf die Variablen des Zieles G (Projektion) ist wünschenswert, aber abhängig vom Constraintsystem und Constraintlöser nicht immer (effizient) möglich. Im Rest der Arbeit wird in den Beispielen anstatt das Gleichheitssymbol $\doteq$ das Symbol = verwendet.

Beispiel 2.14 Sei P das CLP-Programm aus Beispiel 1.2:

```
max(X,Y,Z) ← X ≤ Y ∧ Y=Z.
max(X,Y,Z) ← X > Y ∧ X=Z.
```

Hierbei sind $\leq$, $>$ und $=$ Constraintsymbole. Ein Anfangszustand von der Form $\langle \texttt{max(3,5,C)}, true\rangle$ resultiert in folgenden Zustandsübergängen:

$$\begin{array}{ll} & \langle \texttt{max}(3,5,\texttt{C}), true\rangle \\ \mapsto & \langle \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} = \texttt{Z}, 3 = \texttt{X} \wedge 5 = \texttt{Y} \wedge \texttt{C} = \texttt{Z}\rangle \\ \mapsto & \langle \top, \texttt{C} = 5\rangle \end{array}$$

Hierbei erfolgt zuerst eine Reduktion **Unfold**, bei der die erste CLP-Klausel ausgewählt wird. Dann erfolgt eine Constraintvereinfachung durch Anwendung der Reduktion **Simplify**, die der Constraintlöser intern durchführt. Die Ableitung ist erfolgreich und das Antwortconstraint ist `C=5` (Projektion des Antwortconstraints).

Die Auswahl der zweiten CLP-Klausel im ersten Reduktionschritt führt zu einem inkonsistenten Constraintspeicher, nämlich $3 > 5 \wedge 3 = \texttt{C}$. Damit scheitert diese Ableitung.

Ein Anfangszustand von der Form $\langle \texttt{max}(\texttt{A},2,5), true\rangle$ resultiert in folgenden Zustandsübergängen:

$$\begin{array}{ll} & \langle \texttt{max}(\texttt{A},2,5), true\rangle \\ \mapsto & \langle \texttt{X} > \texttt{Y} \wedge \texttt{X} = \texttt{Z}, \texttt{A} = \texttt{X} \wedge 2 = \texttt{Y} \wedge 5 = \texttt{Z}\rangle \\ \mapsto & \langle \top, \texttt{A} = 5\rangle \end{array}$$

In Prolog würde die Abarbeitung zu einer Fehlermeldung führen, da noch kein Wert für `A` bekannt ist und der Vergleich nur zwischen bekannten Werten möglich ist.

Wenn der Anfangszustand von der Form $\langle \texttt{max}(\texttt{A},\texttt{A},\texttt{B}), true\rangle$ ist, dann bekommt man folgende intensionale Antwort:

$$\begin{array}{ll} & \langle \texttt{max}(\texttt{A},\texttt{A},\texttt{B}), true\rangle \\ \mapsto & \langle \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} = \texttt{Z}, \texttt{A} = \texttt{X} \wedge \texttt{A} = \texttt{Y} \wedge \texttt{B} = \texttt{Z}\rangle \\ \mapsto & \langle \top, \texttt{A} = \texttt{B}\rangle \end{array}$$

Ebenfalls würde die Abarbeitung des Zieles `max(A,A,B)` in Prolog zu einer Fehlermeldung führen.

2.4 Korrektheit und Vollständigkeit des CLP-Kalküls

Eine CLP-Klausel $A \leftarrow G$ kann als der universelle Abschluß der Implikation $G \rightarrow A$ verstanden werden. Die Bedeutung der Constraints wird von der Constrainttheorie CT festgelegt. Die deklarative Semantik eines CLP-Programms P wird dann durch die Konjunktion der universellen Abschlüße der Klauseln von P (bezeichnet als $P^{\rightarrow}$) und die Constrainttheorie CT festgelegt. Dies legt die sogenannte deklarative Semantik von CLP-Programmen fest.

Im folgenden wird ein Zusammenhang zwischen der durch den CLP-Kalkül definierten operationalen Semantik und der deklarativen Semantik hergestellt. Der Korrektheitssatz des CLP-Kalküls wurde erstmals in [JL86] vorgestellt. Der Korrektheitssatz des LP-Kalküls und dessen Beweis können auf einfache Art und Weise auf das CLP-Schema übertragen werden: Das Herbrand-Universum wird durch einen beliebigen Wertebereich ersetzt und die Clarksche Gleichheitstheorie CET durch eine entsprechende Constrainttheorie. Diese Übertragung wurde in [Mah93] detailliert diskutiert.

Satz 2.15 Seien P ein CLP-Programm und G ein Ziel. Wenn G eine erfolgreiche Ableitung mit Antwortconstraint C hat, dann gilt

$$P^{\rightarrow} \cup CT \models \forall\, (C \rightarrow G)$$

Der Vollständigkeitssatz [Mah87] des CLP-Kalküls unterscheidet sich stark von dem des LP-Kalküls.

Satz 2.16 Seien P ein CLP-Programm und G ein Ziel. Wenn $P^{\rightarrow} \cup CT \models \forall\, (C \rightarrow G)$ und C in CT erfüllbar ist (d.h. $CT \models \exists C$), dann hat G erfolgreiche Ableitungen mit Antwortconstraints $C_1, \ldots, C_n$, sodaß

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

Dieser Satz zeigt, daß man im Gegensatz zum LP-Kalkül mehrere Antworten berechnen muß, um die Vollständigkeit zu erhalten, wie das folgende Beispiel illustriert:

Beispiel 2.17 Sei P das folgende Programm und CT eine Constrainttheorie, die u.a. die Constraintsymbole $\leq$ und $\geq$ als totale Ordnungen definiert.

```
p(X,Y) ← X ≤ Y.
p(X,Y) ← X ≥ 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 $P^{\rightarrow} \cup CT \models \forall(\mathit{true} \rightarrow \mathtt{p(X,Y)})$. Daher muß nach obigem Satz auch gelten $CT \models \forall(\mathit{true} \rightarrow \mathtt{X} \leq \mathtt{Y} \vee \mathtt{X} \geq \mathtt{Y})$. Dies ist der Fall. Jedes Antwortconstraint für sich allein reicht aber nicht aus, um dies zu zeigen: $CT \not\models \forall(\mathit{true} \rightarrow \mathtt{X} \leq \mathtt{Y})$ und $CT \not\models \forall(\mathit{true} \rightarrow \mathtt{X} \geq \mathtt{Y})$.

2.5 Nebenläufige Constraint-Logikprogrammierung

Die Untersuchung der nebenläufigen Logik-Programmierung begann mit [CG81] und wurde unter anderem von Ehud Shapiro propagiert [Sha89]. Die Nebenläufige Constraint-Logikprogrammierung (NCLP) kombiniert Constraint-Logikprogrammierung und nebenläufige Logik-Programmierung. Vijay Saraswat schlug in [SRP91, Sar93] ein erstes einheitliches Modell vor.

Der Schwerpunkt von NCLP liegt auf nebenläufigen und inkrementellen Aspekten von Berechnung mit partieller, logischer Information. Ihre Modelle basieren auf der Metapher eines Berechnungsraumes, welcher aus Aktoren und Constraintspeicher besteht. Die Aktoren können auf die Information des Constraintspeichers zugreifen und, falls genügend Information vorhanden ist, reduzieren. Ein Aktor verschwindet bei Reduktion, fügt aber möglicherweise neue Information und Aktoren hinzu.

In dieser Arbeit werden wir uns auf die NCLP-Sprachen beschränken, die in [JM94] vorgestellt wurden. Diese Sprachen sind „committed choice“, d.h., falls es mehrere passende Klauseln gibt, nimmt man irgendeine, die alternativen Klauseln werden nicht mehr berücksichtigt. Man spricht in diesem Zusammenhang auch von „don't care“-Nichtdeterminismus.

Die Syntax von NCLP-Sprachen unterscheidet sich von der Syntax der CLP-Sprachen darin, daß Klauseln syntaktisch um Vorbedingungen (Wächter) erweitert werden (Abb. 2.3).

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

Abbildung 2.3: NCLP-Syntax

Definition 2.18 Eine NCLP-Klausel ist von der Form

$$A \leftarrow C \mid G,$$

wobei der Kopf A ein Atom ist, C ein Constraint, genannt *Wächter*, und der Rumpf G ein Ziel. Ein *NCLP-Programm* ist eine endliche Menge von NCLP-Klauseln.

Unfold

$$\frac{\begin{array}{c}(B \leftarrow C \mid H) \text{ eine frische Variante aus } P\\ CT \models (D \rightarrow \exists \bar{x}(B \doteq A \wedge C))\end{array}}{\langle A \wedge G, D\rangle \mapsto \langle H \wedge G, B \doteq A \wedge C \wedge D\rangle}$$

Simplify

$$\frac{CT \models C \wedge C_1 \leftrightarrow C_2}{\langle C \wedge G, C_1\rangle \mapsto \langle G, C_2\rangle}$$

Abbildung 2.4: NCLP-Reduktionsregeln

Der NCLP-Kalkül unterscheidet sich vom CLP-Kalkül in folgenden zwei Punkte:

- Die Bedingung für die Reduktionsregel **Unfold**: Ein Ziel A kann erst mit einer frischen Variante einer Regel $B \leftarrow C \mid G$ reduziert werden, wenn aus dem aktuellen Constraintsspeicher die logische Formel $\exists \bar{x}\ (B \doteq A \wedge C)$ folgt, wobei $\bar{x}$ die in B vorkommenden Variablen sind. Operational entspricht der Folgerungstest für die Gleichheit $\exists \bar{x}\ (B \doteq A)$ einer Überprüfung, ob das ausgewählte Ziel eine Instanz des Klauselkopfes ist, und einer nachfolgenden Unifikation. Man spricht auch von einseitiger Unifikation oder Matching. Die Idee, die Reduktionsbedingung logisch zu beschreiben stammt von Michael Maher [Mah87]:
- Keine Reduktionsregel fürs Scheitern: Die Abarbeitung der Atome, für die keine passende Klausel gefunden wurde, wird verzögert in der Hoffnung, daß **Unfold** später möglich ist. Daraus ergibt sich auch eine neue Art von Endzustand (*verklemmter* Zustand), die daher kommt, daß kein Atom mehr entfaltet werden kann.

Die deklarative Semantik von NCLP-Programmen ist analog zur deklarativen Semantik von CLP-Programmen. Das Symbol „|“ wird dabei als Konjunktion interpretiert. Die logische Leseweise einer NCLP-Klausel $A \leftarrow C \mid G$ ist daher der universelle Abschluß der Formel $C \wedge G \rightarrow A$.

Verglichen mit CLP-Sprachen werden NCLP-Sprachen für andere Zwecke verwendet: NCLP-Sprachen erlauben die Beschreibung von Programmen, die nebeneinander ausgeführt werden und miteinander interagieren können, sogenannten Prozessen. Prozesse können entweder lokale Aktionen ausführen oder miteinander kommunizieren und sich synchronisieren, indem sie Nachrichten senden und empfangen. Der Preis, den man dafür bezahlt, ist die weitere Entfernung von der deklarativen Semantik der Programme. Im Prinzip würden die Korrektheits- und Vollständigkeitssätze wie in CLP (Abschnitt 2.4) gelten. Das Problem ist aber, daß im NCLP-Kalkül wegen des „don't care" Nichtdeterminismus immer nur eine Ableitung betrachtet wird und nicht wie im CLP-Kalkül alle Ableitungen. Deswegen ist ein Vollständigkeitssatz von der Art wie Satz 2.16 ungenügend, weil nicht alle Ableitungen eines Zieles untersucht werden sollen. Etliche Arbeiten untersuchten deswegen Unterklassen von Programmen, bei denen man einen zufriendenstellenden Vollständigkeitssatz angeben kann. Eine Teilklasse davon ist die Klasse der *deterministischen CCLP-Programme*: In *deterministischen* Programmen schließen sich die Wächter der Klauseln zur Definition eines Prädikates gegenseitig aus. Dies hat als Konsequenz, daß in jedem beliebigen Zustand maximal eine Klausel angewendet werden kann.

Die Einschränkung auf deterministische NCLP-Programme und auf datenausreichende Ziele (Ziele, die mindestens eine faire Ableitung haben) erlaubt dann einen zufriedenstellenden Vollständigkeitssatz, der in [Mah87] vorgestellt wurde.

Definition 2.19 Eine Ableitung ist *fair*, wenn sie entweder scheitert oder wenn jedes Atom, das in der Ableitung vorkommt, nach endlich vielen Reduktionen ausgewählt wird.
Ein Ziel ist *datenausreichend*, wenn es mindestens eine faire Ableitung hat.

Satz 2.20 Seien P ein deterministisches NCLP-Programm und G ein datenausreichendes Ziel. Wenn gilt $P^{\rightarrow} \cup CT \models \forall\,(C \rightarrow G)$ und C in CT erfüllbar ist (d.h. $CT \models \exists C$), dann hat jede erfolgreiche Ableitung von G ein Antwortconstraint C', so daß

$$CT \models \forall\,(C \rightarrow C').$$

Kapitel 3

Constraint Handling Rules

Wie wir in Kapitel 2 gesehen haben, sind CLP-Sprachen logische Programmiersprachen, die Constraintlöser benutzen. Der Constraintlöser war in diesem Sinne ein Werkzeug, das einem Programmierer der CLP-Sprache zur Verfügung gestellt wird. Die Funktionalität und die Qualität des Constraintlösers spielt dann eine wichtige Rolle bei der constraintbasierten Programmierung. Anfangs war das Constraintlösen in einem vordefinierten Constraintlöser, der in einer low-level Sprache geschrieben ist, fest verdrahtet („Blackbox"-Ansatz). Der Benutzer hat in die Implementierung dieser Löser keinen Einblick, kann sie also nur mit erheblichem Aufwand ergänzen oder modifizieren ([vHSD92]). Die Erfahrungen mit praktischen Anwendungen zeigen aber, daß oftmals kein homogenes Constraintproblem vorliegt, sondern eine subtile Kombination verschiedenster Constraintsysteme wünschenswert ist. Außerdem treten oft neuartige (anwendungsspezifische) Constraints auf, die (wenn überhaupt) nur mit viel Aufwand in existierende Constraints übersetzt werden können.

Verschiedene Ansätze wurden dann vorgeschlagen, um die Flexibilität von Constraintlösern zu erreichen. Man spricht dann von „glass-box"- oder „no-box"-Ansatz:

- Constraint-Kombinatoren in cc(FD) [DVS+88, vH89] erlauben, komplexere Constraints aus einfacheren Constraints zu bilden.
- Dämonen, Vorwärtsregeln und Konditionale in CHIP [vH91] erlauben, Propagation von Constraints in eingeschränkter Weise zu definieren.
- Constraints gebunden an einer Booleschen Variable in BNR-Prolog [OB93] erlauben, beliebige logische Formeln über primitive Constraints auszudrücken.
- Indexicals in clp(FD) [CD93] erlauben, Constraints über endlichen Domänen zu implementieren.
- Meta- und attribuierte Variablen [Neu90, Hui90, Hol92] erlauben, Constraints an Booleschen Variablen anzuhängen, womit logische Formeln über Constraints als Boolesche Constraints ausgedrückt und abgearbeitet werden können.

Die ersten vier Ansätze können nur einen vorgegebenen Constraintlöser über einem bestimmten Bereich, typischerweise über endlichen Domänen, erweitern. Anwendungsorientierte Bereiche können nur mit dem letzten Ansatz implementiert werden. Dies ist aber sehr umständlich, da dieser Ansatz eine Art „Constraint-Assembler"-Programmierung ist.

Um Constraintlöser in einer höheren Sprache schnell und ausreichend effizient zu spezifizieren und implementieren, wurde in der ersten Hälfte der 90er Jahre eine spezielle Sprache, Constraint Handling Rules (CHR) [Frü95, Frü97], von Thom Frühwirth entwickelt. CHR kann in eine beliebige herkömmliche Programmiersprache als Bibliothek eingebettet werden. In der Basissprache werden Programme geschrieben, die dann die in CHR geschriebenen Constraints benutzen können.

Andererseits können CHR-Programme die in der Basissprache vordefinierten Funktionen als Hilfsfunktionen verwenden. Die derzeit wichtigsten Implementierungen von CHR-Bibliotheken gibt es in der Prolog-Implementierung ECL^iPS^e [ACD+94] und Sicstus Prolog [AAB+93], die die weltweit verbreitetsten Prolog-Systeme im Forschungsbereich sind.

CHR-Programme werden in einem Regelformalismus geschrieben. Die Regeln beschreiben, wie Constraints sich zu neuen Constraints vereinfachen (Simplifikationsregeln) und wie Constraints neue Constraints propagieren (Propagationsregeln). Es gibt zusätzlich sogenannte Simpagationsregeln, die eine Mischform der beiden Arten sind. Fortgesetzte Vereinfachung und Propagierung von Constraints führt schließlich zur Lösung der Constraints. Ein CHR-Programm repräsentiert also gleichzeitig die Spezifikation (die Constrainttheorie) und die Implementierung des zugehörigen Constraintlösers. Mit CHR kann man Algorithmen für alle Constraintsysteme spezifieren und gleichzeitig als Constraintlöser implementieren. Damit kann man kompakt und deklarativ die wesentlichen Aspekte der Algorithmen beschreiben. Durch die Verwendung von CHR wird die Abstimmung eines Constraintsystems auf eine bestimmte Anwendung erheblich erleichtert, da es viel einfacher ist, neue Regeln zu formulieren als beispielsweise ein C-Programm, das einen Constraintlöser implementiert, zu modifizieren.

Beispiel 3.1 Wir definieren mit CHR eine Ordnungsrelation `X ≤ Y` als Constraint [Frü95]. Das Gleichheitsconstraintsymbol `=` sei in der Basissprache vorhanden.

```
reflexivitaet @ X ≤ X ⇔ true.
antisymmetrie @ X ≤ Y ∧ Y ≤ X ⇔ X=Y.
transitivitaet @ X ≤ Y ∧ Y ≤ Z ⇒ X ≤ Z.
```

Dabei sind `reflexivitaet @`, `antisymmetrie @` und `transitivitaet @` Namen, die die Regeln identifizieren.

Die Reflexivitätsregel besagt, daß `X ≤ X` wahr ist. Aufgrund der Reflexivitätsregel kann jedes Vorkommen von `A ≤ A` durch `true` ersetzt werden. Diese Regel erkennt die Erfüllbarkeit eines Constraints. Die Antisymmetrieregel drückt aus, daß man die beiden Constraints `X ≤ Y` und `Y ≤ X` zum Gleichheitsconstraint `X=Y` vereinfachen kann. Diese beiden Regeln sind Simplifikationsregeln, die Constraints vereinfachen bzw. lösen. Simplifikationsregeln sind durch $\Leftrightarrow$ gekennzeichnet.

Die Transitivitätsregel besagt, daß die Konjunktion `X ≤ Y ∧ Y ≤ Z` das Constraint `X ≤ Z` impliziert. Diese Regel ist eine Propagationsregel. Propagationsregeln sind durch $\Rightarrow$ gekennzeichnet. Damit werden logische Konsequenzen als zusätzliche Constraints eingefügt. Dies ist durchaus sinnvoll, wie die Behandlung des Constraints `A ≤ B ∧ C ≤ A ∧ B ≤ C` durch die CHR-Regeln zeigt. Die Anwendung der Transitivitätsregel auf die Constraints `C ≤ A ∧ A ≤ B` propagiert das neue Constraint `C ≤ B`. Dieses Constraint zusammen mit `B ≤ C` kann durch die Antisymmetrieregel zu `B=C` vereinfacht werden. Nachdem `B` und `C` nun gleich sind, kann die Antisymmetrieregel auch `A ≤ B ∧ C ≤ A` zu `A=B` vereinfachen. Die (intensionale) Antwort ist also „`A, B` und `C` haben denselben Wert“.

Die Grundidee von Constraint Handling Rules erinnert stark an Termersetzungssysteme, genauer gesagt an die bedingten Termersetzungssysteme [DOS88] (Conditional Term Rewrite Systems, CTRS). Ein ausführlicher Vergleich findet sich in Abschnitt 8.1. Außerdem ähnelt der Regelformalismus von CHR den „Guarded Rules“ [Smo91]. „Guarded Rules“ entsprechen einköpfigen Simplifikationsregeln. Sie werden aber nicht benutzt, um benutzerdefinierte Constraints zu schreiben, sondern nur als Lemmata für Prädikate. [AKP94] beschreibt auch eine Art von „Guarded Rules“, die als Programmiersprache für sich verwendet werden kann. Ein detaillierter Vergleich von CHR mit anderen Sprachen findet sich in [Frü97].

3.1 CHR-Kalkül

3.1.1 Syntax

Im Gegensatz zur Signatur im CLP-Kalkül ist die Menge $\mathcal{P}$ der Prädikatensymbole in der Signatur im CHR-Kalkül in drei disjunkte Mengen unterteilt: $\mathcal{P}_p$ enthält die Prädikatensymbole der Basissprache, $\mathcal{P}_B$ enthält die in der Basissprache vordefinierten Constraintsymbole (engl. Built-in), und $\mathcal{P}_U$ enthält benutzerdefinierte Constraintsymbole (engl. User-defined), die durch CHR-Regeln definiert werden.

Die vordefinierten Constraints werden von einem vordefinierten Constraintlöser behandelt. Er ist ein Blackbox-System, das bestimmte Constraints zur Verfügung stellt, deren Semantik bekannt ist. Die Implementierung der vordefinierten Constraints bleibt jedoch verborgen. Die Eigenschaften des vordefinierten Constraintlösers werden durch eine Constrainttheorie CT beschrieben. Im folgenden wird davon ausgegangen, daß die Constrainttheorie zusätzlich die Eigenschaften der Gleichheit $\doteq$ als Äquivalenzrelation beschreiben muß. Hierzu genügt es, die Clarksche Gleichheitstheorie CET aus Beispiel 2.2 in CT zu integrieren. Außerdem wird davon ausgegangen, daß der Constraintlöser deterministisch und terminierend ist.

Im Rest der Arbeit werden wir uns auf die vordefinierten und benutzerdefinierten Constraints einschränken. Wie CHR mit der Basissprache interagiert und wieso man bei der Analyse von CHR auf die Basissprache verzichten kann, wird in Abschnitt 3.1.4 erläutert.

Benutzerdefinierte Constraints:	H, C_U	::=	$p(t_1, \ldots, t_n) \mid H \wedge H,\ n \geq 0$
Vordefinierte Constraints:	C, C_B	::=	$c(t_1, \ldots, t_n) \mid C \wedge C,\ n \geq 0$
Ziel:	B, G	::=	$\top \mid \bot \mid H \mid C \mid B \wedge B$
Name:	N	::=	String
CHR-Regeln:	R	::=	$N@H \Leftrightarrow C \mid B$
			$N@H \Rightarrow C \mid B$
			$N@H_1 \backslash H_2 \Leftrightarrow C \mid B$
CHR-Programme:	P	::=	$\{R_1, \ldots, R_n\}$

Abbildung 3.1: Syntax

Definition 3.2 Ein CHR-*Programm* ist eine endliche Menge von CHR-Regeln. Man unterscheidet drei Arten von CHR-*Regeln*:

Simplifikationsregel: $N@H \Leftrightarrow C \mid B$
Propagationsregel: $N@H \Rightarrow C \mid B$
Simpagationsregel: $N@H_1 \backslash H_2 \Leftrightarrow C \mid B$

Dabei ist N der Name, der die Regel identifiziert, H (bzw. H_1, H_2) eine Konjunktion von benutzerdefinierten Constraints (*Kopf*), der *Wächter* C eine Konjunktion von vordefinierten Constraints und der *Rumpf* B eine Konjunktion von benutzerdefinierten und vordefinierten Constraints. Der Rumpf B wird auch *Ziel* genannt. „|“ ist der *Commit-Operator*.
Ein CHR-Programm, das nur aus Simplifikationsregeln besteht, heißt SiR-*Programm*.

Regelnamen und Wächter sind optional. Wenn der Wächter einer Regel *true* ist, dann kann man ihn und den Commit-Operator weglassen. Man schreibt also z.B. für $H \Leftrightarrow true \mid B$ abkürzend $H \Leftrightarrow B$.

In den Implementierungen von CHR werden Konjunktionen von benutzerdefinierten Constraints durch Multimengen repräsentiert. In dieser Arbeit werden wir, wie in der obigen Definition dargestellt ist, zugunsten der besseren Übersichtlichkeit eine Multimenge von Constraints als eine Konjunktion von Constraints schreiben, deren Eigenschaften durch die unten beschriebene Kongruenz festgelegt sind.

3.1.2 Deklarative Semantik

Jeder CHR-Regel kann eine prädikatenlogische Formel über der Signatur zugeordnet werden. Eine Simplifikationsregel (bzw. eine Simpagationsregel) entspricht einer Äquivalenz, die vom Wächter impliziert wird und eine Propagationsregel einer Implikation, die vom Wächter impliziert wird.

Simplifikationsregel: $\forall \bar{x}\, \forall \bar{y}\, (C \rightarrow (H \leftrightarrow \exists \bar{z}\, B))$
Propagationsregel: $\forall \bar{x}\, \forall \bar{y}\, (C \rightarrow (H \rightarrow \exists \bar{z}\, B)$
Simpagationsregel: $\forall \bar{x}\, \forall \bar{y}\, (C \rightarrow (H_1 \wedge H_2 \leftrightarrow \exists \bar{z}\, (H_1 \wedge B)))$

Dabei sind $\bar{x}$ die in H (bzw. H_1, H_2) vorkommenden Variablen, $\bar{y}$ die Variablen, die in C aber nicht in H (bzw. H_1, H_2) vorkommen und $\bar{z}$ sind die Variablen, die nur in B vorkommen.

Zusammen mit der Constrainttheorie CT, die die Bedeutung der in einem CHR-Programm P vorkommenden vordefinierten Constraints festlegt, entspricht die Menge $\mathcal{P}$ der logischen Leseweisen der CHR-Regeln der deklarativen Semantik von P.

3.1.3 Operationale Semantik

In diesem Abschnitt wird eine verbesserte operationale Semantik von CHR vorgestellt, die die Wirkung eines CHR-Programms präziser als die in [Frü95] vorgestellte Semantik beschreibt.

Die operationale Semantik von CHR wird als Zustandsübergangssystem beschrieben. Operational entspricht eine Simplifikationsregel einer Vereinfachung des Constraintspeichers durch Ersetzung. Eine Propagationsregel entspricht der Hinzunahme weiterer Constraints. Eine Simpagationsregel entspricht gleichzeitig der Ersetzung von Constraints und der Hinzunahme weiterer Constraints. Eine Regel kann aber nur „feuern“, wenn der Wächter dieser Regel im aktuellen Kontext erfüllt ist. Der „Commit-Operator“ hat als Konsequenz, daß eine getroffene Entscheidung nicht mehr rückgängig gemacht werden kann[1].

Da CHR dazu dient, Constraintlöser zu schreiben und sie in eine beliebige Sprache einzubetten, wird der Zustand im Gegensatz zum CLP-Kalkül als Tripel definiert, damit man die Constraints, die von einem CHR-Programm behandelt werden (benutzerdefinierte Constraints), von anderen unterscheidet, die von einem Blackbox-Löser gelöst werden (vordefinierte Constraints).

Definition 3.3 Ein *Zustand* ist ein Tripel

$$\langle G, C_U, C_B \rangle,$$

wobei G ein Ziel (Zielspeicher) ist, C_U eine Konjunktion von benutzerdefinierten Constraints (benutzerdefinierter Speicher) und C_B eine Konjunktion von vordefinierten Constraints (vordefinierter Speicher). Ein leerer Ziel- oder benutzerdefinierter Speicher wird durch $\top$ dargestellt. Ein vordefinierter Speicher kann nicht leer sein. In seiner einfachsten Form besteht er entweder aus *true* oder *false*.

Intuitiv repräsentiert G die Ziele, die noch zu lösen sind, C_U bzw. C_B die benutzerdefinierten Constraints bzw. die vordefinierten Constraints, die bisher angesammelt und vereinfacht sind.

Definition 3.4 Ein *Anfangszustand* ist ein Zustand von der Form

$$\langle G, \top, \mathit{true} \rangle.$$

Ein *Endzustand* ist von einer der beiden Formen

- $\langle G, C_U, \mathit{false} \rangle$, d.h. der vordefinierte Speicher ist *false*. Er wird dann *erfolglos* genannt.
- $\langle \top, C_U, C_B \rangle$, d.h. daß keine Ziele mehr im Zielspeicher sind, keine Regel mehr auf den benutzerdefinierten Speicher C_U anwendbar ist und der vordefinierte Speicher C_B nicht *false* ist. Er wird dann *erfolgreich* genannt.

[1] Diese Tatsache geht aber nicht in das Zustandsübergangssystem ein, sondern in die darüberliegende Steuerung.

Im folgenden wird vorausgesetzt, daß alle erfolglosen Endzustände, d.h. die Zustände mit $C_B =$ *false*, miteinander identifiziert werden.

Die Unterscheidung von Endzuständen in erfolgreiche und erfolglose ist aber unnatürlich, da man mit CHR interessiert ist, Constraints zu vereinfachen. Wenn man die Inkonsistenz von Constraints feststellt, ist dies auch ein zufriedenstellendes Ergebnis. Trotzdem wird diese Unterscheidung gemacht, damit man die Korrektheits- und Vollständigkeitssätze für CHR-Programme besser mit denen für CLP-Programme vergleichen kann.

Die Kongruenz zwischen Konjunktionen von Constraints ist dieselbe wie im LP-Kalkül (Abschnitt A.2).

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

Das Zustandsübergangssystem von CHR (Abb. 3.2) ist eine Kombination der Reduktionsregel des vordefinierten Constraintlösers (**Solve**) und der Reduktionsregeln, die die Anwendung der drei Arten von CHR-Regeln beschreiben (**Simplify**, **Propagate** und **Simpagate**).

Wie im CLP-Kalkül steht $p(t_1, \ldots, t_n) \doteq q(s_1, \ldots, s_n)$ als Abkürzung für $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$, falls p und q die gleichen Prädikatensymbole sind und für *false* sonst. Außerdem steht $(p_1 \wedge \ldots \wedge p_n) \doteq (q_1 \wedge \ldots \wedge q_m)$ für $p_1 \doteq q_1 \wedge \ldots \wedge p_n \doteq q_n$, falls $n = m$ und für *false* sonst.

Solve

C ist ein vordefiniertes Constraint

$$\frac{CT \models C \wedge C_B \leftrightarrow C'_B}{\langle C \wedge G, C_U, C_B \rangle \mapsto \langle G, C_U, C'_B \rangle}$$

Introduce

$$\frac{H \text{ ist ein benutzerdefiniertes Constraint}}{\langle H \wedge G, C_U, C_B \rangle \mapsto \langle G, H \wedge C_U, C_B \rangle}$$

Simplify

$(H \Leftrightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \doteq H') \wedge C)}{\langle G, H' \wedge C_U, C_B \rangle \mapsto \langle G \wedge B, C_U, H \doteq H' \wedge C \wedge C_B \rangle}$$

Propagate

$(H \Rightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \doteq H') \wedge C)}{\langle G, H' \wedge C_U, C_B \rangle \mapsto \langle G \wedge B, H' \wedge C_U, H \doteq H' \wedge C \wedge C_B \rangle}$$

Simpagate

$(H_1 \backslash H_2 \Leftrightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H_1 \doteq H'_1 \wedge H_2 \doteq H'_2) \wedge C)}{\langle G, H'_1 \wedge H'_2 \wedge C_U, C_B \rangle \mapsto \langle G \wedge B, H'_1 \wedge C_U, H_1 \doteq H'_1 \wedge H_2 \doteq H'_2 \wedge C \wedge C_B \rangle}$$

Abbildung 3.2: CHR-Reduktionsregeln

Solve entfernt ein vordefiniertes Constraint C aus dem Zielspeicher und ersetzt den Inhalt des vordefinierten Speichers C_B durch einen neuen Inhalt C'_B, der logisch äquivalent zur Konjunktion des ursprünglichen Inhalts C_B mit dem vordefinierten Constraint C ist. Dieses C'_B berechnet der vordefinierter Constraintlöser.

Introduce transportiert ein im Zielspeicher gefundenes benutzerdefiniertes Constraint in den benutzerdefinierten Speicher. Dort kann es später durch **Simplify**, **Propagate** und **Simpagate** vereinfacht werden.

Simplify, **Propagate** und **Simpagate** sind die Zustandsübergänge, die den benutzerdefinierten Speicher gemäß dem CHR-Programm P vereinfachen oder dem benutzerdefinierten Speicher zusätzliche Informationen hinzufügen: Ein oder mehrere benutzerdefinierte Constraints werden mit den Köpfen der Regeln verglichen. Es wird immer eine Variante einer Regel mit frischen Variablen ausgewählt. Wenn die Constraints mit dem Kopf einer Regel „matchen" (spezieller als der Kopf sind) und der zugehörige Wächter bezüglich des aktuellen vordefinierten Speicher erfüllt ist, feuert diese Regel, und der Rumpf der Regel wird dem Zielspeicher hinzugefügt. Die Gleichheitsconstraints und der Wächter werden in den vordefinierten Speicher eingefügt. Die Constraints, die mit dem Kopf gematcht haben, werden aus dem benutzerdefinierten Speicher entfernt (**Simplify**) oder bleiben im benutzerdefinierten Speicher (**Propagate**) oder werden teilweise entfernt (**Simpagate**). Die Reduktionsregel **Simplify** ist ähnlich der Reduktionsregel *unfold* von NCLP-Sprachen. Der wesentliche Unterschied liegt darin, daß nicht nur ein Atom, sondern eine beliebige Konjunktion von benutzerdefinierten Constraints durch eine Regel vereinfacht werden kann.

Den **Simplify**-, **Propagate**-, bzw. **Simpagate**-Schritt nennt man auch *Anwendung* der entsprechenden Simplifikations-, Propagations- bzw. Simpagationsregel auf den Zustand.

Da bei der Anwendung einer Propagationsregel die Constraints nur erweitert werden, insbesondere sich die Kopf-Constraints, die die Anwendbarkeit entscheiden, nicht verändern, läßt sich eine Propagationsregel immer wieder auf dieselben Constraints anwenden. Um diese trivialen Zyklen zu vermeiden, sollte man sicherstellen, daß eine Propagationsregel nur einmal auf dieselben Constraints angewendet wird. Bisher wurde keine Formalisierung der operationalen Semantik vorgeschlagen, die die triviale Nichtterminierung verhindert. Eine operationale Semantik, die dies leistet, wird in Kapitel 4.3 vorgestellt.

Zur besseren Lesbarkeit werden in den folgenden Beispielen keine frischen Varianten der Regeln ausgewählt, wenn dies der Verständlichkeit dient.

Beispiel 3.5 Die informelle Vereinfachung des Zieles $\mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}$ aus Beispiel 3.1 kann als Ableitung wie folgt angegeben werden:

$$
\begin{array}{ll}
 & \langle \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \top, \mathtt{true} \rangle \\
\mapsto^3_{\mathbf{Introduce}} & \langle \top, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true} \rangle \\
\mapsto_{\mathbf{Propagate}} & \langle \mathtt{C} \leq \mathtt{B}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true} \rangle \\
\mapsto_{\mathbf{Introduce}} & \langle \top, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{B}, \mathtt{true} \rangle \\
\mapsto_{\mathbf{Simplify}} & \langle \mathtt{B} = \mathtt{C}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{true} \rangle \\
\mapsto_{\mathbf{Solve}} & \langle \top, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{B} = \mathtt{C} \rangle \\
\mapsto_{\mathbf{Simplify}} & \langle \mathtt{A} = \mathtt{B}, \top, \mathtt{B} = \mathtt{C} \rangle \\
\mapsto_{\mathbf{Solve}} & \langle \top, \top, \mathtt{A} = \mathtt{B} \wedge \mathtt{B} = \mathtt{C} \rangle
\end{array}
$$

Die obige operationale Semantik von CHR unterscheidet sich von der in [Frü95] vorgestellten Semantik in den Reduktionsregeln **Simplify**, **Propagate** und **Simpagate**. In den hier vorgestellten Reduktionsregeln wird der Wächter C in den vordefinierten Speicher eingefügt. Dies ist nötig, wie das folgende Beispiel zeigt:

Beispiel 3.6 Gegeben sei folgendes CHR-Programm:

```
p(X) ⇔ Y=2*X | r(Y)
```

Die Berechnungssequenz ausgehend vom Zustand $\langle \mathtt{p(2)}, \top, \mathtt{true} \rangle$ sieht wie folgt aus:

$$
\begin{array}{ll}
 & \langle \mathtt{p(2)}, \top, \mathtt{true} \rangle \\
\mapsto_{\mathbf{Introduce}} & \langle \top, \mathtt{p(2)}, \mathtt{true} \rangle \\
\mapsto_{\mathbf{Simplify}} & \langle \mathtt{r(Y)}, \top, \mathtt{Y} = 2 * \mathtt{X} \wedge \mathtt{X} = 2 \rangle \\
\mapsto_{\mathbf{Introduce}} & \langle \top, \mathtt{r(Y)}, \mathtt{Y} = 2 * \mathtt{X} \wedge \mathtt{X} = 2 \rangle
\end{array}
$$

Wenn man beim **Simplify**-Schritt den Wächter in den vordefinierten Speicher nicht einfügen würde, dann würde die Information, daß `Y=4` verloren gehen.

Außerdem wird die Simpagationsregel in [Frü95] nicht behandelt, da man auf dem ersten Blick den Eindruck gewinnt, daß man durch die Übersetzung einer Simpagationsregel in eine Simplifikationsregel: $H_1 \backslash H_2 \Leftrightarrow C \,|B$ entspricht $H_1 \wedge H_2 \Leftrightarrow C \mid H_1 \wedge B$, die gleiche operationale Wirkung erreicht. Dies ist aber nur der Fall, wenn das CHR-Programm ausschließlich aus Simplifikationsregeln und Simpagationsregeln besteht, wie in Abschnitt 4.3 erläutert wird.

3.1.4 Einbettung von CHR in eine Basissprache

CHR ist eine Spracherweiterung, die in eine Programmiersprache (auch *Basissprache* genannt) als Bibliothek eingebettet werden kann. Im allgemeinen ist die Basissprache eine CLP-Sprache. In der Basissprache werden Programme geschrieben, die dann neben eventuell bereits vorhandenen Constraints die in CHR implementierten Constraints benutzen können. Für den Programmierer in der Basissprache besteht kein Unterschied zwischen vordefinierten und benutzerdefinierten Constraints. CHR-Programme wiederum können in der Basissprache implementierte Prädikate als Hilfsprädikate verwenden. Zugleich können sie auch auf die bereits vordefinierten Constraints in der CLP-Sprache zurückgreifen.

Wenn im Zielspeicher Prozeduraufrufe der Basissprache vorkommen, dann werden sie dann an die Basissprache übergeben und dort weiter bearbeitet. Dies geschieht durch die Reduktionsregel **Hostcall** (Abb. 3.3).

Hostcall

$$\frac{\begin{array}{c} P \text{ ist ein Prozeduraufruf der Basissprache} \\ B \text{ ist eine Antwort auf den Prozeduraufruf } P \text{ im Kontext } C_U \wedge C_B \end{array}}{\langle P \wedge G, C_U, C_B \rangle \mapsto \langle B \wedge G, C_U, C_B \rangle}$$

Abbildung 3.3: CHR und die Basisprache

Für den Programmierer des benutzerdefinierten Constraintlösers ist CHR die verwendete Programmiersprache. Der vordefinierte Constraintlöser ist ein BlackBox-System. Mit der Basissprache kommt der Programmierer des benutzerdefinierten Constraintlösers nicht in Berührung.

So sind die Schnittstellen klar definiert, und es ergibt sich die in Abbildung 3.4 dargestellte Schichtenarchitektur zwischen Basissprache, benutzerdefiniertem Constraintlöser und vordefiniertem Constraintlöser.

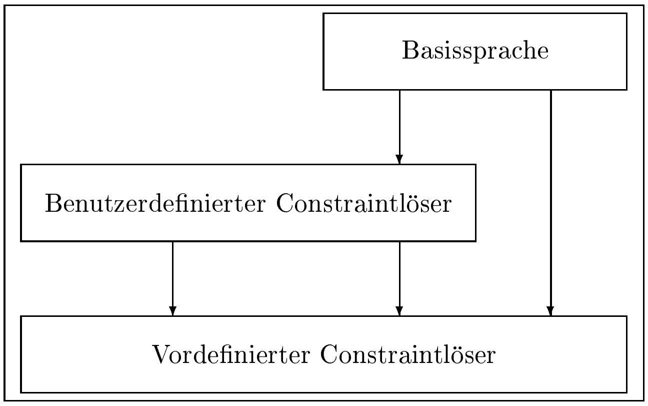

Abbildung 3.4: Schichtenarchitektur von CHR

Da die Constraintlöser durch diese Schichtung klar von der Basissprache getrennt sind, ist es möglich, die Basissprache bei der Untersuchung der Constraintlöser auszuklammern. Dies wird im folgenden auch geschehen, und es wird davon ausgegangen, daß im Zielspeicher keine Prozeduraufrufe der Basissprache vorkommen.

3.2 Erste Formalisierung der Operationalen Semantik

Bei der im Abschnitt 3.1.3 vorgestellten operationalen Semantik sind einige Punkte noch nicht hinreichend präzise. Die Behandlung der vordefinierten Constraints bedarf einer grundlegenden Verbesserung. Dies hängt stark mit der Behandlung der Variablen zusammen. Um dies zufriedenstellend darstellen zu können, muß man jedoch einen etwas aufwendigeren mathematischen Formalismus wählen. Der Zustand wird um eine weitere Komponente erweitert, die über die Variablen Buch führt, die im Zielspeicher des Anfangszustandes vorkommen. Dann wird der Zustand nach jeder Anwendung einer Reduktionsregel „normalisiert“.

Definition 3.7 Ein *Zustand* ist ein Quadrupel

$$\langle G, C_U, C_B, \mathcal{V} \rangle,$$

wobei G ein Ziel, C_U eine Konjunktion von benutzerdefinierten Constraints, C_B eine Konjunktion von vordefinierten Constraints und $\mathcal{V}$ eine Liste von Variablen (globaler Variablenspeicher) ist. Die Menge aller Zustände wird mit $\mathcal{S}$ bezeichnet.

In Analogie zu Definition 3.4 werden Anfangs- und Endzustände wie folgt definiert:

Definition 3.8 Ein *Anfangszustand* ist ein Zustand von der Form

$$\langle G, \top, \mathit{true}, \mathcal{V} \rangle,$$

wobei $\mathcal{V}$ die Liste aller Variablen ist, die in G vorkommen.
Ein *Endzustand* ist von einer der beiden Formen

- $\langle G, C_U, \mathit{false}, \mathcal{V} \rangle$, d.h. der vordefinierte Speicher ist *false*. Er wird dann *erfolglos* genannt.
- $\langle \top, C_U, C_B, \mathcal{V} \rangle$, d.h. daß keine Ziele mehr im Zielspeicher sind, keine Regel mehr auf den benutzerdefinierten Speicher C_U anwendbar ist und der vordefinierte Speicher C_B nicht false ist. Er wird dann *erfolgreich* genannt.

Definition 3.9 Eine Variable X heißt bezüglich eines Zustandes $\langle G, C_U, C_B, \mathcal{V} \rangle$

- *global*, wenn sie in $\mathcal{V}$ vorkommt.
- *lokal*, wenn sie nicht in $\mathcal{V}$ vorkommt.
- *strikt lokal*, wenn sie nur in C_B und nicht in G, C_U und $\mathcal{V}$ vorkommt.

Definition 3.10 Jedem Zustand $S = \langle G, C_U, C_B, \mathcal{V} \rangle$ wird eine Formel

$$\exists Y_1, \ldots, Y_m \;\; G \wedge C_U \wedge C_B$$

als *logische Bedeutung* (oder *logische Beschreibung*) zugeordnet, wobei $Y_1, \ldots, Y_m$ die lokalen Variablen in S sind. Die globalen Variablen von S bleiben in der Formel frei.

Normalisierungsfunktion Im folgenden wird vorausgesetzt, daß die Zustände eine eindeutige Normalform haben, die die Besonderheiten des vordefinierten Constraintlösers abstrahiert. Die Normalisierung wird mit einer Funktion modelliert, die „äquivalente“ Zustände in einen syntaktisch eindeutigen Zustand transformiert. Die Normalisierungsfunktion angewandt auf einen Zustand normalisiert seinen vordefinierten Speicher, entfernt alle strikt lokalen Variablen aus dem Zustand und propagiert Gleichheiten, die im vordefinierten Speicher explizit sind, in den benutzerdefinierten Speicher und Zielspeicher. Für die weiteren Sätze und Beweise werden die Anforderungen im folgenden präzisiert. Es wird vorausgesetzt, daß es eine Ordnung über die Variablen gibt, die in einem Zustand vorkommen. Diese Ordnung kann wie folgt definiert werden: Die globalen Variablen in der Ordnung, wie sie in $\mathcal{V}$ vorkommen, anschließend die lokalen Variablen in lexikographischer Reihenfolge.

Definition 3.11 Eine Funktion $\mathcal{N} : \mathcal{S} \to \mathcal{S}$ ist eine *Normalisierungsfunktion*, wenn sie folgende Bedingungen erfüllt:
Sei $\mathcal{N}(\langle G, C_U, C_B, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{V}\rangle$[2].

- *Propagation der Gleichheit:* G' und C'_U ergeben sich aus G und C_U, indem alle Variablen X, für die $CT \models \forall\ (C_B \to X \doteq t)$ gilt, durch den entsprechenden Term t ersetzt werden, wenn t keine Variable ist, die in der Ordnung nach X vorkommt.

- *Projektion:* Folgendes soll gelten:
$$CT \models \forall\ ((\exists \bar{x} C_B) \leftrightarrow C'_B),$$
wobei $\bar{x}$ die strikt lokalen Variablen von $\langle G', C'_U, C_B, \mathcal{V}\rangle$ sind.

- *Eindeutigkeit:* Wenn
$$\begin{array}{rcl} \mathcal{N}(\langle G_1, C_{U1}, C_{B1}, \mathcal{V}\rangle) & = & \langle G'_1, C'_{U1}, C'_{B1}, \mathcal{V}\rangle, \\ \mathcal{N}(\langle G_2, C_{U2}, C_{B2}, \mathcal{V}\rangle) & = & \langle G'_2, C'_{U2}, C'_{B2}, \mathcal{V}\rangle \text{ und} \\ CT \models (\exists \bar{x} C_{B1}) & \leftrightarrow & (\exists \bar{y} C_{B2}), \end{array}$$
gilt, wobei $\bar{x}$ bzw. $\bar{y}$ die strikt lokalen Variablen bezüglich beider Zustände sind, dann:
$$C'_{B1} = C'_{B2}.$$

Beispiel 3.12

$$\begin{array}{l} \mathcal{N}(\langle p(X) \wedge r(U,V,W), q(a,Y,Z), X \doteq Z \wedge U \doteq V \wedge W \doteq b, [X,Z,W]\rangle) = \\ \langle p(X) \wedge r(U,U,b), q(a,Y,X), X \doteq Z \wedge W \doteq b, [X,Z,W]\rangle, \end{array}$$

wenn U in der Variablenordnung V vorangeht. Hier sind X, Z, W die globalen und U, V die lokalen Variablen. Die Ordnung X, Z, W, U, V verbietet zum Beispiel, daß X durch Z ersetzt wird.

Die syntaktische Form des Ergebnisses der Normalisierung ist egal, so lange die obigen drei Bedingungen und vor allem die Eindeutigkeitsbedingung gelten. Eine wichtige Eigenschaft von $\mathcal{N}$ ist die Erhaltung der logischen Bedeutung der Zustände:

Lemma 3.13 Sei
$$\mathcal{N}(\langle G, C_U, C_B, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{V}\rangle.$$
Dann gilt folgende Äquivalenz:
$$CT \models \forall\ (\exists \bar{x}(G \wedge C_U \wedge C_B) \leftrightarrow \exists \bar{x}'(G' \wedge C'_U \wedge C'_B)),$$
wobei $\bar{x}$ bzw. $\bar{x}'$ die lokalen Variablen in $\langle G, C_U, C_B, \mathcal{V}\rangle$ bzw. $\langle G', C'_U, C'_B, \mathcal{V}\rangle$ sind.

[2] D.h. die Normalisierungsfunktion läßt den globalen Variablenspeicher unverändert.

Beweis: Die Behauptung folgt direkt aus den folgenden drei Aussagen:

$$\bar{x}' \subseteq \bar{x} \tag{3.1}$$
$$CT \models \forall\ (C_B \rightarrow ((G \wedge C_U) \leftrightarrow (G' \wedge C'_U))) \tag{3.2}$$
$$CT \models \forall\ (\exists \bar{y} C_B \leftrightarrow C'_B) \text{ und } \bar{y} \subseteq \bar{x}, \tag{3.3}$$

wobei $\bar{y}$ die strikt lokalen Variablen in $\langle G', C'_U, C_B, \mathcal{V}\rangle$. Aussage (3.1) gilt, da wegen der Projektionseigenschaft die Normalisierungsfunktion $\mathcal{N}$ keine neue Variablen einführt. Aussage (3.2) gilt, da CT Gleichheiten enthält und $G' \wedge C'_U$ sich aus $G \wedge C_U$ ergibt, indem die von C_B vorgeschriebenen Substitutionen angewandt werden. Aussage (3.3) folgt aus der Eindeutigkeitseigenschaft von $\mathcal{N}$. □

Die Eindeutigkeitsbedingung von $\mathcal{N}$ garantiert, daß es genau eine Darstellung für eine Menge von äquivalenten vordefinierten Speichern gibt. Deswegen können wir voraussetzen, daß ein inkonsistenter vordefinierter Speicher durch das Constraint *false* repräsentiert wird und ein „allgemeingültiger“ durch *true*.

Eine Eigenschaft von $\mathcal{N}$ ist, daß alle strikt lokalen Variablen eliminiert werden:

Beispiel 3.14 Sei

$$\begin{aligned}\mathcal{N}(\langle p(Z), \top, X \doteq Z, [X]\rangle) &= \langle p(X), \top, C_B, [X]\rangle \text{ und} \\ \mathcal{N}(\langle p(X), \top, true, [X]\rangle) &= \langle p(X), \top, true, [X]\rangle\end{aligned}$$

Da $CT \models \forall\ (\exists Z(X \doteq Z) \leftrightarrow true)$, impliziert die Eindeutigkeitsbedingung von $\mathcal{N}$, daß C_B *true* sein soll, da $\mathcal{N}$ keine neuen Variablen einführt.

Lemma 3.15 Die Normalisierungsfunktion ist idempotent, d.h. $\mathcal{N}(S) = \mathcal{N}(\mathcal{N}(S))$.
Beweis: Sei

$$\mathcal{N}(\langle G, C_U, C_B, \mathcal{V}\rangle) = \langle G, C'_U, C'_B, \mathcal{V}\rangle$$

Zu zeigen ist, daß dann $\mathcal{N}(\langle G', C'_U, C'_B, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{V}\rangle$ ist. Seien $\bar{x}$ die strikt lokalen Variablen in $\langle G, C_U, C_B, \mathcal{V}\rangle$. Dann gilt $CT \models (\exists \bar{x} C_B) \leftrightarrow C'_B$. C'_B enthält keine strikt lokalen Variablen, also gilt wegen der Eindeutigkeit von $\mathcal{N}$:

$$\mathcal{N}(\langle G', C'_U, C'_B, \mathcal{V}\rangle) = \mathcal{N}(\langle G, C_U, C_B, \mathcal{V}\rangle).$$

Mit $\mathcal{N}(\langle G, C_U, C_B, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{V}\rangle$ folgt dann die Behauptung. □

Durch das Reduktionssystem in Abbildung 3.5 wird eine erste formale Beschreibung der operationalen Semantik von CHR vorgestellt. Der einzige Unterschied zu der in Abschnitt 3.1.3 vorgestellten Semantik ist die Anwendung der Normalisierungsfunktion auf jeden resultierenden Zustand.

Definition 3.16 $S \mapsto^* S'$ gilt, wenn

- $S' = S$ oder
- $S' = \mathcal{N}(S)$ oder
- $S \mapsto S_1 \mapsto \ldots \mapsto S_n \mapsto S' \quad (n \geq 0)$.

Definition 3.17 Eine *Berechnung* (oder *Berechnungssequenz*) ist eine wiederholte Anwendung $S_1 \mapsto S_2 \mapsto \ldots$ von Reduktionsschritten oder die Anwendung der Normalisierungsfunktion.
Eine *Ableitung* ist eine Berechnung von der Form $S_1 \mapsto \ldots \mapsto S_n$, wobei S_n ein Endzustand ist, oder eine unendliche Berechnung. Wenn die Einzelschritte nicht von Interesse sind, kann man auch $S_1 \mapsto^* S_n$ schreiben.

Die Definitionen für die Arten der Ableitungen und Ziele aus dem im Abschnitt 3.1.3 vorgestellten operationalen Semantik gelten weiterhin.

Solve

$$\frac{C \text{ ist ein vordefiniertes Constraint}}{\langle C \wedge G, C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G, C_U, C \wedge C_B, \mathcal{V}\rangle)}$$

Introduce

$$\frac{H \text{ ist ein benutzerdefiniertes Constraint}}{\langle H \wedge G, C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G, H \wedge C_U, C_B, \mathcal{V}\rangle)}$$

Simplify

$(H \Leftrightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \doteq H' \wedge C))}{\langle G, H' \wedge C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, C_U, H \doteq H' \wedge C \wedge C_B, \mathcal{V}\rangle)}$$

Propagate

$(H \Rightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \doteq H') \wedge C)}{\langle G, H' \wedge C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, H' \wedge C_U, H \doteq H' \wedge C \wedge C_B, \mathcal{V}\rangle)}$$

Simpagate

$$\frac{\begin{array}{c}(H_1 \backslash H_2 \Leftrightarrow C \mid B) \text{ ist eine frische Variante einer Regel aus } P \text{ mit den Variablen } \bar{x} \\ CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H_1 \doteq H_1' \wedge H_2 \doteq H_2') \wedge C)\end{array}}{\langle G, H_1' \wedge H_2' \wedge C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, H_1' \wedge C_U, H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C \wedge C_B, \mathcal{V}\rangle)}$$

Abbildung 3.5: CHR-Reduktionsregeln: Erste Formalisierung

3.3 Korrektheit und Vollständigkeit des CHR-Kalküls

Die Angabe der operationalen Semantik eines Programmes findet darin ihre Bestätigung, daß die operationale Semantik genau die Vorgänge widerspiegelt, die im Rechner vor sich gehen, wenn ein Ziel bearbeitet wird. Die deklarative Semantik gibt dem Betrachter in den meisten Fällen zwar eine intuitivere Vorstellung von der Bedeutung des Programmes, bedarf jedoch einer Rechtfertigung, wenn sie tatsächlich als Bedeutung des Programmes angesehen werden will. Diese Rechtfertigung wird in diesem Abschnitt gegeben, indem Korrektheits- und Vollständigkeitssätze bewiesen werden, die einen Zusammenhang zwischen deklarativer und operationaler Semantik beschreiben.

Korrektheit und Vollständigkeit des CHR-Kalküls sind unabhängig von der Art der Ableitung. Eine Unterscheidung in erfolgreiche und erfolglose Ableitungen wie in den bisherigen Kalkülen ist nicht notwendig. Statt mit Implikationen zwischen Ziel und Antwortconstraint haben wir es mit logischen Äquivalenzen zu tun. Die Resultate basieren vor allem darauf, daß Reduktionen für CHR die Äquivalenz von den Zuständen erhalten:

Definition 3.18 Ein *herleitbares Constraint eines Zieles G* ist die logische Beschreibung eines Zustandes, der in einer Ableitung beginnend mit $\langle G, \top, \mathit{true}, \mathcal{V}\rangle$ vorkommt.
Ein *Antwortconstraint eines Zieles G* ist die logische Beschreibung eines Endzustandes einer Ableitung, die mit $\langle G, \top, \mathit{true}, \mathcal{V}\rangle$ beginnt.

Lemma 3.19 Sei P ein CHR-Programm und G ein Ziel. Wenn C ein herleitbares Constraint von G ist, dann gilt

$$\mathcal{P} \cup CT \models \forall\, (C \leftrightarrow G).$$

Beweis: Durch strukturelle Induktion über die Ableitungen:

Induktionsanfang: Keine Transition ist auf den Anfangszustand $\langle G, \top, \mathit{true}, \mathcal{V}\rangle$ anwendbar, das heißt, $G = \top$ und $C = \mathit{true}$ Dann gilt

$$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G),$$

weil der leere Zielspeicher $\top$ und das Constraint *true* von jeder Interpretation erfüllt wird.
Induktionsschritt: Wir haben die folgende Ableitung

$$\langle G, \top, \mathit{true}, \mathcal{V}\rangle \mapsto^* \langle G_1', C_U', C_B', \mathcal{V}\rangle \mapsto \langle G_1'', C_U'', C_B'', \mathcal{V}\rangle.$$

Um zu beweisen, daß der letzte Schritt die logische Äquivalenz erhält, beweisen wir, daß jeder Ableitungsschritt der operationalen Semantik die logische Äquivalenz erhält:

(1) **Solve**: Dann ist G_1' von der Form $C \wedge G_1$, wobei C ein vordefiniertes Constraint ist. Der Ableitungsschritt angewandt auf den Zustand $\langle C \wedge G_1, C_U', C_B', \mathcal{V}\rangle$ führt zu dem neuen Zustand $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle = \mathcal{N}(\langle G_1, C_U', C_B' \wedge C, \mathcal{V}\rangle)$.

Seien $\bar{x}_1$ die lokalen Variablen von $\langle C \wedge G_1, C_U', C_B', \mathcal{V}\rangle$. Nach Induktionsvoraussetzung gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (\exists \bar{x}_1\ (C \wedge G_1 \wedge C_U' \wedge C_B') \leftrightarrow G).$$

Laut Lemma 3.13 gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (\exists \bar{x}_1\ (C \wedge G_1 \wedge C_U' \wedge C_B') \leftrightarrow \exists \bar{x}_2 (G_1'' \wedge C_U'' \wedge C_B'')),$$

wobei $\bar{x}_2$ die lokalen Variablen von $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle$ sind. Deswegen gilt

$$\mathcal{P} \cup CT \models \forall\ (\exists \bar{x}_2\ (G_1'' \wedge C_U'' \wedge C_B'') \leftrightarrow G).$$

(2) **Introduce**: Dann ist G_1' von der Form $C \wedge G_1$, wobei C ein benutzerdefiniertes Constraint ist. **Introduce** angewandt auf den Zustand $\langle C \wedge G_1, C_U', C_B', \mathcal{V}\rangle$ führt zu dem neuen Zustand $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle = \mathcal{N}(\langle G_1, C \wedge C_U', C_B', \mathcal{V}\rangle)$. Seien $\bar{x}_1$ die lokalen Variablen des Zustandes $\langle C \wedge G_1, C_U', C_B', \mathcal{V}\rangle$. Nach Induktionsvoraussetzung gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (\exists \bar{x}_1\ (C \wedge G_1 \wedge C_U' \wedge C_B') \leftrightarrow G).$$

$\bar{x}_1$ sind auch die lokalen Variablen von dem Zustand $\langle G_1, C \wedge C_U', C_B', \mathcal{V}\rangle$. Da beim Transportieren eines benutzerdefinierten Constraints der vordefinierte Speicher nicht verändert wird und $\mathcal{N}$ laut Lemma 3.15 idempotent ist, sind x_1 auch die lokalen Variablen von dem Zustand $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle$, und es gilt $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle = \langle G_1, C \wedge C_U', C_B', \mathcal{V}\rangle$. Es gilt also

$$\begin{aligned}\mathcal{P} \cup CT \models \forall\ (G \quad &\leftrightarrow \quad \exists \bar{x}_1\ (C \wedge G_1 \wedge C_U' \wedge C_B')) \\ &\leftrightarrow \quad \exists \bar{x}_1\ (G_1'' \wedge C_U'' \wedge C_B'')\end{aligned}$$

(3) **Simplify**: Dann ist C_U' von der Form $H' \wedge C_U$, wobei $(H \Leftrightarrow C \mid B)$ eine frische Variante einer Simplifikationsregel aus P und $CT \models \forall\ (C_B' \rightarrow \exists \bar{y}\ (H \doteq H' \wedge C))$.
Der Ableitungsschritt angewandt auf den Zustand $\langle G_1', H' \wedge C_U, C_B', \mathcal{V}\rangle$ führt zu dem neuen Zustand $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle = \mathcal{N}(\langle G_1' \wedge B, C_U, H \doteq H' \wedge C \wedge C_B', \mathcal{V}\rangle)$. Seien $\bar{x}_1$ die lokalen Variablen von $\langle G_1', H' \wedge C_U, C_B', \mathcal{V}\rangle$. Nach Induktionsvoraussetzung gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ (G_1' \wedge H' \wedge C_U \wedge C_B') \leftrightarrow G). \tag{3.4}$$

Der Folgerungstest $CT \models \forall\ (C_B' \rightarrow \exists \bar{y}\ (H \doteq H' \wedge C))$ besagt, daß der Kontext C_B' zu seiner Konjunktion mit dem instantiierten Wächter äquivalent ist:

$$CT \models \forall\ (C_B' \leftrightarrow (C_B' \wedge \exists \bar{y}\ (H \doteq H' \wedge C))).$$

Da $\bar{y}$ nicht in C_B' vorkommen, gilt folgendes:

$$CT \models \forall\ (C_B' \leftrightarrow \exists \bar{y}\ (C_B' \wedge H \doteq H' \wedge C)).$$

Nach Ersetzen von C'_B durch $\exists\bar{y}\ (C'_B \wedge H \doteq H' \wedge C)$ in der Äquivalenz (3.4) erhalten wir:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ (G'_1 \wedge H' \wedge C_U \wedge \exists\bar{y}\ (C'_B \wedge H \doteq H' \wedge C)) \leftrightarrow G).$$

Die Variablen $\bar{y}$ kommen nicht in $G'_1 \wedge H' \wedge C_U$ vor, also gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ \exists\bar{y}\ (G'_1 \wedge H' \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge C) \leftrightarrow G).$$

Da $CT \models H \doteq H' \rightarrow (H \leftrightarrow H')$, erhält man:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ \exists\bar{y}\ (G'_1 \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge H \wedge C) \leftrightarrow G). \tag{3.5}$$

Da die Simplifikationsregel $(H \Leftrightarrow C \mid B)$ aus P ist, ist ihre logische Leseweise eine logische Folgerung aus der Menge $\mathcal{P}$ der logischen Leseweisen der CHR-Regeln von P und der Constrainttheorie CT: $\mathcal{P} \cup CT \models \forall\ ((C \rightarrow (H \leftrightarrow \exists\bar{y}_2\ B)))$. Daraus folgt dann:

$$\mathcal{P} \cup CT \models \forall\ ((H \wedge C) \leftrightarrow \exists\bar{y}_2\ (B \wedge C)).$$

Nach Ersetzen von $H \wedge C$ durch $\exists\bar{y}_2\ (B \wedge C)$ in Äquivalenz (3.5) erhalten wir:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ \exists\bar{y}\ (G'_1 \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge \exists\bar{y}_2\ (B \wedge C)) \leftrightarrow G).$$

Die Variablen $\bar{y}_2$ kommen nur in B vor, damit folgt:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ \exists\bar{y}\ \exists\bar{y}_2\ (G'_1 \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge B \wedge C) \leftrightarrow G).$$

Die Variablen $\bar{y}_2$ sind die Variablen, die nur in B vorkommen und $\bar{y}$ sind die frischen Variable der Simplifikationsregel, dann sind $\bar{x}_2 = \bar{x}_1 \cup \bar{y}_2 \cup \bar{y}$ die lokalen Variablen von dem Zustand $\langle G'_1 \wedge B, C_U, H \doteq H' \wedge C \wedge C'_B, \mathcal{V}\rangle$:

$$\mathcal{P} \cup CT \models \forall\ (\exists\bar{x}_2\ (G'_1 \wedge B \wedge C_U \wedge H \doteq H' \wedge C \wedge C'_B) \leftrightarrow G).$$

Seien $\bar{x}_3$ die lokalen Variablen von $\langle G''_1, C''_U, C''_B, \mathcal{V}\rangle$, dann gilt nach Lemma 3.13:

$$\mathcal{P} \cup CT \models \forall\ (\exists\bar{x}_2\ (G'_1 \wedge B \wedge C_U \wedge H \doteq H' \wedge C \wedge C'_B) \leftrightarrow \exists\bar{x}_3\ (G''_1 \wedge C''_U \wedge C''_B)).$$

Folglich gilt:

$$\mathcal{P} \cup CT \models \forall\ (\exists\bar{x}_3\ (G''_1 \wedge C''_U \wedge C''_B) \leftrightarrow G).$$

(4) **Propagate**: Dann ist C'_U von der Form $H' \wedge C_U$, wobei $(H \Rightarrow C \mid B)$ eine frische Variante einer Propagationsregel aus P und $CT \models \forall\ (C'_B \rightarrow \exists\bar{y}\ (H \doteq H' \wedge C))$.
Der Ableitungsschritt angewandt auf den Zustand $\langle G'_1, H' \wedge C_U, C'_B, \mathcal{V}\rangle$ führt zu dem neuen Zustand $\langle G''_1, C''_U, C''_B, \mathcal{V}\rangle = \mathcal{N}(\langle G'_1 \wedge B, H' \wedge C_U, C \wedge H \doteq H' \wedge C'_B, \mathcal{V}\rangle)$.

Seien $\bar{x}_1$ die lokalen Variablen von $\langle G'_1, H' \wedge C_U, C'_B, \mathcal{V}\rangle$. Nach Induktionsvoraussetzung gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ (G'_1 \wedge H' \wedge C_U \wedge C'_B) \leftrightarrow G).$$

Mit der gleichen Argumentation wie im Falle **Simplify** erhalten wir folgende Äquivalenz:

$$\mathcal{P} \cup CT \models \forall\ (\exists x_1\ \exists\bar{y}\ (G'_1 \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge H \wedge C) \leftrightarrow G). \tag{3.6}$$

Da $(H \Rightarrow C \mid B)$ eine Propagationsregel aus P ist, gilt $\mathcal{P} \cup CT \models \forall\ ((C \rightarrow (H \rightarrow \exists\bar{y}_2\ B)))$. Daraus folgt dann:

$$\mathcal{P} \cup CT \models \forall\ ((H \wedge C) \rightarrow \exists\bar{y}_2\ (B \wedge C))$$

Durch Hinzufügen von $\exists\bar{y}_2(B \wedge C)$ in der linken Seite der Äquivalenz (3.6) erhalten wir:

$$\mathcal{P} \cup CT \models \forall\ (\exists\bar{x}_1\ \exists\bar{y}\ (G'_1 \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge H \wedge C \wedge \exists\bar{y}_2\ (B \wedge C)) \leftrightarrow G).$$

Die Variablen $\bar{y}_2$ kommen nur in B vor, dann gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_1\ \exists\bar{y}\ \exists\bar{y}_2\ (G_1' \wedge C_U \wedge C_B' \wedge H \doteq H' \wedge H \wedge C \wedge B \wedge C) \leftrightarrow G).$$

Dies ist äquivalent zu folgender Aussage:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_1\ \exists\bar{y}\ \exists\bar{y}_2\ (G_1' \wedge C_U \wedge C_B' \wedge H \doteq H' \wedge H' \wedge C \wedge B) \leftrightarrow G).$$

Die Variablen $\bar{y}_2$ sind die Variablen, die nur in B vorkommen und $\bar{y}$ sind die frischen Variable der Propagationsregel, dann sind $\bar{x}_2 = \bar{x}_1 \cup \bar{y}_2 \cup \bar{y}$ die lokalen Variablen von dem Zustand $\langle G_1' \wedge B, H' \wedge C_U, H \doteq H' \wedge C_B', \mathcal{V}\rangle$:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_2\ (G_1' \wedge B \wedge H' \wedge C_U \wedge H \doteq H' \wedge C_B') \leftrightarrow G).$$

Seien $\bar{x}_3$ die lokalen Variablen von $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle$, dann gilt nach Lemma 3.13:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_2\ (G_1' \wedge B \wedge H' \wedge C_U \wedge H \doteq H' \wedge C_B') \leftrightarrow \exists\bar{x}_3\ (G_1'' \wedge C_U'' \wedge C_B'')).$$

Folglich gilt:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_3\ (G_1'' \wedge C_U'' \wedge C_B'') \leftrightarrow G).$$

(5) **Simpagate**: Dann ist C_U' von der Form $H_1' \wedge H_2' \wedge C_U$, wobei $(H_1 \backslash H_2 \Leftrightarrow C \mid B)$ eine frische Variante einer Simpagationsregel aus P und $CT \models \forall\,(C_B' \rightarrow \exists\bar{y}\ (H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C))$. Der Ableitungsschritt angewandt auf den Zustand $\langle G_1', H_1' \wedge H_2' \wedge C_U, C_B', \mathcal{V}\rangle$ führt zu dem neuen Zustand $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle = \mathcal{N}(\langle G_1' \wedge B, H_1' \wedge C_U, H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C \wedge C_B', \mathcal{V}\rangle)$. Seien $\bar{x}_1$ die lokalen Variablen von $\langle G_1', H_1' \wedge H_2' \wedge C_U, C_B', \mathcal{V}\rangle$. Nach Induktionsvoraussetzung gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\,(\exists x_1\ (G_1' \wedge H_1' \wedge H_2' \wedge C_U \wedge C_B') \leftrightarrow G). \tag{3.7}$$

Mit der gleichen Argumentation wie im Falle **Simplify** erhalten wir folgende Äquivalenz:

$$\mathcal{P} \cup CT \models \forall\,(\exists x_1\ \exists\bar{y}\ (G_1' \wedge C_U \wedge C_B' \wedge H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge H_1 \wedge H_2 \wedge C) \leftrightarrow G). \tag{3.8}$$

Da $(H_1 \backslash H_2 \Leftrightarrow C \mid B)$ eine Simpagationsregel aus P ist, gilt $\mathcal{P} \cup CT \models \forall\,((C \rightarrow (H_1 \wedge H_2 \leftrightarrow \exists\bar{y}_2\ (H_1 \wedge B))))$. Daraus folgt

$$\mathcal{P} \cup CT \models \forall\,((H_1 \wedge H_2 \wedge C) \leftrightarrow \exists\bar{y}_2\ (H_1 \wedge B \wedge C))$$

Nach Ersetzen von $H_1 \wedge H_2 \wedge C$ durch $\exists\bar{y}_2\ (H_1 \wedge B \wedge C)$ in Äquivalenz (3.8) erhalten wir:

$$\mathcal{P} \cup CT \models \forall\,(\exists x_1\ \exists\bar{y}\ (G_1' \wedge C_U \wedge C_B' \wedge H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge \exists\bar{y}_2\ (H_1 \wedge B \wedge C)) \leftrightarrow G).$$

Die Variablen $\bar{y}_2$ kommen nur in B vor, dann folgt folgendes:

$$\mathcal{P} \cup CT \models \forall\,(\exists x_1\ \exists\bar{y}\ \exists\bar{y}_2\ (G_1' \wedge C_U \wedge C_B' \wedge H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge H_1 \wedge B \wedge C) \leftrightarrow G).$$

Die Variablen $\bar{y}_2$ sind die Variablen, die nur in B vorkommen und die Variablen $\bar{y}$ sind die frischen Variablen aus der Simpagationsregel, dann sind $\bar{x}_2 = \bar{x}_1 \cup \bar{y}_2 \cup \bar{y}$ die lokalen Variablen von dem Zustand $\langle G_1' \wedge B, H_1 \wedge C_U, H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C \wedge C_B', \mathcal{V}\rangle$:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_2\ (G_1' \wedge B \wedge H_1 \wedge C_U \wedge H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C \wedge C_B') \leftrightarrow G).$$

Seien $\bar{x}_3$ die lokalen Variablen von $\langle G_1'', C_U'', C_B'', \mathcal{V}\rangle$, dann gilt nach Lemma 3.13:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_2\ (G_1' \wedge B \wedge H_1 \wedge C_U \wedge H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C \wedge C_B') \leftrightarrow \exists\bar{x}_3\ (G_1'' \wedge C_U'' \wedge C_B'')).$$

Folglich gilt:

$$\mathcal{P} \cup CT \models \forall\,(\exists\bar{x}_3\ (G_1'' \wedge C_U'' \wedge C_B'') \leftrightarrow G).$$

$\square$

Aus diesem Lemma folgt unmittelbar der folgende Korrektheitssatz, da ein Antwortconstraint auch ein herleitbares ist:

Satz 3.20 Sei P ein CHR-Programm und G ein Ziel. Wenn G eine Ableitung mit einem Antwortconstraint C hat, dann gilt

$$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G).$$

Aus Lemma 3.19 folgt auch, daß alle herleitbaren Constraints eines Zieles logisch äquivalent sind:

Korollar 3.21 Sei P ein CHR-Program und G ein Ziel. Dann gilt für alle herleitbaren Constraints C_1 und C_2 von G folgendes:

$$\mathcal{P} \cup CT \models \forall\ (C_1 \leftrightarrow C_2).$$

Der folgende Satz ist stärker als das Vollständigkeitsergebnis für CLP-Sprachen [Mah87]. Die Disjunktion von Satz 2.16 kann zu einem einzigen Disjunkt reduziert werden, da nach Korollar 3.21 alle herleitbaren Constraints von einem gegebenen Ziel logisch äquivalent sind (Abbildung 3.6). Dies ist möglich, da die deklarative Semantik von CHR „stärker" als die der CLP-Sprachen ist. Für CLP-Programme existiert kein vergleichbares Vollständigkeitsergebnis [Mah87] (Siehe Beispiel 2.17).

Abbildung 3.6: Zusammenhang zwischen herleitbaren Constraints in CLP (A) und CHR (B)

Satz 3.22 Sei P ein CHR-Programm und G ein Ziel, das mindestens eine endliche Ableitung hat. Wenn $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$ gilt, dann hat G eine Ableitung mit Antwortconstraint C', wobei

$$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C').$$

Beweis: G hat mindestens eine endliche Ableitung. Sei C' das Antwortconstraint von G resultierend aus dieser Ableitung.
Nach dem Korrektheitssatz 3.20 gilt folgendes:

$$\mathcal{P} \cup CT \models \forall(C' \leftrightarrow G)$$

Aus $\mathcal{P} \cup CT \models \forall(C \leftrightarrow G)$ folgt dann $\mathcal{P} \cup CT \models \forall(C \leftrightarrow C')$. □

Der Vollständigkeitssatz gilt nicht, wenn G keine endlichen Ableitungen hat:

Beispiel 3.23 Sei P das CHR-Programm:

`p` $\Leftrightarrow$ `p.`

Gegeben sei G das Ziel `p`. Es gilt $\mathcal{P} \cup CT \models \mathtt{p} \leftrightarrow \mathtt{p}$. Aber G hat nur unendliche Ableitungen.

Der Korrektheitssatz 3.20 gilt sowohl für erfolgreiche als auch für erfolglose Ableitungen:

Korollar 3.24 Sei P ein CHR Programm und G ein Ziel. Wenn G eine erfolglose Ableitung hat, dann gilt:

$$\mathcal{P} \cup CT \models \neg\exists G.$$

Beweis: Wenn G eine erfolglose Ableitung hat, dann hat G eine Ableitung mit Antwortconstraint äquivalent zu *false*. Nach Satz 3.20 gilt dann: $\mathcal{P} \cup CT \models \forall\ (\mathit{false} \leftrightarrow G)$, daher gilt auch: $\mathcal{P} \cup CT \models \forall\neg G$. Dies ist äquivalent zu: $\mathcal{P} \cup CT \models \neg\exists G$. □

Die Vollständigkeitsaussage von Satz 3.22 ist für erfolglose Ableitungen allerdings unbefriedigend, weil das Antwortconstraint nicht explizit inkonsistent (d.h. identisch zu *false*) sein muß: Wenn $\mathcal{P} \cup CT \models \neg\exists G$ gilt, dann hat G eine Ableitung mit Antwortconstraint C', wobei

$$\mathcal{P} \cup CT \models \forall\ (\mathit{false} \leftrightarrow C').$$

Insbesondere kann das Antwortconstraint C' identisch zu G sein, wie das folgende Beispiel zeigt:

Beispiel 3.25 Sei P das folgende CHR-Programm

```
p ⇔ q.
p ⇔ false.
```

Es gilt $\mathcal{P} \cup CT \models \neg$`q`. Aber `q` hat keine gescheiterte Ableitung, sondern eine erfolgreiche mit Antwortconstraint `q`.

Die Umkehrrichtung von Satz 3.24 gilt also nicht. Wir werden in Abschnitt 6.1.2 sehen, daß man bessere Vollständigkeitsergebnisse für konfluente Programme geben kann.

Kapitel 4

Konfluenz von Constraint Handling Rules

4.1 Motivation

„Committed choice"-Sprachen haben den Vorteil, daß bei der Auswertung eines Zieles nicht ein ganzer Suchbaum aufgespannt wird, sondern nur eine lineare Berechnungssequenz nötig ist. Dies hat aber den Nachteil, daß einem durch eine ungünstige Regelauswahl und wegen des „don't care"-Nichtdeterminismus „einfachere" Lösungen verlorengehen können. Da CHR eine „committed choice"-Sprache ist, muß sich der Benutzer mit diesem Problem auseinandersetzen. Es wäre also nützlich, wenn er die Möglichkeit hätte, sicherzustellen, daß die Antwortconstraints immer gleich ausfallen, egal welche Regeln in einer Berechnungssequenz ausgewählt wurden. Diese essentielle Eigenschaft eines Constraintlösers, die in diesem Kapitel untersucht wird, nennt man *Konfluenz*. Bei CHR-Programmen hat man zwar wegen der Korrektheitssätze aus Abschnitt 3.3 die Garantie, eine „korrekte" Antwort zu bekommen, es ist jedoch möglich, daß der Constraintlöser durch ungünstige Regelauswahl nicht die einfachste Antwort liefert, im Extremfall sogar eine Inkonsistenz nicht erkennt. Der Hauptgrund liegt im allgemeinen entweder an der „Unvollständigkeit" des Constraintlösers, wie Beispiel 4.1 und Beispiel 4.2 zeigen, oder an Programmfehlern, wie Beispiel 4.3 erörtert. Die Beispiele sind Ausschnitte existierender Constraintlöser, die bestimmte Constraintsysteme implementieren. Eine detaillierte Beschreibung von Constraintsystemen in der Sprache CHR findet sich in [FA97b].

Beispiel 4.1 Gegeben sei ein CHR-Programm, das die Aufzählungsbereiche implementiert. Das Constraint `X::L` bedeutet, daß `X` einen Wert aus der Liste `L` annehmen muß, wobei `durchschnitt/3` ein vordefiniertes Constraintsymbol ist.

```
nicht_leer @ X::[]  ⇔  false.

binden @ X::[Y]  ⇔  X=Y.

durchschnitt @ X::L1 ∧ X::L2  ⇔
        durchschnitt(L1,L2,L3) ∧ X::L3.
```

Die Regel `nicht_leer` stellt sicher, daß keine Variable einen Wert aus der leeren Liste annehmen kann. Die Regel `binden` macht den Wert einer eindeutig bestimmten (determinierten) Variablen explizit. Die Regel `durchschnitt` ersetzt zwei Aufzählungsbereiche für dieselbe Variable durch einen einzigen Aufzählungsbereich, indem sie den Durchschnitt der Aufzählungsbereiche bildet.

Gegeben sei das Ziel `X::[3] ∧ X::[4,6,7]`. Wenn man die Regel `binden` auswählt, dann erhält man die Antwort `X = 3 ∧ 3::[4,6,7]`. Wenn aber anstatt der Regel `binden` die Regel `durchschnitt` ausgewählt wird, erhält man die „aussagekräftigere“ Antwort `false`. Daher ist das CHR-Programm nicht konfluent.

$$\begin{tikzcd}
 & \mathtt{X} :: [3] \wedge \mathtt{X} :: [4,6,7] \arrow[dl, "\mathtt{binden}"'] \arrow[dr, "\mathtt{durchschnitt}"] & \\
\mathtt{X} = 3 \wedge 3 :: [4,6,7] & & \mathtt{X} :: [\,] \arrow[d, "\mathtt{nicht_leer}"] \\
 & & \mathtt{false}
\end{tikzcd}$$

Es sind weitere Regeln für gebundene Variablen notwendig, die sich aus den Regeln für ungebundene Variablen spezialisiert auf den einelementigen Wertebereich ergeben. Dies kann man sich ersparen, wenn man für `binden` eine Propagationssregel wählt, so daß der Wertebereich erhalten bleibt:

```
binden @ X::[Y]  ⇒  X=Y.
```

Beispiel 4.2 Gegeben sei ein in CHR geschriebener Constraintlöser für $<$. Die Sprache CHR ermöglicht u.a. die schnelle Erstellung von Erweiterungen. Wir erweitern dann den Constraintlöser für $<$ um die Constrainttheorie für $\leq$, = und $\neq$, indem wir folgende Regeln hinzufügen. In diesem Beispiel ist die Gleichheit $=$ kein vordefiniertes Constraintsymbol, sondern ein benutzerdefiniertes.

```
r1 @ X ≤ Y ∧ Y ≤ X  ⇔  X = Y.
r2 @ X = Y ∧ X ≠ Y   ⇔  false.
r3 @ X ≤ Y ∧ X ≠ Y  ⇔  X < Y.
```

Gegeben sei das Ziel $\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}$. Mit der ersten Regel `r1` reduziert sich dieses Ziel zu $\mathtt{X} = \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Y}$ und dies führt mit der Regel `r2` zu *false*. Wenn man anstatt der ersten Regel die Regel `r3` auswählt, dann bekommt man die Antwort $\mathtt{X} < \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}$.

$$\begin{tikzcd}
 & \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \arrow[dl, "\mathtt{r3}"'] \arrow[dr, "\mathtt{r1}"] & \\
\mathtt{X} < \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} & & \mathtt{X} = \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Y} \arrow[d, "\mathtt{r2}"] \\
 & & \mathtt{false}
\end{tikzcd}$$

In diesem Beispiel sieht man, daß eine Regel für die Interaktion zwischen den Constraintsymbolen $<$ und $\leq$ vergessen wurde:

```
r4 @ X < Y ∧ Y ≤ X  ⇔  false.
```

Beispiel 4.3 Sei P ein CHR-Programm, das das Constraintsymbol `max` implementiert, wobei der Rumpf der zweiten Regel einen Tippfehler enthält.

```
r1 @ max(X,Y,Z) ⇔ X≤Y | Z = Y.
r2 @ max(X,Y,Z) ⇔ Y≤X | Y = X.
```

Gegeben sei das Ziel `max(X,X,Z)`. Mit der ersten Regel reduziert sich das Ziel zu `Z = X` aber mit der zweiten zu `X = X`.

$$\begin{tikzcd}
 & \mathtt{max(X,X,Z)} \arrow[dl, "\mathtt{r1}"'] \arrow[dr, "\mathtt{r2}"] & \\
\mathtt{Z} = \mathtt{X} & & \mathtt{X} = \mathtt{X}
\end{tikzcd}$$

Hier sieht man, daß dem Benutzer ein Tippfehler unterlaufen ist. Anstatt `Z = X` hat der Benutzer das Gleichheitsconstraint `Y = X` geschrieben.

Die obigen Beispiele zeigen, daß zu einem Zeitpunkt im allgemeinen mehrere CHR-Regeln anwendbar sind, um Constraints zu vereinfachen. Da aber bei der Implementierung eines Constraintlösers Fehler unterlaufen können oder manchmal Abstriche bei der Vollständigkeit gemacht werden, um effizient zu bleiben, ist es nicht immer garantiert, daß das Ergebnis einer Berechnung immer das gleiche ist, egal welche CHR-Regeln ausgewählt wurden. Im Gegensatz zu den Korrektheitssätzen aus Abschnitt 3.3, die nur semantische Äquivalenz der hergeleiteten Zustände garantieren, garantiert die Konfluenz, daß zu einem Ziel immer die (syntaktisch) gleiche Antwort hergeleitet wird, egal welche Auswahlstrategie für Regeln und Ziele verwendet wird. Ein CHR-Programm heißt dann konfluent, wenn je zwei endliche Ableitungen des gleichen Zieles zusammengeführt werden können, d.h. syntaktisch gleiche Endzustände haben (bis auf Variantenbildung).

Die Konfluenz ist auch eine grundlegende Eigenschaft von Termersetzungssystemen, deren Einsatz für andere Zwecke als der Einsatz von CHR gedacht ist. CHR-Programme unterscheiden sich jedoch grundlegend von Termersetzungssystemen insofern, daß sie logische Variablen beinhalten, die nicht wie in Termersetzungssystemen behandelt werden können. Da aber die Ähnlichkeit zwischen beiden Formalismen sehr groß ist, werden wir uns bei der Untersuchung der Konfluenz auf existierende Arbeiten im Bereich bedingter Termersetungssysteme [DOS88] stützen. Dieser Abschnitt gibt Kriterien an, unter denen ein CHR-Programm konfluent ist. Die Konfluenz ist im allgemeinen nicht entscheidbar, aber wenn man sich auf terminierende CHR-Programme beschränkt, dann wird sie mit den hier vorgestellten Ergebnissen entscheidbar. Für terminierende CHR-Programme reicht für die Konfluenz die lokale Konfluenz aus.

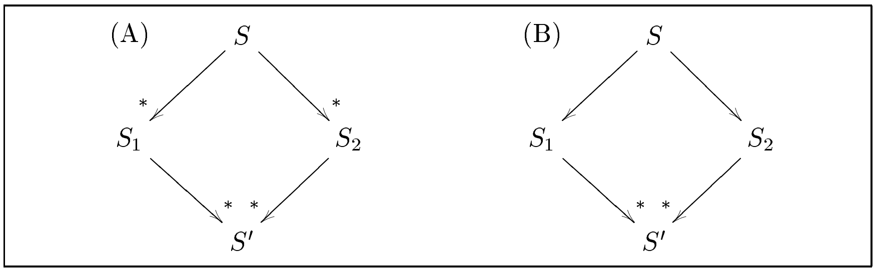

Abbildung 4.1: Konfluenz (A) und Lokale Konfluenz (B)

Bei der Konfluenz wird folgendes gefordert: Sind S_1, S_2 beliebige Nachfolger von S, so sind S_1 und S_2 zusammenführbar (Abbildung 4.1 (A)). Bei der lokalen Konfluenz wird dies nur für direkte Nachfolger S_1, S_2 von S gefordert (Abbildung 4.1 (B)).

Im folgenden werden einige Definitionen angegeben, die nötig sind, um ein Kriterium anzugeben, unter dem ein CHR-Programm konfluent ist.

Definition 4.4 Ein CHR-Programm heißt *terminierend*, falls es keine unendliche Ableitung $S_0 \mapsto S_1 \mapsto S_2 \ldots$ gibt.

Definition 4.5 Zwei Zustände S_1 und S_2 heißen *zusammenführbar* ($S_1 \downarrow S_2$), wenn es zwei Zustände S_1' und S_2' gibt, so daß $S_1 \mapsto^* S_1'$ und $S_2 \mapsto^* S_2'$ gilt und S_1' und S_2' Varianten sind.

Die Zusammenführbarkeit in CHR ist durch die Varianz gegenüber Termersetzungssysteme verallgemeinert. In Termersetzungssystemen werden Variablen in Termen wie Konstanten behandelt. Aus diesem Grund können die Konfluenzkriterien für Termersetzungssysteme nicht unmittelbar auf CHR-Programme übertragen werden.

Definition 4.6 Ein CHR-Programm heißt *konfluent*, wenn für alle Zustände S, S_1, S_2 gilt: Wenn $S \mapsto^* S_1$ und $S \mapsto^* S_2$, dann sind S_1 und S_2 zusammenführbar.

Definition 4.7 Ein CHR-Programm heißt *lokal konfluent*, wenn für alle Zustände S, S_1, S_2 gilt: Wenn $S \mapsto S_1$ und $S \mapsto S_2$, dann sind S_1 und S_2 zusammenführbar.

Um die lokale Konfluenz von CHR-Programmen feststellen zu können, müßte man also alle möglichen Paare von Zuständen, die aus einem gemeinsamen Zustand durch unterschiedliche Reduktionsschritte hervorgehen, auf ihre Zusammenführbarkeit testen. Es gibt aber im allgemeinen unendlich viele solcher Paare. Also liefert dieses naive Verfahren keinen terminierenden Algorithmus. Im folgenden wird gezeigt, daß es ausreicht, nur endlich viele Paare, *kritische Paare*, zu betrachten, um die Konfluenz von terminierenden CHR-Programmen zu überprüfen.

Ein kritisches Paar bezeichnet die zwei allgemeinsten Folgezustände, die durch die Anwendung zweier CHR-Regeln bzw. zweier Varianten einer Regel auf einen Ausgangszustand entstehen können. Das Anwenden zweier Regeln auf denselben Ausgangszustand kommt formal durch die Überlappung ihrer Kopfatome durch Unifikation zum Ausdruck. Durch die Forderung nach einer beliebigen Anzahl unifizierbarer Kopfatome wird die Tatsache berücksichtigt, daß sich die Regelköpfe in einer beliebigen Anzahl von Atomen überlappen können. Für zwei Regeln bzw. zwei Varianten einer Regel sind also u.U. mehrere kritische Paare möglich. Wie ein kritisches Paar definiert ist, hängt natürlich von der Art der Regeln ab, die angewendet werden. In diesem Abschnitt wird keine formale Definition der kritischen Paare angegeben. Die Konfluenz der SiR-Programme, d.h. aus lediglich Simplifikationsregeln bestehenden Programme, wird im nächsten Abschnitt behandelt. Wenn die Propagationsregel in die Betrachtungen miteinbezogen wird, verkomplizieren sich die Verhältnisse beträchtlich [Abd97]. Im Abschnitt 4.3 werden dann diese Probleme diskutiert und abschließend wird eine entscheidbare Charakterisierung der konfluenten terminierenden CHR-Programme präsentiert, nachdem eine neue operationale Semantik, die eine sehr gute Annäherung an die tatsächliche Implementierung von CHR ist, vorgestellt wird.

Definition 4.8 Ein kritisches Paar (S_1, S_2) ist *zusammenführbar*, falls S_1 und S_2 zusammenführbar sind.

Die folgende Aussage stellt einen Zusammenhang zwischen lokaler Konfluenz und der Zusammenführbarkeit der kritischen Paare her. Diese Aussage wird auch der *Satz der lokalen Konfluenz* genannt und ist die zentrale Aussage dieses Abschnittes. Um diesen Satz zu beweisen, muß man mehrere Begriffe definieren und Lemmata beweisen. Dies wird in Abschnitt 4.3.3 geschehen.

Satz 4.9 Ein CHR-Programm ist genau dann lokal konfluent, wenn alle kritischen Paare des Programmes zusammenführbar sind.

Der nächste Satz ist ein Analogon zu Newmans Lemma [New42] (oder Diamantenlemma) für Reduktionssysteme. Der Beweis läßt sich direkt auf CHR übertragen. Der Satz wird durch Noethersche Induktion bewiesen.

Definition 4.10 Eine zweistellige Relation $\sqsupseteq$ auf einer Menge $\mathcal{S}$ heißt eine *Halbordnung*, falls für alle $S, T, Z \in \mathcal{S}$ gilt

i) $S \sqsupseteq S$ (Reflexivität)

ii) Wenn $S \sqsupseteq T$ und $T \sqsupseteq S$, dann $S = T$ (Antisymmetrie)

iii) Wenn $S \sqsupseteq Z$ und $Z \sqsupseteq T$, dann $S \sqsupseteq T$ (Transitivität)

Die strikte Relation $\sqsupset$ von $\sqsupseteq$ ist durch „$S \sqsupset T$ gdw $S \sqsupseteq T$ und $S \neq T$“ gegeben.

Definition 4.11 Eine Halbordnung $\sqsupseteq$ heißt *Noethersch*, wenn es keine unendliche absteigende Kette $S_0 \sqsupset S_1 \sqsupset \ldots$ gibt.

Definition 4.12 Seien $\sqsupseteq$ eine Halbordnung auf $\mathcal{S}$ und E ein Prädikat auf $\mathcal{S}$. Dann heißt E *$\sqsupseteq$-vollständig*, falls für alle $S \in \mathcal{S}$ gilt: Gilt $E(T)$ für alle T mit $S \sqsupset T$, so gilt $E(S)$.

Prinzip der Noetherschen Induktion: Das Prinzip der Noetherschen Induktion lautet: Ist $\sqsupseteq$ Noethersch und E $\sqsupseteq$-vollständig auf $\mathcal{S}$, so gilt $E(S)$ für alle $S \in \mathcal{S}$

Satz 4.13 Ein terminierendes CHR-Programm P ist genau dann konfluent, wenn P lokal konfluent ist.
Beweis: Ist P konfluent, so ist P lokal konfluent. Sei jetzt P lokal konfluent und terminierend. Man zeigt durch Noethersche Induktion, daß P konfluent ist.

Sei

$$\begin{array}{rll} E(S) & \text{gdw} & \text{für alle } S_1, T_1 \text{ wenn } S \mapsto^* S_1 \text{ und } S \mapsto^* T_1 \text{ dann } S_1 \downarrow T_1, \\ S \sqsupseteq T & \text{gdw} & S \mapsto^* T. \end{array}$$

Dann ist $\sqsupseteq$ eine Noethersche Halbordnung, da P terminierend ist. Zu zeigen ist, daß E $\sqsupseteq$-vollständig ist. Sei $S \in \mathcal{S}$, und für alle T mit $S \sqsupset T$ gelte $E(T)$. Zu zeigen ist $E(S)$.
Sei $S \mapsto^* S_1$ und $S \mapsto^* T_1$. Zu zeigen ist $S_1 \downarrow T_1$.

a) Gilt $S_1 = S$ oder $T_1 = S$, so gilt trivialerweise $S_1 \downarrow T_1$.

b) Gilt $S_1 \neq S$ und $T_1 \neq S$, so gibt es S_1', T_1' mit $S \mapsto S_1' \mapsto^* S_1$ und $S \mapsto T_1' \mapsto^* T_1$. Da P lokal konfluent ist, gibt es ein S_0 mit $S_1' \mapsto^* S_0$ und $T_1' \mapsto^* S_0$. Nach Induktionsvoraussetzung gilt $E(S_1')$, da $S \sqsupset S_1'$ ist. Das heißt, wenn $S_1' \mapsto^* S_0$ und $S_1' \mapsto^* S_1$, dann gibt es ein S_0', so daß $S_0 \mapsto^* S_0'$ und $S_1 \mapsto^* S_0'$. Es gilt auch $E(T_1')$, da $S \sqsupset T_1'$. Also wenn $T_1' \mapsto^* S_0'$ und $T_1' \mapsto^* T_1$, dann $S_0' \downarrow T_1$. Also gilt $S_1 \downarrow T_1$ und damit auch $E(S)$.

$$\begin{tikzcd}
 & & S \arrow[dl] \arrow[dr] & & \\
 & S_1' \arrow[dl, "*"'] \arrow[dr, "*"] & & T_1' \arrow[dl, "*"'] \arrow[dr, "*"] & \\
S_1 \arrow[dr, "*"] & & S_0 \arrow[dl, "*"'] & & T_1 \arrow[ddll, "*"'] \\
 & S_0' \arrow[dr, "*"] & & & \\
 & & S_2 & &
\end{tikzcd}$$

□

Ohne die Voraussetzung, daß P terminierend ist, ist Satz 4.13 falsch, wie das folgende Beispiel zeigt:

Beispiel 4.14 Gegeben sei folgendes CHR-Programm P:

```
p ⇔ q.
p ⇔ r.
r ⇔ p.
r ⇔ s.
```

Es gilt folgendes:

q $\leftarrowtail$ p $\mapsto$ r: wegen r $\mapsto^*$ q gilt q $\downarrow$ r
p $\leftarrowtail$ r $\mapsto$ s: wegen p $\mapsto^*$ s gilt p $\downarrow$ s

P ist lokal konfluent. Es gilt aber auch s $^*\!\leftarrowtail$ p $\mapsto^*$ q, aber nicht q $\downarrow$ s. Also ist P nicht konfluent.

Aus den letzten beiden Sätzen folgt unmittelbar das folgende Korollar:

Korollar 4.15 Ein terminierendes CHR-Programm ist genau dann konfluent, wenn seine kritischen Paare zusammenführbar sind.

Korollar 4.15 bietet eine entscheidbare Charakterisierung der Klasse der konfluenten in der Klasse der terminierenden CHR-Programme, wie das folgende Lemma zeigt:

Lemma 4.16 Zu jedem terminierenden CHR-Programm P ist es entscheidbar, ob P konfluent ist.

Beweis: Die Menge $K(P)$ der kritischen Paare zu P ist endlich, sei etwa $K(P) = \{(S_i, T_i) \mid i = 1, \ldots, n\}$. Es ist zu testen, ob $S_i \downarrow T_i$ für alle i gilt. Dies leistet der folgende Algorithmus:

Bilde zu jedem Paar (S_i, T_i) beliebige Ableitungen $S_i \mapsto^* S_i'$ und $T_i \mapsto^* T_i'$, so daß S_i' und T_i' Endzustände sind. Gilt, daß S_i' und T_i' Varianten sind, dann antworte „P ist konfluent“, sonst antworte „P ist nicht konfluent“.

Dieser Algorithmus ist korrekt. Sind S_i' und T_i' Varianten für alle i, so sind alle kritischen Paare zusammenführbar und P ist nach Satz 4.15 konfluent. Gibt es ein i, so daß S_i' und T_i' keine Varianten sind, so gibt es einen Zustand S mit $S \mapsto S_i \mapsto^* S_i'$ und $S \mapsto T_i \mapsto^* T_i'$ und nicht $S_i' \downarrow T_i'$, da weder S_i' noch T_i' Nachfolger haben. Also ist P nicht konfluent. □

4.2 Konfluenz von SiR-Programmen

SiR-Programmen sind Programme, die aus lediglich Simplifikationsregeln bestehen. Wenn man die operationale Semantik von SiR-Programmen mit Termersetzungssystemen vergleicht, dann sieht man leicht, daß die Grundidee beider Ansätze sehr ähnlich ist: Man hat ein gerichtetes System, das eine Reduktionsrelation auf der Menge der Terme (bei den Termersetzungssystemen) und auf der Menge der Zustände (bei SiR) definiert, und man wendet ausgehend von einem Ausdruck so lange Ableitungsschritte an, bis keine Regel mehr anwendbar ist. Ein Ableitungsschritt hat dann als Konsequenz, daß ein Term bei Termersetzungssystemen oder ein Constraint bei SiR durch ein anderes ersetzt wird (Einen ausführlichen Vergleich bietet Kapitel 8). Deswegen war der erste Schritt bei der Untersuchung der Konfluenz von CHR, nur die SiR-Programmen zu betrachten [AFM96b, Meu96, AFM98], die ähnliche Wirkung auf Zustände wie die bedingten Termersetzungsregeln auf Terme haben. Eine unmittelbare Übertragung der Konfluenzergebnisse von CTRS war aber leider nicht möglich, wie in Abschnitt 8.1 erklärt ist.

Die Bezeichnung kritisches Paar rührt also vom analogen Konstrukt in Termersetzungssystemen her. Das „Paar“, auf das diese Bezeichnung beruht, besteht aus zwei Zuständen, die entstehen, wenn zwei SiR-Regeln auf einen „kritischen Zustand“ angewendet werden. Diesen kritischen Zustand kann man wie bei Termersetzungssystemen durch sogenannte „Superposition“ der SiR-Regeln berechnen. Um zwei SiR-Regeln R_1, R_2 zu „überlagern“, hält man Konjunkte des Kopfes einer Regel R_1 fest. Für diese Konjunkte prüft man, ob sie mit einigen Konjunkten des Kopfes von Regel R_2 unifizierbar sind. Der Zielspeicher des kritischen Zustandes bleibt leer, sein benutzerdefinierte Speicher besteht aus der Konjunktion des Kopfes der Regel R_1 mit dem Rest des Kopfes der Regel R_2, d.h. den anderen Konjunkten, die bei dem Unifizierbarkeitstest nicht miteinbezogen wurden. Sein vordefinierter Speicher besteht aus der Konjunktion der Wächter und den Gleichheitsconstraints der unifizierbaren Konjunkte. Die vierte Komponente des Zustandes ist die Liste der (globalen) Variablen, die in den Köpfen von R_1 und R_2 vorkommen. Darüber braucht man bei den Termersetzungssystemen nicht Buch zu führen, da dort alle Variablen gleich behandelt werden.

Definition 4.17 Wenn ein oder mehrere Constraints $H_{i_1}, \ldots, H_{i_k}$ $(1 \leq k \leq n)$ des Kopfes einer Simplifikationsregel $R_1 @ H_1, \ldots, H_n \Leftrightarrow G \mid B$ mit einem oder mehreren Constraints $H'_{j_1}, \ldots, H'_{j_k}$ des Kopfes einer anderen oder derselben[1] Simplifikationsregel $R_2 @ H'_1, \ldots, H'_m \Leftrightarrow G' \mid B'$ unifizieren, dann heißt der Zustand

$$\langle \top, H_1 \wedge \ldots \wedge H_n \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}, G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}, \mathcal{V} \rangle$$

[1]gemeint ist eine Variante mit frischen Variablen

kritischer Zustand für die beiden Regeln. Hierbei sind $\{i_1, \ldots, i_n\}$ und $\{j_1, \ldots, j_m\}$ Permutationen von $\{1, \ldots, n\}$ bzw. $\{1, \ldots, m\}$, und $\mathcal{V}$ ist die Menge der Variablen aus $H_1, \ldots, H_n, H'_1, \ldots, H'_m$.

Nach der Konstruktion des kritischen Zustandes für die SiR-Regeln R_1, R_2 sind sowohl R_1 als auch R_2 auf diesem Zustand anwendbar. Durch die beiden **Simplify**-Schritte erhält man dann das kritische Paar, wie man es auch in bedingten Termersetzungssystemen [DOS88] kennt.

Definition 4.18 Sei $\langle\top, H_1 \wedge \ldots \wedge H_n \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}, G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}, \mathcal{V}\rangle$ ein kritischer Zustand für die Simplifikationsregeln $H_1, \ldots, H_n \Leftrightarrow G \mid B$ und $H'_1, \ldots, H'_m \Leftrightarrow G' \mid B'$. Durch die Anwendung der beiden **Simplify**-Schritte auf dem kritischen Zustand erhält man ein *kritisches Paar* von Zuständen für die beiden Regeln:

$$(\langle B, H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}, \bar{G}, \mathcal{V}\rangle, \langle B', H_{i_{k+1}} \wedge \ldots \wedge H_{i_n}, \bar{G}, \mathcal{V}\rangle)$$

Hierbei ist $\bar{G} = G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}$.

Dadurch, daß Regeln auch mit sich selber verglichen werden, entstehen viele kritische Paare, die keine Information tragen:

Definition 4.19 Ein kritisches Paar (S_1, S_2) heißt *trivial*, wenn S_1 und S_2 Varianten sind.

Beispiel 4.20 Sei P das CHR-Programm für das Constraintsymbol `max`:

```
r1 @ max(X,Y,Z) ⇔ X ≤ Y | Y = Z.
r2 @ max(X,Y,Z) ⇔ Y ≤ X | X = Z.
```

Die beiden Regeln haben drei kritische Paare, von denen jedoch zwei trivial sind. Der kritische Zustand, aus dem ein nichttriviales kritisches Paar entsteht, ist

$$\langle\top, \mathtt{max}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}), \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}, [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]\rangle$$

und das kritische Paar

$$(\langle\mathtt{Y} = \mathtt{Z}, \top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}, [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]\rangle, \langle\mathtt{X} = \mathtt{Z}, \top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}, [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]\rangle)$$

Es ist leicht zu sehen, daß beide Zustände des kritischen Paares zu einem syntaktisch gleichen Zustand

$$\langle\top, \top, \mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z}, [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]\rangle$$

durch die Reduktionsregel **Solve** zusammengeführt werden können, vorausgesetzt die Normalisierungsfunktion die vordefinierten Constraints $\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}$ zu $\mathtt{X} = \mathtt{Y}$ vereinfacht.

Beispiel 4.21 Gegeben sei das CHR-Programm aus Beispiel 4.1, wobei `durchschnitt/3` ein vordefiniertes Constraintsymbol ist.

```
nicht_leer @ X::[]   ⇔  false.

binden @ X::[Y]  ⇔  X=Y.

durchschnitt @ X::L1 ∧ X::L2  ⇔
       durchschnitt(L1,L2,L3) ∧ X::L3.
```

Das folgende kritische Paar stammt aus dem Unifizieren des Kopfes der Regel `binden` mit dem ersten Constraintatom des Kopfes der Regel `durchschnitt`:

$$\langle\mathtt{X} = \mathtt{Y}, \mathtt{X} :: \mathtt{L2}, \mathtt{true}, [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle, \langle\mathtt{X} :: \mathtt{L3} \wedge \mathtt{durchschnitt}([\mathtt{Y}], \mathtt{L2}, \mathtt{L3}), \top, \mathtt{true}, [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle)$$

Dieses kritische Paar ist nicht zusammenführbar, da sein erster Zustand folgende Ableitung hat:

$$\begin{array}{ll} & \langle \mathtt{X} = \mathtt{Y}, \mathtt{X} :: \mathtt{L2}, \mathtt{true}, [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle \\ \mapsto_{\mathbf{Solve}} & \langle \top, \mathtt{X} :: \mathtt{L2}, \mathtt{X} = \mathtt{Y}, [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle \end{array}$$

Der zweite Zustand des kritischen Paares reduziert sich zu dem gleichen Endzustand, egal welches Constraint als erstes vom Zielspeicher transportiert wird:

$$\begin{array}{ll} & \langle \mathtt{X} :: \mathtt{L3} \wedge \mathtt{durchschnitt}([\mathtt{Y}], \mathtt{L2}, \mathtt{L3}), \top, \mathtt{true}, [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle \\ \mapsto_{\mathbf{Introduce}} & \langle \mathtt{durchschnitt}([\mathtt{Y}], \mathtt{L2}, \mathtt{L3}), \mathtt{X} :: \mathtt{L3}, \mathtt{true}, [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle \\ \mapsto_{\mathbf{Introduce}} & \langle \top, \mathtt{X} :: \mathtt{L3}, \mathtt{durchschnitt}([\mathtt{Y}], \mathtt{L2}, \mathtt{L3}), [\mathtt{X}, \mathtt{Y}, \mathtt{L2}, \mathtt{L3}]\rangle \end{array}$$

Der Satz der lokalen Konfluenz (Satz 4.9) besagt, daß die Zusammenführbarkeit kritischer Paare eine Garantie ist, daß trotz unterschiedlicher Berechnungsschritte an einem Verzweigungspunkt ein „gemeinsamer“ Zustand erreicht werden kann. Die Idee des Beweises ist, die Zusammenführbarkeit der kritischen Paare im allgemeinsten Kontext (S_1, S_2) auf einen aktuellen Kontext (S_1', S_2') zu übertragen (Abbildung 4.2). S_1' (S_2') geht aus dem Zustand S_1 (S_2) hervor, indem in die Speicher in S_1 (S_2) zusätzlich weitere Constraints eingefügt wurden (Abbildung 4.3). Die Idee des Beweises ist es also, zu zeigen, daß die Zusammenführbarkeit der kritischen Paare in gewissem Sinn ein hermetisches Modul darstellt, das sich in jedem aktuellen Kontext einklinken läßt. Das heißt, daß die Identität des aktuellen Zustandes keinen Einfluß auf die Durchführbarkeit der bei der Zusammenführbarkeit angewendeten Schritte hat, solange der aktuelle Zustand eine Ausprägung des allgemeinsten Zustandes ist. Diese Eigenschaft kann mit „Kontextmonotonie“ bezeichnet werden. Die Kontextmonotonie garantiert, daß die Durchführbarkeit von Berechnungsschritten nicht beeinträchtigt wird, wenn in die Speicher zusätzliche harmlose Constraints eingefügt werden. Für SiR-Programme wird der Beweis des Satzes der lokalen Konfluenz nicht durchgeführt. Er ist in [AFM96b] zu finden. Die Idee des Beweises beruht auf der Kontextmonotonie. Diese Eigenschaft wird in Lemma 4.23 gezeigt.

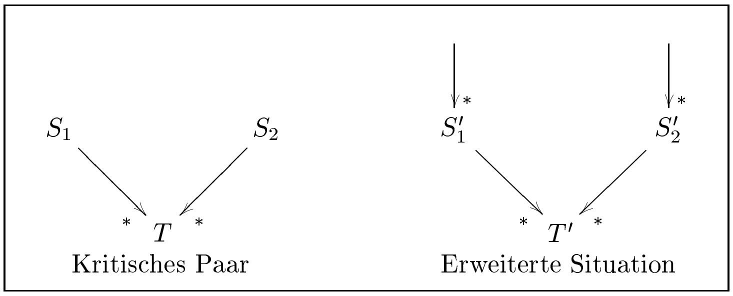

Abbildung 4.2: Kontextmonotonie

Definition 4.22 Ein Paar (C_1, C_2) von Constraints heißt *abgedeckt* von einer Liste $\mathcal{V}$ von Variablen, wenn alle Variablen, die sowohl in C_1 als auch in C_2 auftreten, in $\mathcal{V}$ vorkommen.

Lemma 4.23 Sei $(C_1 \wedge C_2 \wedge C_3, G \wedge C_U \wedge C_B)$ von $\mathcal{V}$ abgedeckt. Für jeden Berechnungsschritt $\langle G, C_U, C_B, \mathcal{V}\rangle \mapsto \langle G', C_U', C_B', \mathcal{V}\rangle$ gilt, daß

$$\langle G \wedge C_1, C_U \wedge C_2, C_B \wedge C_3, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G' \wedge C_1, C_U' \wedge C_2, C_B' \wedge C_3, \mathcal{V}\rangle).$$

Beweis: durch Fallunterscheidung nach dem angewendeten Berechnungsschritt:

- **Introduce:** Dann ist G von der Form $H \wedge G_1$ und der Berechnungsschritt

  $$\langle H \wedge G_1, C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G_1, H \wedge C_U, C_B, \mathcal{V}\rangle).$$

  Nach Hinzunahme der Constraints C_1, C_2 und C_3 zu den Speichern ist folgender **Introduce**-Schritt möglich

  $$\langle H \wedge G_1 \wedge C_1, C_U \wedge C_2, C_B \wedge C_3, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G_1 \wedge C_1, H \wedge C_U \wedge C_2, C_B \wedge C_3, \mathcal{V}\rangle).$$

$$\begin{tikzcd}
\langle B_1, \top, G, \mathcal{V}\rangle \arrow[dr, "*"] & & \langle B_2, \top, G, \mathcal{V}\rangle \arrow[dl, "*"'] \\
& T &
\end{tikzcd}$$

$$\begin{tikzcd}
\langle B_1 \wedge Gs, C_U, G \wedge C_B, \mathcal{V}\rangle \arrow[dr, "*"'] & & \langle B_2 \wedge Gs, C_U, G \wedge C_B, \mathcal{V}\rangle \arrow[dl, "*"] \\
& T' &
\end{tikzcd}$$

Abbildung 4.3: Erweiterte Zustände

- **Solve:** Dann ist G_s von der Form $C \wedge G_1$ und der Berechnungsschritt:

  $$\langle C \wedge G_1, C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G_1, C_U, C_B \wedge C, \mathcal{V}\rangle).$$

  Wenn die Constraints C_1, C_2 und C_3 zu den Speichern hinzugefügt werden, kann folgender **Solve**-Schritt durchgeführt werden:

  $$\langle C \wedge G_1 \wedge C_1, C_U \wedge C_2, C_B \wedge C_3, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G_1 \wedge C_1, C_U \wedge C_2, C_B \wedge C \wedge C_3, \mathcal{V}\rangle).$$

  Jetzt bleibt also nur noch zu zeigen, daß $\mathcal{N}(\langle G_1 \wedge C_1, C_U \wedge C_2, C_B \wedge C \wedge C_3, \mathcal{V}\rangle) = \mathcal{N}(\langle G' \wedge C_1, C'_U \wedge C_2, C'_B \wedge C_3, \mathcal{V}\rangle)$. Dies folgt aus der Anwendung von Lemma 4.24.

- **Simplify:** Dann ist C_U von der Form $H \wedge C_{U_1}$ und der Berechnungsschritt ist von der folgenden Gestalt:

  $$\langle G, H \wedge C_{U_1}, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, C_{U_1}, C_B \wedge E, \mathcal{V}\rangle).$$

  Der gleiche **Simplify**-Schritt ist auch nach dem Hinzufügen der drei Constraints C_1, C_2 und C_3 anwendbar, weil die logische Konsequenz monoton ist:

  $$\langle G \wedge C_1, H \wedge C_{U_1} \wedge C_2, C_B \wedge C_3, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge C_1 \wedge B, C_{U_1} \wedge C_2, C_B \wedge E \wedge C_3, \mathcal{V}\rangle).$$

  Die Behauptung folgt jetzt mit Lemma 4.24 und der Abdeckungsbedingung. □

Lemma 4.24 Wenn (C, C_B) von $\mathcal{V}$ abgedeckt ist und C ein vordefiniertes Constraint, so folgt aus

$$\mathcal{N}(\langle G, C_U, C_B, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{V}\rangle,$$

daß $\mathcal{N}(\langle G \wedge C_1, C_U \wedge C_2, C_B \wedge C \wedge C_3, \mathcal{V}\rangle) = \mathcal{N}(\langle G' \wedge C_1, C'_U \wedge C_2, C'_B \wedge C_3, \mathcal{V}\rangle)$.

Dieses Lemma besagt, daß es egal ist, ob man Constraints vor oder nach der Anwendung der Normalisierungsfunktion zum vordefinierten Speicher hinzufügt. Wie wichtig die Abdeckungsbedingung ist, wird in Abschnitt 4.3.3 erläutert. Es wird hier auf einen Beweis des Lemmas verzichtet, da es in Abschnitt 4.3.3 für den allgemeineren Fall durchgeführt wird.

4.3 Konfluenz von CHR-Programmen

Die im Abschnitt 3.2 vorgestellte operationale Semantik für SiR-Programme spiegelt genau die Implementierung wider, die in [FB95] vorgestellt wurde. Die operationale Semantik der Propagationsregeln bedarf aber noch eingehender Untersuchungen. Die Schwierigkeiten gründen in der

Eigenschaft der Propagationsregel, daß ihre Anwendung einer Bedingung unterworfen ist, der Anwendungsbedingung. Ursache für die Existenz der Anwendungsbedingung ist die Tatsache, daß sonst ein und dieselbe Propagationsregel unendlich oft auf dieselbe Menge von Constraints angewendet werden könnte. In [Frü95] wird die tatsächliche Definition der Anwendungsbedingung in der dort vorgestellten operationalen Semantik nicht beschrieben. Es wird nur beschrieben, wie die Implementierung von CHR sicherstellt, daß eine Propagationsregel nicht unendlich oft hintereinander auf dieselben Constraints angewendet werden kann. Diese triviale Nichtterminierung wird in der Implementierung verhindert, indem jede Propagationsregel nur einmal auf eine Menge von Constraints angewendet werden kann. Hierbei haben syntaktisch gleiche Constraints eine eigene Identität.

Die Untersuchung der Anwendungsbedingung im Zusammenhang mit Konfluenz ist aus zwei Gründen sehr wichtig. Erstens hätte das Verzichten auf die Anwendungsbedingung bei der Konfluenzuntersuchung die Konsequenz, daß ein CHR-Programm, sobald es eine Propagationsregel beinhaltet, nicht mehr terminiert. Dabei taucht ein praktisches Problem auf: Die Zusammenführbarkeit kritischer Paare ist nicht mehr entscheidbar, da nicht mehr alle Berechnungssequenzen in endlicher Zeit verglichen werden können. Damit scheint diesem Weg keine Zukunft gegeben zu sein, da die Entscheidbarkeit wesentlich für die Automatisierung und sinnvolle Verwendung eines Konfluenzkriteriums ist. Der zweite Grund ist, daß eine Formalisierung der Anwendungsbedingung, die in der Implementierung verwendet wird, für die Konfluenzergebnisse nicht ausreicht. Dies wird durch das folgende Beispiel illustriert.

Zur besseren Lesbarkeit werden in diesem Abschnitt häufig nur die benutzerdefinierten Speicher statt der ganzen Zustände angegeben, wenn nur benutzerdefinierte Constraints im CHR-Programm vorkommen. Nach Anwendung einer Propagations-, Simplifikations- oder Simpagationsregel werden implizit soviele **Introduce**-Schritte ausgeführt wie nötig, um alle benutzerdefinierten Constraints vom Zielspeicher in den benutzerdefinierten Speicher zu transportieren und somit kann direkt **Simplify**, **Propagate**, oder **Simpagate** durchgeführt werden.

Beispiel 4.25 Gegeben sei folgendes CHR-Programm:

r1 @ p $\Rightarrow$ q.
r2 @ r $\wedge$ q $\Leftrightarrow$ true.
r3 @ r $\wedge$ p $\wedge$ q $\Leftrightarrow$ s.
r4 @ s $\Leftrightarrow$ p $\wedge$ q.
r5 @ q $\backslash$ q $\Leftrightarrow$ true.

Der kritische Zustand r $\wedge$ p $\wedge$ q, der aus der zweiten und dritten Regel entsteht, ergibt das kritische Paar (p,s). Wenn man davon ausgeht, daß der kritische Zustand keine Information über die schon angewendeten Propagationsregeln hat, dann ist das kritische Paar (p,s) zusammenführbar:

p $\mapsto_{r1}$ p $\wedge$ q.
s $\mapsto_{r4}$ p $\wedge$ q $\mapsto_{r1}$ p $\wedge$ q $\wedge$ q $\mapsto_{r5}$ p $\wedge$ q.

Wenn man aber von einem Zustand ausgeht, der mehr Informationen als der kritische Zustand enthält, dann kann man nicht sicher sein, daß das Paar, das durch die jeweilige Anwendung der Regeln r2 und r4 entsteht, zusammenführbar bleibt. Eine solche Situation entsteht, wenn man von dem Ziel r $\wedge$ p ausgeht und darauf die Propagationsregel r1 anwendet. Dann erhält man den Zustand r $\wedge$ p $\wedge$ q mit der zusätzlichen Information, daß die Propagationsregel auf das Constraint p angewendet wurde. Dann kann die Propagationsregel als letzter Schritt nicht mehr auf p angewendet werden (Abbildung 4.4). Die Zustände sind dann nicht zusammenführbar. Das Programm ist also nicht konfluent, obwohl alle kritischen Paare zusammenführbar sind. Ursache dafür ist, daß die Kontextmonotonie unter dieser naiven operationalen Semantik nicht hält und daß die Definition der kritischen Paare die Propagationsregeln nicht berücksichtigen.

Im folgenden werden unterschiedliche Möglichkeiten, die Anwendungsbedingung zu definieren, in Bezug auf ihre Auswirkungen auf Korollar 4.15 vorgestellt. Diese Bedingungen wurden in [Meu96] untersucht, sie haben aber leider nicht zu einem zufriedenstellenden Ergebnis geführt:

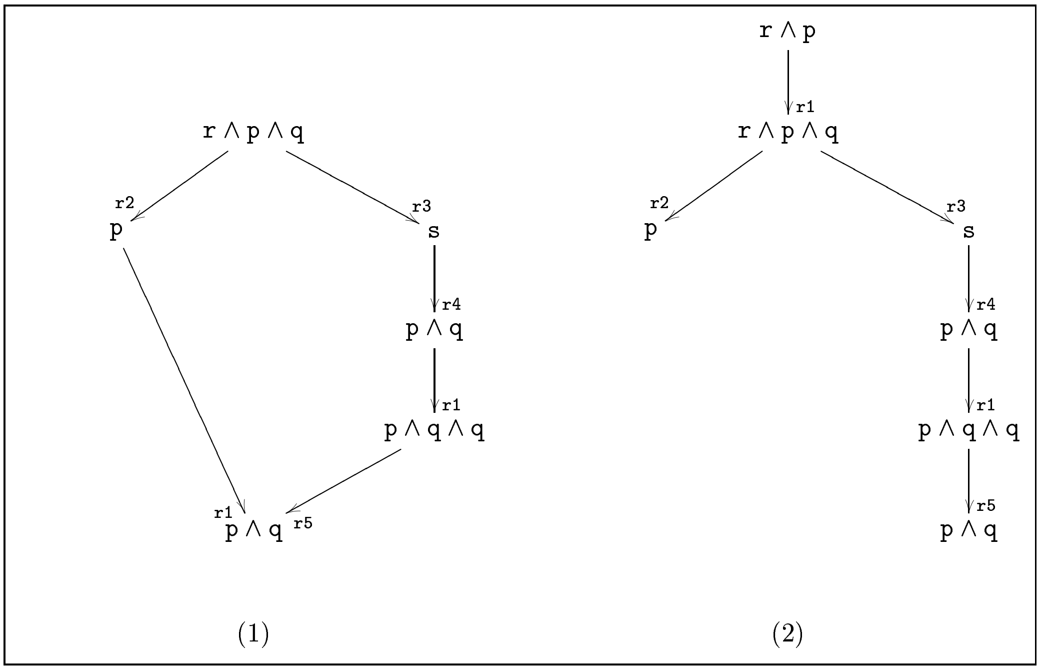

Abbildung 4.4: Verletzung der Konfluenz

Die Zustandsbedingung verhindert, daß eine Propagationsregel angewendet wird, wenn diese Regel bereits in diesem Zustand vorher angewendet wurde.

Die Redundanzbedingung sagt aus, daß eine Propagationsregel auf einen Zustand nur angewendet werden kann, wenn sie dem Zustand neue Information hinzufügt, entweder in Form von neuen benutzerdefinierten Constraints oder in Form von stärkeren vordefinierten Constraints.

Die Instanzbedingung fordert, daß die Instanz einer Propagationsregel nur dann angewendet werden darf, wenn sie noch nie angewendet worden ist.

Um über das Feuern der Propagationsregeln buchzuführen, verlangt die Zustandsbedingung ein „Gedächtnis“ als fünfte Komponente der Zustände. Dieses Gedächtnis enthält Informationen über die Zustände der bisherigen Berechnungssequenz und die Propagationsregeln, die darauf angewendet wurden. Bevor eine Propagationsregel angewendet wird, wird dann im Gedächtnis überprüft, ob diese Regel auf diesem Zustand bereits angewendet wurde. Ein Problem bei der Zustandsbedingung ist, daß die Terminierung bei keinem CHR-Programm mit Propagationsregeln sichergestellt wird.

Beispiel 4.26 Gegeben sei die Progagationsregel aus Beispiel 3.1

$\texttt{transitivitaet @ } X \le Y \wedge Y \le Z \Rightarrow X \le Z.$

Beginnend mit dem Zustand $X \le Y \wedge Y \le Z$ gibt es folgende Berechnungssequenz:

$$
\begin{array}{l}
\quad X \le Y \wedge Y \le Z \\
\mapsto X \le Y \wedge Y \le Z \wedge X \le Z \\
\mapsto X \le Y \wedge Y \le Z \wedge X \le Z \wedge X \le Z \\
\mapsto \ldots
\end{array}
$$

Man kann immer wieder die Propagationsregel auf den resultierenden Zustand anwenden, da die Zustände voneinander unterschiedlich sind. Dadurch werden immer neue Zustände erzeugt und man erhält eine unendliche Ableitung, d.h. das Programm ist nicht terminierend.

Die Redundanzbedingung hat den Nachteil, daß sie nicht bei allen sinnvollen CHR-Programmen die triviale Nichtterminierung verhindern kann, wie das folgende Beispiel zeigt:

Beispiel 4.27 Gegeben sei die Reflexivitätsregel und Transitivitätsregel aus Beispiel 3.1:

$\texttt{reflexivitaet @ X} \leq \texttt{X} \Leftrightarrow \textit{true}.$
$\texttt{transitivitaet @ X} \leq \texttt{Y} \wedge \texttt{Y} \leq \texttt{Z} \Rightarrow \texttt{X} \leq \texttt{Z}.$

Man kann folgende unendliche Berechnungssequenz erzeugen:

$$
\begin{array}{l}
 \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} \leq \texttt{X} \\
\mapsto \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} \leq \texttt{X} \wedge \texttt{X} \leq \texttt{X} \\
\mapsto \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} \leq \texttt{X} \\
\mapsto \ldots
\end{array}
$$

Die Instanzbedingung verhindert die triviale Nichtterminierung und ist auch der in [Frü95] für die Implementierung von CHR vorgeschlagenen Verhinderung der Nichtterminierung sehr ähnlich. Dort wird eine speziellere Anwendungsbedingung, die *Instanzbedingung mit Indizierung*, verwendet. Diese fordert, daß die Instanz einer Propagationsregel nur dann angewendet werden darf, wenn sie noch nie angewendet worden ist, wobei die Indizierung die Unterscheidung syntaktisch gleicher Constraints ermöglicht. Das bedeutet insbesondere, daß jede Propagationsregel nur einmal auf dieselben Constraints angewendet werden darf.

Die Instanzbedingung verlangt wie die Zustandsbedingung ein Gedächtnis als fünfte Komponente der Zustände. Das Gedächtnis verfügt über die Information „welche Propagationsregeln mit welchen benutzerdefinierten Constraints bereits gefeuert haben“. Dieses Gedächtnis ist in einem gewissen Sinne antimonoton (im Sinne, je größer das Gedächtnis ist, desto weniger Reduktionsschritte sind erlaubt). Die Instanzbedingung ist also für die Untersuchung der Konfluenz unzufriedenstellend, da sie eine wichtige Eigenschaft für das Konfluenzkriterium, die Kontextmonotonie, verletzt. Diese Eigenschaft besagt, daß Berechnungssequenzen in einen spezielleren Kontext übertragen werden können, d.h. daß die gleichen Berechnungsschritte anwendbar sind, auch wenn man von Zuständen ausgeht, die mehr Informationen in den Speichern haben, also mehr Constraints in den Speichern oder eben ein größeres Gedächtnis.

Aus den oben vorgestellten Anwendungsbedingungen sieht man, daß es wichtig ist, eine neue Formulierung für die Anwendungsbedingung zu finden, die einerseits die Kontextmonotonie von CHR erhält, andererseits aber die Terminierung von sinnvollen CHR-Programmen mit Propagationsregeln garantiert.

Eine Anwendungsbedingung, die dies leistet, ist die sogenannte Token-Bedingung [Abd97]. Die Idee ist, einen Tokenspeicher in dem Zustand mitzuführen, der Informationen über die Propagationsregeln enthält, die noch feuern können. Im Vergleich zu der Instanzbedingung, die auf einer Repräsentation der verbotenen, weil schon durchgeführten, Regelanwendungen basiert, repräsentiert der neue Ansatz die noch erlaubten Regelanwendungen. Auf den ersten Blick scheint eine triviale Dualität zwischen den beiden Darstellungen zu bestehen. Der entscheidende Punkt ist aber, daß eine Regelanwendung im Verlauf einer Ableitung mehrmals zwischen dem Status erlaubt und verboten wechseln kann. Dieses Phänomen ist mit dem Token-Ansatz recht natürlich zu beschreiben, aber es könnte auch mit „Antitoken“ für verbotene Anwendungen modelliert werden.

Bevor wir ein entscheidbares, notwendiges und hinreichendes Kriterium für die Konfluenz terminierender CHR-Programme angeben, in dem die Definition von kritischen Paaren verallgemeinert wird, werden wir eine zweite Formalisierung der operationalen Semantik vorstellen, die eine sehr gute Annäherung an die tatsächliche Implementierung ist.

4.3.1 Zweite Formalisierung der operationalen Semantik

Bei der ersten Formalisierung der operationalen Semantik von CHR im Abschnitt 3.2, wurde der Zustand als Quadrupel definiert. Im folgenden wird der Zustand um eine fünfte Komponente erweitert, um die Anwendungsbedingung der Propagationsregeln mittels eines Gedächtnisses in Griff zu bekommen.

Definition 4.28 Ein *Zustand* ist ein Tupel

$$\langle G, C_U, C_B, C_T, \mathcal{V} \rangle,$$

wobei G ein Ziel, C_U eine Konjunktion von benutzerdefinierten Constraints, C_B eine Konjunktion von vordefinierten Constraints, C_T Liste von Tokens von der Form $R@C$ (Tokenspeicher), wobei C eine Konjunktion benutzerdefinierter Constraints ist und R ein Regelname einer Propagationsregel, und $\mathcal{V}$ eine Liste von Variablen ist.

Intuitiv repräsentiert der Tokenspeicher Informationen über die Propagationsregeln, die noch feuern können. Die logische Bedeutung der Zustände ändert sich nicht gegenüber Definition 3.10.

In Analogie zu Definition 3.8 werden Anfangs- und Endzustände wie folgt definiert:

Definition 4.29 Ein *Anfangszustand* ist ein Zustand von der Form

$$\langle G, \top, \mathit{true}, \emptyset, \mathcal{V} \rangle.$$

Ein *Endzustand* ist von einer der beiden Formen

- $\langle G, C_U, \mathit{false}, C_T, \mathcal{V} \rangle$, d.h. der vordefinierte Speicher ist *false*. Er wird dann *erfolglos* genannt.
- $\langle \top, C_U, C_B, \emptyset, \mathcal{V} \rangle$, wobei kein Berechnungsschritt mehr angewendet werden kann und der vordefinierte Speicher nicht *false* ist. Er wird dann *erfolgreich* genannt.

Der Tokenspeicher

Eine Propagationsregel braucht eine Anwendungsbedingung, um die triviale Nichtterminierung zu verhindern. Unser Ansatz speichert Informationen über Propagationsregeln, die möglicherweise auf ein gegebenes benutzerdefiniertes Constraint angewendet werden können. Sobald eine Propagationsregel angewendet wird, wird der zugehörige Token gelöscht (**Propagate**).

Der Tokenspeicher enthält Regelnamen mit den zugehörigen benutzerdefinierten Constraints, die mit den Köpfen der entsprechenden Propagationsregeln unifizieren. Die Aktualisierung des Tokenspeichers beim Transportieren von benutzerdefinierten Constraints vom Zielspeicher in den Benutzerspeicher hängt von diesen Constraints und dem aktuellen Benutzerspeicher ab (**Introduce**).

Diese Abhängigkeit offenbart sich in der Multimenge $T_{(C,C_U)}$ von Tokens, die die neuen Möglichkeiten, Propagationsregeln anzuwenden, berechnet, wobei die neuen Möglichkeiten das neu einzufügende Constraint C miteinbeziehen:

Definition 4.30 Sei P ein CHR-Programm und C ein benutzerdefiniertes Constraintatom.

$$\begin{aligned} T_{(C,C_U)} \;:=\; & \{R@C \wedge H' \mid (R \;@\; H \Rightarrow G \mid B) \in P, H' \text{ ist eine Subkonjunktion von } C_U, \\ & \;C \text{ ist ein Konjunkt aus } H', \text{ und } C \wedge H' \text{ unifiziert mit } H\} \end{aligned}$$

ist die *Tokenmultimenge* von C bzgl. C_U.
Wenn C eine Konjunktion von Constraints $C_1, \ldots, C_n$ ist, dann

$$T_{(C,C_U)} := T_{(C_1,C_U)} \cup T_{(C_2,C_1 \wedge C_U)} \cup \ldots \cup T_{(C_n,C_1 \wedge \ldots \wedge C_{n-1} \wedge C_U)}$$

Subkonjunktionen können wie folgt definiert werden:

Definition 4.31 Sei $C = \bigwedge_{i=1}^{n} C_i$ eine Konjunktion von Constraints, π eine Permutation über $[1, \ldots, n]$ und $m \leq n$, dann ist $\bigwedge_{i=1}^{m} C_{\pi_k}$ eine *Subkonjunktion* aus C.

Beispiel 4.32 Gegeben sei folgende Propagationsregel:

```
r1 @ p(X) ∧ q(Y) ⇒ r(X,Y).
```

$$T_{(\mathtt{p(a)},\{\mathtt{q(b)} \wedge \mathtt{q(c)}\})} = \{\mathtt{r1@p(a)} \wedge \mathtt{q(b)}, \mathtt{r1@p(a)} \wedge \mathtt{q(c)}\}$$

Erweiterte Normalisierungsfunktion

Die Normalisierungsfunktion aus Abschnitt 3.2 ist für Zustände ohne Tokenspeicher definiert. Im folgenden wird die Definition so erweitert, daß der Tokenspeicher durch die Anwendung dieser Funktion normalisert wird.

Definition 4.33 Eine Funktion $\mathcal{N} : \mathcal{S} \to \mathcal{S}$ ist eine *Normalisierungsfunktion*, wenn sie folgende Bedingungen erfüllt:
Sei $\mathcal{N}(\langle G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V}\rangle$.

1. *Propagation der Gleichheit:* G' und C'_U ergeben sich aus G und C_U, indem alle Variablen X, für die $CT \models \forall\ (C_B \to X \doteq t)$ gilt, durch den entsprechenden Term t ersetzt werden, wenn t keine Variable ist, die in der Ordnung nach X vorkommt. G' und C'_U werden auch *normalisierte Form* von G und C_U genannt.

2. *Projektion:* Folgendes soll gelten:
$$CT \models \forall\ ((\exists \bar{x} C_B) \leftrightarrow C'_B),$$
wobei $\bar{x}$ die strikt lokalen Variablen von $\langle G', C'_U, C_B, \mathcal{T}', \mathcal{V}\rangle$ sind.

3. *Eindeutigkeit:* Wenn
$$\begin{array}{rcl} \mathcal{N}(\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle) & = & \langle G'_1, C'_{U1}, C'_{B1}, \mathcal{T}'_1, \mathcal{V}\rangle \text{ und} \\ \mathcal{N}(\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle) & = & \langle G'_2, C'_{U2}, C'_{B2}, \mathcal{T}'_2, \mathcal{V}\rangle \text{ und} \\ CT \models (\exists \bar{x} C_{B1}) & \leftrightarrow & (\exists \bar{y} C_{B2}) \end{array}$$
gilt, wobei $\bar{x}$ bzw. $\bar{y}$ die strikt lokalen Variablen bezüglich beider Zustände sind, dann:
$$C'_{B1} = C'_{B2}.$$

4. *Tokenbehandlung:* $\mathcal{T}' = \mathcal{T}_1 \cap T_{(C_U, \top)}$, wobei $\mathcal{T}_1$ sich aus $\mathcal{T}$ ergibt, indem alle Variablen X, für die $CT \models \forall\ (C_B \to X \doteq t)$ gilt, durch den entsprechenden Term t ersetzt werden, wenn t keine Variable ist, die in der Ordnung nach X vorkommt.

Die ersten drei Bedingungen entsprechen den Bedingungen aus Definition 3.11 mit dem einzigen Unterschied, daß der Tokenspeicher mit in den Zustand aufgenommen wird. Die letzte Bedingung ist eine neue Eigenschaft der Normalisierungsfunktion, die die Tokens behandelt.

Um die Formalisierung zu vereinfachen, soll im folgenden ohne Beschränkung der Allgemeinheit davon ausgegangen werden, daß die Wächter-Constraints, die Variablen enthalten, die im Rumpf aber nicht im Kopf vorkommen, in dem Rumpf nochmals vorkommen müssen. Es ist leicht zu sehen, daß sich jedes Programm transformieren läßt, so daß diese Bedingung erfüllt ist: Die vordefinierten Constraints, die im Wächter einer Regel vorkommen, werden nochmal in ihrem Rumpf aufgerufen. Diese Transformation verändert die deklarative Semantik nicht.

Beispiel 4.34 Eine CHR-Regel von der Form `p(X)` $\Leftrightarrow$ `Y = 3 | X = Y` muß in eine Regel von der Form `p(X)` $\Leftrightarrow$ `Y = 3 | X = Y, Y = 3` transformiert werden.

Die Reduktionsregeln in Abbildung 4.5 sind ähnlich wie bei der ersten Formalisierung der operationalen Semantik von CHR in Abschnitt 3.2. Es sind einige Unterschiede zu vermerken: Erstens wird der Wächter bei den Reduktionsregeln **Simplify**, **Propagate** und **Simpagate** nicht in den benutzerdefinierten Speicher eingefügt, da durch das Hinzufügen des Rumpfes einer Regel indirekt auch die vordefinierten Constraints des Wächters miteingefügt werden. Das transformierte Programm hat also mit der neuen operationalen Semantik die gleiche Wirkung wie das Ursprungsprogramm im Bezug auf die vordefinierten Constraints im Wächter. Ein weiterer Unterschied ist, daß nach dem Einfügen eines benutzerdefinierten Constraints durch die Reduktionsregel **Introduce** die Information über die Menge der möglichen anwendbaren Propagationsregeln in den

Tokenspeicher hinzugefügt wird. (Das Vereinigungssymbol $\cup$ aus Abbildung 4.5 ist eine Multimengenoperation.) Außerdem kann eine Propagationsregel nur feuern, wenn der entsprechende Token im Tokenspeicher vorhanden ist. Zusätzlich bewirkt die Normalisierungsfunktion, daß bei einem **Simplify**- oder **Simpagate**-Schritt möglicherweise Tokens eliminiert werden, weil dabei benutzerdefinierte Constraints gelöscht werden.

Solve

$$\frac{C \text{ ist ein vordefiniertes Constraint}}{\langle C \wedge G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G, C_U, C \wedge C_B, \mathcal{T}, \mathcal{V}\rangle)}$$

Introduce

$$\frac{H \text{ ist ein benutzerdefiniertes Constraint}}{\langle H \wedge G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G, H \wedge C_U, C_B, \mathcal{T} \cup T_{(H,C_U)}, \mathcal{V}\rangle)}$$

Simplify

$(R@H \Leftrightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \doteq H' \wedge C))}{\langle G, H' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, C_U, H \doteq H' \wedge C_B, \mathcal{T}, \mathcal{V}\rangle)}$$

Propagate

$(R@H \Rightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \doteq H' \wedge C))}{\langle G, H' \wedge C_U, C_B, \{R@H'\} \cup \mathcal{T}, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, H' \wedge C_U, H \doteq H' \wedge C_B, \mathcal{T}, \mathcal{V}\rangle)}$$

Simpagate

$(R@H_1 \backslash H_2 \Leftrightarrow C \mid B)$ ist eine frische Variante einer Regel aus P mit den Variablen $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C))}{\langle G, H_1' \wedge H_2' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle G \wedge B, H_1' \wedge C_U, H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C_B, \mathcal{T}, \mathcal{V}\rangle)}$$

Abbildung 4.5: CHR-Reduktionsregeln: Zweite Formalisierung

Beispiel 4.35 Gegeben sei das CHR-Programm aus Beispiel 3.1

```
reflexivitaet @ X ≤ X ⇔ true.
antisymmetrie @ X ≤ Y ∧ Y ≤ X ⇔ X=Y.
transitivitaet @ X ≤ Y ∧ Y ≤ Z ⇒ X ≤ Z.
```

Die Ableitung aus Beispiel 3.1 und Beispiel 3.5 mit der neuen operationalen Semantik sieht wie folgt aus, wobei $\mathcal{V} = [\mathtt{A}, \mathtt{B}, \mathtt{C}]$:

$$\begin{array}{ll}
 & \langle \underline{\mathtt{A} \le \mathtt{B}} \wedge \mathtt{C} \le \mathtt{A} \wedge \mathtt{B} \le \mathtt{C}, \top, \mathtt{true}, \emptyset, \mathcal{V}\rangle \\
\mapsto (\textbf{Introduce}) & \langle \underline{\mathtt{C} \le \mathtt{A}} \wedge \mathtt{B} \le \mathtt{C}, \mathtt{A} \le \mathtt{B}, \mathtt{true}, \emptyset, \mathcal{V}\rangle \\
\mapsto (\textbf{Introduce}) & \langle \underline{\mathtt{B} \le \mathtt{C}}, \mathtt{A} \le \mathtt{B} \wedge \mathtt{C} \le \mathtt{A}, \mathtt{true}, \mathcal{T}_1, \mathcal{V}\rangle \\
 & \mathcal{T}_1 \supset \{\mathtt{r3@C} \le \mathtt{A} \wedge \mathtt{A} \le \mathtt{B}\} \\
\mapsto (\textbf{Introduce}) & \langle \top, \underline{\mathtt{A} \le \mathtt{B}} \wedge \underline{\mathtt{C} \le \mathtt{A}} \wedge \mathtt{B} \le \mathtt{C}, \mathtt{true}, \mathcal{T}_2, \mathcal{V}\rangle \\
 & \mathcal{T}_2 \supset \mathcal{T}_1 \cup \{\mathtt{r3@A} \le \mathtt{B} \wedge \mathtt{B} \le \mathtt{C}, \mathtt{r3@B} \le \mathtt{C} \wedge \mathtt{C} \le \mathtt{A}\} \\
\mapsto (\textbf{Propagate} \text{ mit } \mathtt{r3}) & \langle \underline{\mathtt{C} \le \mathtt{B}}, \mathtt{A} \le \mathtt{B} \wedge \mathtt{C} \le \mathtt{A} \wedge \mathtt{B} \le \mathtt{C}, \mathtt{true}, \mathcal{T}_3, \mathcal{V}\rangle \\
 & \mathcal{T}_3 \supset \{\mathtt{r3@A} \le \mathtt{B} \wedge \mathtt{B} \le \mathtt{C}, \mathtt{r3@B} \le \mathtt{C} \wedge \mathtt{C} \le \mathtt{A}\} \\
\mapsto (\textbf{Introduce}) & \langle \top, \mathtt{A} \le \mathtt{B} \wedge \mathtt{C} \le \mathtt{A} \wedge \underline{\mathtt{B} \le \mathtt{C}} \wedge \underline{\mathtt{C} \le \mathtt{B}}, \mathtt{true}, \mathcal{T}_4, \mathcal{V}\rangle \\
 & \mathcal{T}_4 \supset \mathcal{T}_3 \cup \{\mathtt{r3@C} \le \mathtt{B} \wedge \mathtt{B} \le \mathtt{C}, \mathtt{r3@B} \le \mathtt{C} \wedge \mathtt{C} \le \mathtt{B}\} \\
\mapsto (\textbf{Simplify} \text{ mit } \mathtt{r2}) & \langle \underline{\mathtt{B} = \mathtt{C}}, \mathtt{A} \le \mathtt{B} \wedge \mathtt{C} \le \mathtt{A}, \mathtt{true}, \mathcal{T}_5, \mathcal{V}\rangle \\
 & \mathcal{T}_5 \supset \{\mathtt{r3@C} \le \mathtt{A} \wedge \mathtt{A} \le \mathtt{B}\}
\end{array}$$

$\mapsto$ (**Solve**)	$\langle \top, \underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{B} \leq \mathtt{A}}, \mathtt{B} = \mathtt{C}, \mathcal{T}_6, \mathcal{V} \rangle$
	$\mathcal{T}_6 \supset \{\mathtt{r3@B} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\}$
$\mapsto$ (**Simplify** mit r2)	$\langle \underline{\mathtt{A} = \mathtt{B}}, \top, \mathtt{B} = \mathtt{C}, \emptyset, \mathcal{V} \rangle$
$\mapsto$ (**Solve**)	$\langle \top, \top, \mathtt{A} = \mathtt{B} \wedge \mathtt{B} = \mathtt{C}, \emptyset, \mathcal{V} \rangle$

Bei einem **Introduce**-Schritt wurden nicht alle Tokens aufgelistet. Wir haben uns hier auf die für die obige Ableitung wichtigsten Tokens beschränkt.

Die Sätze aus Abschnitt 3.3 gelten weiterhin mit der neuen Formalisierung der operationalen Semantik, da die Tokens keinen Einfluß auf die logische Bedeutung der Zustände haben.

Im nächsten Abschnitt werden wir mit Hilfe der neuen operationalen Semantik von CHR eine entscheidbare Charakterisierung der konfluenten terminierenden CHR-Programme angeben. Bei der Definition der kritischen Paare werden die drei Arten der CHR-Regeln berücksichtigt.

Die Simpagationsregeln werden nicht in Simplifikationsregeln übersetzt, da eine Übersetzung, wie sie in Abschnitt 3.1.3 vorgestellt wurde, zu unterschiedlichen Ergebnissen führen kann, wie das folgende Beispiel zeigt:

Beispiel 4.36 Gegeben sei folgendes CHR-Programm P

```
r1 @ p \ q ⇔ true.
r2 @ p ⇒ r.
```

Wenn man die Simpagationsregel `r1` in eine Simplifikationsregel übersetzt, dann bekommt man folgendes CHR-Programm P'

```
r1 @ p ∧ q ⇔ p.
r2 @ p ⇒ r.
```

Wenn man von einem Zustand $\langle \top, \mathtt{p} \wedge \mathtt{q}, \mathtt{true}, \emptyset, [] \rangle$ ausgeht, dann bekommt mit P folgende Ableitung:

$$\begin{array}{ll} & \langle \top, \mathtt{p} \wedge \mathtt{q}, \mathtt{true}, \emptyset, [] \rangle \\ \mapsto & \langle \top, \mathtt{p}, \mathtt{true}, \emptyset, [] \rangle \end{array}$$

Mit P' bekommt man aber eine unterschiedliche Ableitung:

$$\begin{array}{ll} & \langle \top, \mathtt{p} \wedge \mathtt{q}, \mathtt{true}, \emptyset, [] \rangle \\ \mapsto & \langle \mathtt{p}, \top, \mathtt{true}, \emptyset, [] \rangle \\ \mapsto & \langle \top, \mathtt{p}, \mathtt{true}, \{\mathtt{r2@p}\}, [] \rangle \\ \mapsto & \langle \mathtt{r}, \mathtt{p}, \mathtt{true}, \emptyset, [] \rangle \\ \mapsto & \langle \top, \mathtt{p} \wedge \mathtt{r}, \mathtt{true}, \emptyset, [] \rangle \end{array}$$

4.3.2 Kritische Paare und Konfluenz

Wie in Abschnitt 4.2 erläutert, zeichnen sich kritische Zustände dadurch aus, daß zwei verschiedene Regeln auf sie anwendbar sind. Ein kritisches Paar entsteht dann nach der Anwendung zweier Regeln auf den entsprechenden kritischen Zustand. Im Gegensatz zu der Definition der kritischen Paare von SiR-Programmen muß bei allgemeinen CHR-Programmen beachtet werden, daß kritische Paare aus beliebigen Regeln stammen können. Zusätzlich müssen die Tokens miteinbezogen werden.

Wir werden uns auf nicht-triviale kritische Zustände einschränken: Die Zusammenführbarkeit kann nur verletzt werden, wenn die Anwendung einer Regel die Anwendung einer anderen Regel verhindert. Die Anwendung einer Regel kann benutzerdefinierte Constraints löschen und neue Constraints einfügen. Nur das Löschen von benutzerdefinierten Constraints kann dazu führen, daß eine Regel nicht mehr anwendbar ist, falls die gelöschten Constraints für die Anwendung dieser Regel gebraucht werden. Eine Regel muß also eine Simplifikationsregel oder eine Simpagationsregel sein und beide Regeln müssen sich überlappen. Lemma 4.44 zeigt, daß Paare, die bei der Anwendung von zwei Propagationsregeln auf einen gemeinsamen Zustand enstehen, zusammenführbar sind. Dies hat als Konsequenz, daß CHR-Programme, die ausschließlich aus Propagationsregeln bestehen, trivialerweise konfluent sind.

Definition 4.37 Sei R_1 eine Simplifikationsregel bzw. eine Simpagationsregel und R_2 eine beliebige CHR-Regel. Sei C_i der Wächter und B_i der Rumpf der Regel R_i $(i = 1, 2)$. Sei H_i^c und H_i eine Partition vom Kopf der Regel R_i in zwei Konjunkte, wobei H_i^c nicht leer ist und H_1^c, H_2^c unifizierbar sind. Dann ist

$$\langle \top, H_1^c \wedge H_1 \wedge H_2, (H_1^c = H_2^c) \wedge C_1 \wedge C_2, \mathcal{T}, \mathcal{V} \rangle$$

der *kritische Zustand* von R_1 und R_2. $\mathcal{V}$ ist die Liste der Variablen, die in H_1^c, H_1, H_2 vorkommen. Wenn R_2 eine Simplifikationsregel bzw. eine Simpagationsregel ist, dann ist $\mathcal{T}$ leer. Wenn R_2 eine Propagationsregel ist, dann ist $\mathcal{T} = \{R_2@H_1^c \wedge H_2\}$

Die Auswahl des Tokenspeichers $\mathcal{T}$ kann durch das Minimalitätskriterium motiviert werden: Hier geht man davon aus, daß alle Propagationsregeln (ausgenommen möglicherweise R_2) auf die Constraints des benutzerdefiniereten Speicher angewendet wurden, bevor der kritische Zustand erreicht wurde.

Definition 4.38 Ist R eine CHR-Regel, dann berechnet

$$\text{survive}(R) := \begin{cases} \text{true}, & \text{falls } R \text{ eine Simplifikationsregel ist} \\ H\ , & \text{falls } R \text{ eine Propagationsregel mit dem Kopf } H \text{ ist} \\ H_1\ , & \text{falls } R \text{ eine Simpagationsregel mit dem Kopf } H_1 \backslash H_2 \text{ ist} \end{cases}$$

die Constraints, die bei Anwendung von R nicht aus dem Constraintspeicher gelöscht werden.

Definition 4.39 Eine Tokenliste S_T ist *überflüssig* bezüglich eines benutzerdefinierten Speichers C_U, wenn $S_T \cap T_{(C_U, \top)} = \emptyset$.

Definition 4.40 (Kritisches Paar) Sei R_1 eine CHR-Regel mit Wächter C, Rumpf B und Kopf H und sei R_2 eine CHR-Regel mit Wächter C', Rumpf B' und Kopf H'. Seien $\{H_i \,|\, 1 \leq i \leq n\}$ und $\{H_i' \,|\, 1 \leq i \leq m\}$ die Mengen der Atome in H bzw. H'.
Wenn ein oder mehrere Atome $H_{i_1}, \ldots, H_{i_k}$ des Kopfs von R_1 mit einem oder mehreren Atomen $H_{j_1}', \ldots, H_{j_k}'$ des Kopfs von R_2 unifizieren, dann heißt das Paar

$$(\langle B, \text{survive}(R_1) \wedge H_{j_{k+1}}' \wedge \ldots \wedge H_{j_m}', \bar{C}, \mathcal{T}_1, \mathcal{V}\rangle, \langle B', \text{survive}(R_2) \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n}, \bar{C}, \mathcal{T}_2, \mathcal{V}\rangle)$$

ein *kritisches Paar* von R_1 und R_2, falls

- R_1 und R_2 Simplifikationsregeln oder
- R_1 und R_2 Simpagationsregeln oder
- R_1 eine Simplifikationsregel und R_2 eine Propagationsregel oder
- R_1 eine Simplifikationsregel und R_2 eine Simpagationsregel oder
- R_1 eine Simpagationsregel und R_2 eine Propagationsregel.

Wobei $\bar{C} = C \wedge C' \wedge H_{i_1} \doteq H_{j_1}' \wedge \ldots \wedge H_{i_k} \doteq H_{j_k}'$, $\{i_1, \ldots, i_n\}$ und $\{j_1, \ldots, j_m\}$ Permutationen von $\{1, \ldots, n\}$ bzw. $\{1, \ldots, m\}$ sind und $1 \leq k \leq \min(n, m)$. $\mathcal{V}$ ist die Liste der Variablen in $H_1, \ldots, H_n$, $H_1', \ldots, H_m'$. Der Tokenspeicher $\mathcal{T}_2$ ist immer leer. Der Tokenspeicher $\mathcal{T}_1$ ist leer für die ersten vier Fälle. Wenn R_1 eine Simpagationsregel und R_2 eine Propagationsregel sind, dann ist $\mathcal{T}_1$ leer, wenn $\{R_2@H'\}$ bezüglich C_U überflüssig ist, wobei C_U die normaliserte Form von survive$(R_1) \wedge H_{j_{k+1}}' \wedge \ldots \wedge H_{j_m}'$ ist, ansonsten ist $\mathcal{T}_1 = \{R_2@H'\}$.

Die jeweilige Anwendung von R_1 und R_2 auf den kritischen Zustand führt zu zwei Zuständen, die das kritische Paar bilden. In den Zuständen des kritischen Paares ist der Rumpf B_i der Regel R_i im Zielspeicher, H_i^c und H_i (bzw. ein Teil davon) werden vom benutzerdefinierten Speicher gelöscht, falls R_i eine Simplifikationsregel ist (bzw. falls R_i eine Simpagationsregel ist). Der Tokenspeicher kann dann leer werden, entweder durch die Anwendung der Ableitungsregel **Propagate** oder durch die Normalisierungsfunktion nach der Anwendung von **Simplify**. Eine Anwendung von **Simpagate** führt nicht unbedingt dazu, daß der Tokenspeicher leer wird, wie das folgende Beispiel zeigt. Der benutzerdefinierte Speicher und die Liste der Variablen $\mathcal{V}$ bleiben erhalten.

Beispiel 4.41 Gegeben sei folgendes CHR-Programm:

```
r1 @ a \ d ⇔ e.
r2 @ a ∧ b ⇒ c.
```

Ein kritischer Zustand von `r1` und `r2` ist

$$\langle \top, \mathtt{a} \wedge \mathtt{b} \wedge \mathtt{d}, \mathtt{true}, \{\mathtt{r1@a} \wedge \mathtt{b}\}, []\rangle.$$

Die Anwendung von `r1` und `r2` auf den kritischen Zustand resultiert in folgenden Zuständen:

$$\begin{array}{ccc} & \langle \top, \mathtt{a} \wedge \mathtt{b} \wedge \mathtt{d}, \mathtt{true}, \{\mathtt{r1@a} \wedge \mathtt{b}\}, []\rangle & \\ \swarrow^{\mathtt{r1}} & & \searrow^{\mathtt{r2}} \\ \langle \mathtt{e}, \mathtt{a} \wedge \mathtt{b}, \mathtt{true}, \{\mathtt{r1@a} \wedge \mathtt{b}\}, []\rangle & & \langle \mathtt{c}, \mathtt{a} \wedge \mathtt{b} \wedge \mathtt{d}, \mathtt{true}, \emptyset, []\rangle \end{array}$$

Mit dieser Definition der kritischen Paare wird der Satz der lokalen Konfluenz in Abschnitt 4.3.3 bewiesen.

Beispiel 4.42 Sei P das CHR-Programm aus Beispiel 3.1, mit einer zusätzlichen Regel `r4`, die die Idempotenz der Konjunktion ausdrückt.

```
r1 @ X ≤ X ⇔ true.               % Reflexivitaet
r2 @ X ≤ Y ∧ Y ≤ X ⇔ X = Y.     % Antisymmetrie
r3 @ X ≤ Y ∧ Y ≤ Z ⇒ X ≤ Z.     % Transitivitaet
r4 @ X≤Y \ X≤Y ⇔ true.           % Idempotenz
```

Das folgende kritische Paar stammt aus dem Unifizieren des ersten Constraintatoms des Kopfes der Antisymmetrieregel (`r2`) mit dem ersten Constraintatom des Kopfes der Transitivitätsregel (`r3`):

$$(\langle \mathtt{X} \leq \mathtt{Z}, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{X}, \mathtt{true}, \emptyset, [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]\rangle, \langle \mathtt{X} = \mathtt{Y}, \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}, \emptyset, [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]\rangle)$$

Das kritische Paar ist zusammenführbar. Eine Ableitung beginnend mit $\langle \mathtt{X} \leq \mathtt{Z}, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{X}, \mathtt{true}, \emptyset, \mathcal{V}\rangle$ kann wie folgt aussehen, wobei $\mathcal{V} = [\mathtt{X}, \mathtt{Y}, \mathtt{Z}]$:

$$\begin{array}{ll} & \langle \underline{\mathtt{X} \leq \mathtt{Z}}, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{X}, \mathtt{true}, \emptyset, \mathcal{V}\rangle \\ \mapsto (\textbf{Introduce}) & \langle \top, \mathtt{X} \leq \mathtt{Z} \wedge \underline{\mathtt{X} \leq \mathtt{Y}} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \underline{\mathtt{Y} \leq \mathtt{X}}, \mathtt{true}, \mathcal{T}, \mathcal{V}\rangle \\ & \mathcal{T} \supset \{\mathtt{r3@Y} \leq \mathtt{X} \wedge \mathtt{X} \leq \mathtt{Z}\} \\ \mapsto (\textbf{Simplify} \text{ mit } \mathtt{r2}) & \langle \underline{\mathtt{X} = \mathtt{Y}}, \mathtt{X} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}, \mathcal{T}_1, \mathcal{V}\rangle \\ \mapsto (\textbf{Solve}) & \langle \top, \underline{\mathtt{X} \leq \mathtt{Z}} \wedge \underline{\mathtt{X} \leq \mathtt{Z}}, \mathtt{X} = \mathtt{Y}, \mathcal{T}_1, \mathcal{V}\rangle \\ \mapsto (\textbf{Simplify} \text{ mit } \mathtt{r4}) & \langle \top, \mathtt{X} \leq \mathtt{Z}, \mathtt{X} = \mathtt{Y}, \emptyset, \mathcal{V}\rangle \end{array}$$

Eine Berechnungssequenz beginnend mit $\langle \mathtt{X} = \mathtt{Y}, \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}, \emptyset, \mathcal{V}\rangle$ führt zum gleichen Endzustand:

$$\begin{array}{ll} & \langle \underline{\mathtt{X} = \mathtt{Y}}, \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}, \emptyset, \mathcal{V}\rangle \\ \mapsto (\textbf{Solve}) & \langle \top, \mathtt{X} \leq \mathtt{Z}, \mathtt{X} = \mathtt{Y}, \emptyset, \mathcal{V}\rangle \end{array}$$

P hat mehrere kritische Paare, die zusammenführbar sind, also ist P lokal konfluent.

Das CHR-Programm aus Beispiel 4.25 ist nicht konfluent. Wir zeigen jetzt, wie der Token-Ansatz dies entdecken kann.

Beispiel 4.43 Wir betrachten eins der kritischen Paare, die aus den Regeln `r2` und `r3` aus Beispiel 4.25 stammen:

Das kritische Paar $(\langle \texttt{true}, \texttt{p}, \texttt{true}, \emptyset, [] \rangle, \langle \texttt{s}, \top, \texttt{true}, \emptyset, [] \rangle)$ ist nicht zusammenführbar.

$$
\begin{array}{ll}
 & \langle \underline{\texttt{true}}, \texttt{p}, \texttt{true}, \emptyset, [] \rangle \\
\mapsto (\textbf{Solve}) & \langle \top, \texttt{p}, \texttt{true}, \emptyset, [] \rangle
\end{array}
$$

Alle Berechnungssequenzen beginned mit $\langle \texttt{s}, \top, \texttt{true}, \emptyset, [] \rangle$ resultieren in demselben Endzustand unabhängig von der Reihenfolge der Reduktionschritte:

$$
\begin{array}{ll}
 & \langle \underline{\texttt{s}}, \top, \texttt{true}, \emptyset, [] \rangle \\
\mapsto (\textbf{Introduce}) & \langle \top, \underline{\texttt{s}}, \texttt{true}, \emptyset, [] \rangle \\
\mapsto (\textbf{Simplify} \text{ mit } \texttt{r4}) & \langle \underline{\texttt{p}} \wedge \texttt{q}, \top, \texttt{true}, \emptyset, [] \rangle \\
\mapsto (\textbf{Introduce}) & \langle \underline{\texttt{q}}, \texttt{p}, \texttt{true}, \{\texttt{r1@p}\}, [] \rangle \\
\mapsto (\textbf{Introduce}) & \langle \top, \underline{\texttt{p}} \wedge \texttt{q}, \texttt{true}, \{\texttt{r1@p}\}, [] \rangle \\
\mapsto (\textbf{Propagate} \text{ mit } \texttt{r1}) & \langle \underline{\texttt{q}}, \texttt{p} \wedge \texttt{q}, \texttt{true}, \emptyset, [] \rangle \\
\mapsto (\textbf{Introduce}) & \langle \top, \texttt{p} \wedge \texttt{q} \wedge \texttt{q}, \texttt{true}, \emptyset, [] \rangle \\
\mapsto (\textbf{Simpagate} \text{ mit } \texttt{r5}) & \langle \top, \texttt{p} \wedge \texttt{q}, \texttt{true}, \emptyset, [] \rangle
\end{array}
$$

Nach Definition 4.40 kann ein kritisches Paar nicht aus zwei Propagationsregeln entstehen, da Zustände, die bei der Anwendung von zwei Propagationsregeln auf einen gemeinsamen Zustand entstehen, trivialerweise zusammenführbar sind, wie das folgende Lemma zeigt:

Lemma 4.44 Seien R_1 und R_2 zwei Propagationsregeln. Aus $S \mapsto_{R_1} S_1$ und $S \mapsto_{R_2} S_2$ folgt, daß $S_1 \downarrow S_2$.

Beweis: analog zum Fall **Propagate** vs. **Propagate** aus dem Beweis des Satzes der lokalen Konfluenz in Abschnitt 4.3.3. □

4.3.3 Beweis des Satzes der lokalen Konfluenz

Dieser Abschnitt beschäftigt sich mit dem Beweis des Satzes der lokalen Konfluenz. Zuerst geben wir einige Lemmata an, die für den Beweis nötig sind: Die ersten beiden Lemmata beschreiben einige Eigenschaften der Normalisierungsfunktion und die restlichen die Verträglichkeit der Zusammenführbarkeit mit dem Einfügen und Löschen von Informationen.

Das folgende Lemma besagt, daß die Normalisierung keinen Einfluß auf die Anwendung der Regeln hat. Durch dieses Lemma kann dann bei dem Beweis des Satzes der lokalen Konfluenz o.B.d.A. davon ausgegangen werden, daß die Zustände in einer normalisierten Form sind.

Lemma 4.45

$$S \mapsto S' \text{ gdw. } \mathcal{N}(S) \mapsto S'.$$

Beweis: Durch Fallunterscheidung nach dem angewendeten Ableitungsschritt:
Sei $S = \langle G, C_U, C_B, \mathcal{T}, \mathcal{V} \rangle$ und $\mathcal{N}(S) = \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V} \rangle$.

Introduce: Dann ist G von der Form $G_1 \wedge H$, wobei H ein benutzerdefiniertes Constraint ist. Die Anwendung von **Introduce** auf S liefert dann $S' = \mathcal{N}(\langle G_1, H \wedge C_U, C_B, \mathcal{T} \cup T_{(H,C_U)}, \mathcal{V} \rangle)$.
$\mathcal{N}(S) = \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V} \rangle = \langle G'_1 \wedge H', C'_U, C'_B, \mathcal{T}', \mathcal{V} \rangle$. Da H' ein benutzerdefiniertes Constraint ist, kann man es durch den **Introduce**-Schritt in den benutzerdefinierten Speicher transportieren und erhält dann den Zustand $\langle G'_1, H' \wedge C'_U, C'_B, \mathcal{T}' \cup T_{(H',C'_U)}, \mathcal{V} \rangle$, da $\mathcal{N}$ idempotent ist.
Zu zeigen ist: $\mathcal{N}(\langle G_1, H \wedge C_U, C_B, \mathcal{T} \cup T_{(H,C_U)}, \mathcal{V} \rangle) = \langle G'_1, H' \wedge C'_U, C'_B, \mathcal{T}' \cup T_{(H',C'_U)}, \mathcal{V} \rangle$.
Nach der Projektionseigenschaft von $\mathcal{N}$ gilt $CT \models (\exists \bar{x}_1 (C_B) \leftrightarrow C'_B)$, wobei $\bar{x}_1$ die strikt lokalen Variablen in $\langle G'_1 \wedge H', C'_U, C_B, \mathcal{T}', \mathcal{V} \rangle$ sind. Man sieht leicht, daß $CT \models (\exists \bar{x} (C_B) \leftrightarrow C'_B)$ gilt, wobei

$\bar{x}$ die strikt lokalen Variablen bezüglich $\langle G_1, H \wedge C_U, C_B, \mathcal{T} \cup T_{(H,C_U)}, \mathcal{V}\rangle$. Nach der Eindeutigkeitseigenschaft gilt dann, daß die normalisierte Form des vordefinierten Speichers in diesem Zustand gleich C'_B ist. Nach der Gleichheitsprojektion und der Tokenbehandlung von $\mathcal{N}$ haben dann die Zustände identische Zielspeicher, benutzerdefinierte Speicher und Tokenspeicher.

Solve: Dann ist G von der Form $G_1 \wedge C$, wobei C ein vordefiniertes Constraint ist. Die Anwendung von **Solve** auf S liefert dann $S' = \mathcal{N}(\langle G_1, C_U, C_B \wedge C, \mathcal{T}, \mathcal{V}\rangle)$.
$\mathcal{N}(S) = \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V}\rangle = \langle G'_1 \wedge C', C'_U, C'_B, \mathcal{T}', \mathcal{V}\rangle$. $G'_1 \wedge C'$ ergibt sich aus $G_1 \wedge C$ nach der Propagation der Gleichheit (Definition 4.33). C' ist ein vordefiniertes Constraint, dann ist **Solve** anwendbar und man erhält den Zustand $\mathcal{N}(\langle G'_1, C'_U, C'_B \wedge C', \mathcal{T}', \mathcal{V}\rangle)$.
Zu zeigen ist: $\mathcal{N}(\langle G_1, C_U, C_B \wedge C, \mathcal{T}, \mathcal{V}\rangle) = \mathcal{N}(\langle G'_1, C'_U, C'_B \wedge C', \mathcal{T}', \mathcal{V}\rangle)$.
Es gilt $CT \models (\exists \bar{z}_1\ C \wedge C_B) \leftrightarrow (\exists \bar{z}_2\ C'_B \wedge C')$, wobei $\bar{z}_1$ bzw. $\bar{z}_2$ die strikt lokalen Variablen in $\langle G_1, C_U, C_B \wedge C, \mathcal{T}, \mathcal{V}\rangle$ bzw. $\langle G'_1, C'_U, C'_B \wedge C', \mathcal{T}', \mathcal{V}\rangle$. Wegen der Eindeutigkeitseigenschaft von $\mathcal{N}$ sind die vordefinierten Speicher von $\langle G_1, C_U, C_B \wedge C, \mathcal{T}, \mathcal{V}\rangle$ und $\langle G'_1, C'_U, C'_B \wedge C', \mathcal{T}', \mathcal{V}\rangle$ nach der Anwendung der Normalisierungsfunktion identisch. Nach der Gleichheitspropagation von $\mathcal{N}$ haben die Zustände identische Zielspeicher, benutzerdefinierte Speicher und Tokenspeicher.

Simplify: Dann ist C_U von der Form $C_{U1} \wedge H$, wobei H ein benutzerdefiniertes Constraint, das mit dem Kopf einer Simplifikationsregel $H_1 \Leftrightarrow C_1 | G_1$ matcht. Es gilt $CT \models \forall\ (C_B \rightarrow \exists \bar{x}\ (H \doteq H_1 \wedge C_1)$, wobei $\bar{x}$ die frischen Variablen aus der Simplifikationsregel sind.
Die Anwendung von **Simplify** auf S liefert dann $S' = \mathcal{N}(\langle G \wedge B_1, C_{U1}, C_B \wedge H \doteq H_1, \mathcal{T}, \mathcal{V}\rangle)$.
$\mathcal{N}(S) = \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V}\rangle = \langle G', C'_{U1} \wedge H', C'_B, \mathcal{T}', \mathcal{V}\rangle$. Es gilt $CT \models \forall\ (C'_B \rightarrow \exists \bar{x}\ (H' \doteq H_1 \wedge C_1)$, da $CT \models C'_B \leftrightarrow \exists \bar{y} C_B$ gilt, wobei $\bar{y}$ die strikt lokalen Variablen, die in $\langle G', C'_U, C_B, \mathcal{T}', \mathcal{V}\rangle$ vorkommen. Also ist dieselbe Simplifikationsregel auf $\mathcal{N}(S)$ anwendbar und man erhält dann folgenden Zustand $\mathcal{N}(\langle G' \wedge B_1, C'_{U1}, C'_B \wedge H_1 \doteq H', \mathcal{T}', \mathcal{V}\rangle)$. Man kann jetzt leicht zeigen, daß $\mathcal{N}(\langle G' \wedge B_1, C'_{U1}, C'_B \wedge H_1 \doteq H', \mathcal{T}', \mathcal{V}\rangle) = \mathcal{N}(\langle G \wedge B_1, C_{U1}, C_B \wedge H \doteq H_1, \mathcal{T}, \mathcal{V}\rangle)$, in dem man die Eindeutigkeitseigenschaft von $\mathcal{N}$ in Betracht zieht.

Propagate und **Simpagate**: Analog zum Fall **Simplify**. □

Mit der Abdeckungsbedingung aus Definition 4.22 kann man eine Bedingung formulieren, unter der es egal ist, ob man Constraints vor oder nach Anwendung der Normalisierungsfunktion zum vordefinierten Speicher hinzufügt:

Lemma 4.46 Wenn (C, C_B) von $\mathcal{V}$ abgedeckt und C ein vordefiniertes Constraint ist, so folgt aus

$$\mathcal{N}(\langle G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle) = \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V}\rangle,$$
$$\text{daß } \mathcal{N}(\langle G, C_U, C_B \wedge C, \mathcal{T}, \mathcal{V}\rangle) = \mathcal{N}(\langle G', C'_U, C'_B \wedge C, \mathcal{T}', \mathcal{V}\rangle).$$

Beweis: Nach der Projektionseigenschaft von $\mathcal{N}$ gilt

$$CT \models (\exists \bar{x} C_B) \leftrightarrow C'_B,$$

wobei $\bar{x}$ die strikt lokalen Variablen von $\langle G', C'_U, C_B, \mathcal{T}', \mathcal{V}\rangle$ sind.

Seien $\bar{z}$ die strikt lokalen Variablen in $\langle Gs, C_U, C_B \wedge C, \mathcal{V}\rangle$. Wegen der Eindeutigkeitseigenschaft von $\mathcal{N}$ ist zu zeigen, daß $CT \models \exists \bar{z}(C_B \wedge C) \leftrightarrow \exists \bar{y}(C'_B \wedge C)$, wobei $\bar{y}$ die strikt lokalen Variablen in $\langle Gs, C_U, C, \mathcal{V}\rangle$ (denn C'_B enthält keine strikt lokalen Variablen). Weil (C, C_B) von $\mathcal{V}$ abgedeckt ist, besteht $\bar{z}$ aus $\bar{x}$ und $\bar{y}$, wobei die Variablen $\bar{x}$ nicht in C vorkommen. Also gelten die folgenden Äquivalenzen:

$$\begin{aligned} CT \models \exists \bar{z}\ C_B \wedge C &\leftrightarrow \exists \bar{x}\bar{y}\ C_B \wedge C \\ &\leftrightarrow (\exists \bar{x} C_B) \wedge \exists \bar{y} C \\ &\leftrightarrow C'_B \wedge \exists \bar{y} C \\ &\leftrightarrow \exists \bar{y}\ C'_B \wedge C \end{aligned}$$

□

Die Abdeckungsbedingung in Lemma 4.46 reflektiert die Abhängigkeit der Normalisierungsfunktion $\mathcal{N}$ von den strikt lokalen Variablen. Sie garantiert, daß die Gleichheitsconstraints mit den Variablen, die in den einzufügenden Constraints vorkommen, von $\mathcal{N}$ wegen der Lokalität nicht gelöscht werden. Das folgende Beispiel zeigt, daß das Verzichten auf diese Voraussetzung zu Konflikten führen kann, wenn man neue Constraints hinzufügt, die gleiche Variablen enthalten:

Beispiel 4.47

$$\mathcal{N}(\langle \top, \top, X \doteq 2, \emptyset, []\rangle) = \langle \top, \top, \mathit{true}, \emptyset, []\rangle.$$

Addiert man auf beiden Seiten das vordefinierte Constraint $X \doteq 1$, erhält man auf der linken Seite:

$$\mathcal{N}(\langle \top, \top, X \doteq 2 \wedge X \doteq 1, \emptyset, []\rangle) = \langle \top, \top, \mathit{false}, \emptyset, []\rangle$$

während man auf der rechten Seite erhält:

$$\mathcal{N}(\langle \top, \top, \mathit{true} \wedge X \doteq 1, \emptyset, []\rangle) = \langle \top, \top, \mathit{true}, \emptyset, []\rangle.$$

Das folgende Lemma besagt, daß $\mathcal{N}$ verträglich mit dem Einfügen von Contraints in den Constraintspeicher und Tokens in den Tokenspeicher ist:

Lemma 4.48 Sei $\langle G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle$ ein Zustand in Normalform. Wenn $(C_1 \wedge C_2 \wedge C_3, G \wedge C_U \wedge C_B)$ von $\mathcal{V}$ abgedeckt ist und

$$\langle G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle \mapsto^* \langle G', C'_U, C'_B, \mathcal{T}', \mathcal{V}\rangle,$$

dann gilt das folgende für beliebige Tokenlisten S_T:

$$\langle G \wedge C_1, C_U \wedge C_2, C_B \wedge C_3, \mathcal{T} \cup S_T, \mathcal{V}\rangle \mapsto^* \mathcal{N}(\langle G' \wedge C_1, C'_U \wedge C_2, C'_B \wedge C_3, \mathcal{T}' \cup S_T, \mathcal{V}\rangle).$$

Beweis: Die Aussage ist eine Konsequenz aus der Monotonie der logischen Folgerung und Lemma 4.46. □

Lemma 4.49 besagt, daß die Zusammenführbarkeit mit dem Modifizieren des Variablenspeichers verträglich ist. Im folgenden wird nur die Beweisidee erläutert. Ein formaler Beweis findet sich in [Meu96].

Lemma 4.49 Wenn $\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle$ und $\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle$ zusammenführbar sind, dann gelten folgende Aussagen:

a) Wenn $\mathcal{V}' \subseteq \mathcal{V}$, dann sind $\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}'\rangle$ und $\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}'\rangle$ zusammenführbar.

b) Wenn $\mathcal{V}'$ nur Variablen enthält, die in den Zielspeichern, benutzerdefinierten Speichern und vordefinierten Speichern der Zuständen $\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle$ und $\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle$ nicht vorkommen, und $\mathcal{V} \circ \mathcal{V}'$ die Konkatenation beider globalen Speicher ist, so sind die Zustände $\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V} \circ \mathcal{V}'\rangle$ und $\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V} \circ \mathcal{V}'\rangle$ zusammenführbar.

Beweisidee:

a) Wenn man die Anzahl der globalen Variablen reduziert, dann kann dies eine einzige Wirkung haben: Die Variablen, die vorher global waren, sind jetzt strikt lokal. Diese Variablen werden dann durch die Normalisierungsfunktion $\mathcal{N}$ gelöscht. Die vordefinierten Constraints, die diese Variablen enthalten, werden zu vordefinierten Constraints ohne diese Variablen transformiert. Aber das Fehlen der Information über diese Variablen beeinträchtiget nicht die Anwendung der Ableitungsschritte, da strikt lokale Variablen nach Definition nur im vordefinierten Speicher vorkommen können. Formal kann man diese Aussage durch Induktion über die Anzahl der Ableitungsschritte durchführen.

b) Man sieht leicht, daß das Hinzufügen von frischen Variablen zum globalen Variablenspeicher keinen Einfluß auf die Anwendbarkeit der Berechnungsschritte hat. Man erhält sogar, bis auf den modifizierten Variablenspeicher, identische Zustände in den Berechnungen. Damit folgt unmittelbar die Zusammenführbarkeit der zwei modifizierten Zustände. Dies kann durch eine strukturelle Induktion über die Ableitungen nachgewiesen werden.

□

Lemma 4.50 zeigt, daß die Zusammenführbarkeit mit dem Einfügen und Löschen von überflüssigen Tokens verträglich ist. Im folgenden schreiben wir Einfachheitshalber überflüssige Tokens bezüglich eines benutzerdefinierten Speichers, wenn man überfüssige Tokens bezüglich der normalisierten Form des benutzerdefinierten Speichers meint.

Lemma 4.50 Sei S_{T_1} eine überflüssige Tokenliste bezüglich des benutzerdefinierten Speichers des Zutandes (4.1) und S_{T_2} eine überflüssige Tokenliste bezüglich des benutzerdefinierten Speichers des Zustandes (4.2). Dann sind die Zustände

$$\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle \text{ und} \tag{4.1}$$
$$\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle \tag{4.2}$$

genau dann zusammenführbar, wenn

$$\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1 \setminus S_{T_1}, \mathcal{V}\rangle \text{ und}$$
$$\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2 \setminus S_{T_2}, \mathcal{V}\rangle$$

zusammenführbar sind.
Beweis: Wegen der Tokenbehandlungseigenschaft von $\mathcal{N}$ gilt folgendes:

$$\mathcal{N}(\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle) = \mathcal{N}(\langle G_1, C_{U1}, C_{B1}, \mathcal{T}_1 \setminus S_{T_1}, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle) = \mathcal{N}(\langle G_2, C_{U2}, C_{B2}, \mathcal{T}_2 \setminus S_{T_2}, \mathcal{V}\rangle).$$

Also gilt die Aussage wegen Lemma 4.45. □

Jetzt ist man in der Lage, den Satz der lokalen Konfluenz (4.9) zu beweisen, der einen Zusammenhang zwischen lokaler Konfluenz und der Zusammenführbarkeit der kritischen Paare herstellt:

Satz der lokalen Konfluenz: Ein CHR-Programm ist genau dann lokal konfluent, wenn alle kritischen Paare des Programmes zusammenführbar sind.

Beweis: "$\Longrightarrow$": Sei P ein CHR-Programm, dessen kritische Paare zusammenführbar sind. Wir zeigen jetzt, daß P lokal konfluent ist. Wir nehmen an, daß wir uns in einem Zustand S befinden, auf den unterschiedliche Berechnungsschritte anwendbar sind:

$$S \mapsto S_1 \text{ und } S \mapsto S_2.$$

Wegen Lemma 4.45 können wir o.B.d.A. annehmen, daß S normalisiert ist. Zu zeigen ist, daß für alle Kombinationen von angewandten Berechnungsschritten S_1 und S_2 zusammenführbar sind.

Solve vs. Solve: Dann ist S von der Form $\langle C_1 \wedge C_2 \wedge G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle$, wobei C_1, C_2 vordefinierte Constraints sind. Durch den **Solve**-Schritt wird ein vordefiniertes Constraints vom Zielspeicher in den vordefinierten Speicher eingefügt:

$$\begin{array}{rcl}
\langle C_1 \wedge C_2 \wedge G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle & \mapsto & \mathcal{N}(\langle C_2 \wedge G, C_U, C_B \wedge C_1, \mathcal{T}, \mathcal{V}\rangle) \\
& = & \langle C_2' \wedge G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle \text{ und} \\
\langle C_1 \wedge C_2 \wedge G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle & \mapsto & \mathcal{N}(\langle C_1 \wedge G, C_U, C_B \wedge C_2, \mathcal{T}, \mathcal{V}\rangle) \\
& = & \langle C_1' \wedge G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle.
\end{array}$$

Auf den jeweils resultierenden Zustand läßt sich natürlich noch ein **Solve**-Schritt anwenden:

$$\begin{array}{lll}\langle C_2' \wedge G_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle & \mapsto & \mathcal{N}(\langle G_1, C_{U1}, C_{B1} \wedge C_2', \mathcal{T}_1, \mathcal{V}\rangle) = \langle G_3, C_{U3}, C_{B3}, \mathcal{T}_3, \mathcal{V}\rangle \text{ und} \\ \langle C_1' \wedge G_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle & \mapsto & \mathcal{N}(\langle G_2, C_{U2}, C_{B2} \wedge C_1', \mathcal{T}_2, \mathcal{V}\rangle) = \langle G_4, C_{U4}, C_{B4}, \mathcal{T}_4, \mathcal{V}\rangle.\end{array}$$

Jetzt bleibt also nur noch zu zeigen, daß

$$\langle G_3, C_{U3}, C_{B3}, \mathcal{T}_3, \mathcal{V}\rangle = \langle G_4, C_{U4}, C_{B4}, \mathcal{T}_4, \mathcal{V}\rangle$$

Dies gilt, da wegen Lemma 4.46 $\mathcal{N}(\langle G, C_U, C_1 \wedge C_2 \wedge C_B, \mathcal{T}, \mathcal{V}\rangle) = \langle G_3, C_{U3}, C_{B3}, \mathcal{T}_3, \mathcal{V}\rangle$ und $\mathcal{N}(\langle G, C_U, C_1 \wedge C_2 \wedge C_B, \mathcal{T}, \mathcal{V}\rangle) = \langle G_4, C_{U4}, C_{B4}, \mathcal{T}_4, \mathcal{V}\rangle$ ist.

Solve vs. Simplify: S ist von der Form $\langle C' \wedge G, H' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}\rangle$, wobei C' ein vordefiniertes Constraint ist und H' eine Konjunktion von benutzerdefinierten Constraints, die mit dem Kopf einer Regel $(H \Leftrightarrow C \mid B)$ matcht und der Wächter C der Regel vom aktuellen Kontext C_B impliziert wird. Dann hat man folgende Situation:

$$\begin{array}{lll}\langle C' \wedge G, H' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}\rangle & \mapsto_{\mathbf{Solve}} & \mathcal{N}(\langle G, H' \wedge C_U, C_B \wedge C', \mathcal{T}, \mathcal{V}\rangle) \\ & = & \langle G_2, H_2 \wedge C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle = S_{\mathbf{Solve}} \text{ und} \\ \langle C' \wedge G, H' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}\rangle & \mapsto_{\mathbf{Simplify}} & \mathcal{N}(\langle C' \wedge G \wedge B, C_U, C_B \wedge H \doteq H', \mathcal{T}, \mathcal{V}\rangle) \\ & = & \langle C_1 \wedge G_1 \wedge B_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle = S_{\mathbf{Simplify}}.\end{array}$$

C_1 ist ein vordefiniertes Constraint, deswegen ist auf $S_{\mathbf{Simplify}}$ ein **Solve**-Schritt anwendbar. Man bekommt dann

$$S_{\mathbf{Simplify}} \mapsto_{\mathbf{Solve}} \mathcal{N}(\langle G_1 \wedge B_1, C_{U1}, C_{B1} \wedge C_1, \mathcal{T}_1, \mathcal{V}\rangle) = S_{END}$$

Die Anwendung des **Simplify**-Schrittes auf $S_{\mathbf{Solve}}$ ist möglich, da $CT \models \forall\ (C_{B2} \leftrightarrow \exists \bar{x}(C' \wedge C_B))$ ($\bar{x}$ sind strikt lokal in $\langle G_2, H_2 \wedge C_{U2}, C' \wedge C_B, \mathcal{V}\rangle$) und $CT \models \forall (C_B \rightarrow \exists \bar{y}(C \wedge H \doteq H'))$ ($\bar{y}$ sind die frischen Variablen in $(H \Leftrightarrow C \mid B)$), deswegen gilt $CT \models \forall (C_{B2} \rightarrow \exists \bar{y}(C \wedge H \doteq H_2))$. Die Anwendung des **Simplify**-Schrittes auf $S_{\mathbf{Solve}}$ liefert dann

$$S_{\mathbf{Solve}} \mapsto_{\mathbf{Simplify}} \mathcal{N}(\langle G_2 \wedge B, C_{U2}, C_{B2} \wedge H \doteq H', \mathcal{T}_2, \mathcal{V}\rangle) = S'_{END}$$

Wir zeigen jetzt, daß $S_{END} = S'_{END}$:
Es gilt $CT \models (\exists \bar{z}_1\ C_1 \wedge C_{B1}) \leftrightarrow (\exists \bar{z}_2\ C_{B2} \wedge H \doteq H_2)$, wobei $\bar{z}_1$ bzw. $\bar{z}_2$ die strikt lokalen Variablen in $\langle B_1 \wedge G_1, C_{U1}, C_1 \wedge C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle$ bzw. $\langle B \wedge G_2, C_{U2}, C_{B2} \wedge H \doteq H_2, \mathcal{T}_2, \mathcal{V}\rangle$, da folgende Äquivalenzen, die nach der Projektionseigenschaft von $\mathcal{N}$ garantiert sind, gelten:

$$\begin{array}{lll}CT & \models & \forall (C_{B1} \leftrightarrow \exists \bar{x}_1\ C_B \wedge H \doteq H') \\ CT & \models & \forall (C_{B2} \leftrightarrow \exists \bar{x}_2\ C_B \wedge C').\end{array}$$

Dabei sind $\bar{x}_1$ (bzw. $\bar{x}_2$) die strikt lokalen Variablen von $\langle C_1 \wedge G_1 \wedge B_1, C_{U1}, C_B \wedge H \doteq H', \mathcal{T}_1, \mathcal{V}\rangle$ (bzw. $\langle G_2, H_2 \wedge C_{U2}, C_B \wedge C', \mathcal{T}_2, \mathcal{V}\rangle$). Es gilt also

$$CT \models \forall (\exists \bar{x}_1\ C_B \wedge H \doteq H') \leftrightarrow (\exists \bar{x}_2\ C_B \wedge C').$$

Es gilt außerdem $CT \models \forall (\exists \bar{x}_1\ C_B \wedge H \doteq H') \leftrightarrow (\exists \bar{z}_1\ C_1 \wedge C_{B1})$ und $CT \models \forall (\exists \bar{x}_2\ C_B \wedge C') \leftrightarrow (\exists \bar{z}_2\ C_{B2} \wedge H \doteq H_2)$. Daraus folgt

$$CT \models (\exists \bar{z}_1\ C_1 \wedge C_{B1}) \leftrightarrow (\exists \bar{z}_2\ C_{B2} \wedge H \doteq H_2)$$

Wegen der Eindeutigkeitseigenschaft von $\mathcal{N}$ sind die vordefinierten Speicher von S_{END} und S'_{END} identisch. Nach der Gleichheitspropagation von $\mathcal{N}$ haben S_{END} und S'_{END} identische Ziel- und benutzerdefinierte Speicher und nach der Tokenbehandlung identische Tokenspeicher.

Solve vs. Propagate und **Solve vs. Simpagate**: Analog zu **Solve** vs. **Simplify**.

Introduce vs. Introduce: S ist von der Form $\langle H_1 \wedge H_2 \wedge G, C_U, C_B, \mathcal{T}, \mathcal{V}\rangle$, wobei H_1 und H_2 benutzerdefinierte Constraints sind. Das Transportieren von H_1 und dann H_2 vom Zielspeicher in den benutzerdefinierten Speicher hat die gleiche Wirkung wie das Transportieren von H_2 und dann H_1, da die Normalisierungsfunktion nach dem Transportieren von benutzerdefinierten Constraints die Komponenten von S nicht verändert (S ist nach Voraussetzung normalisert) und $T_{(H_1 \wedge H_2, C_U)} = T_{(H_2 \wedge H_1, C_U)}$.

Introduce vs. Simplify: S ist von der Form $\langle H_1 \wedge G, H' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}\rangle$, wobei H_1 ein benutzerdefiniertes Constraintatom ist und H' eine Konjunktion von benutzerdefinierten Constraints, die mit dem Kopf der Regel $(H \Leftrightarrow C|B)$ matcht und der Wächter C der Regel vom aktuellen Kontext C_B impliziert wird. **Introduce** und **Simplify** sind auf S anwendbar und man bekommt:

$$\begin{array}{lll} S & \mapsto_{\textbf{Introduce}} & \mathcal{N}(\langle G, H_1 \wedge H' \wedge C_U, C_B, \mathcal{T} \cup T_{(H_1, H' \wedge C_U)}, \mathcal{V}\rangle) \\ & = & \langle G, H_1 \wedge H' \wedge C_U, C_B, \mathcal{T} \cup T_{(H_1, H' \wedge C_U)}, \mathcal{V}\rangle = S_1 \text{ und} \\ S & \mapsto_{\textbf{Simplify}} & \mathcal{N}(\langle H_1 \wedge G \wedge B, C_U, C_B \wedge H \doteq H', \mathcal{T}, \mathcal{V}\rangle) \\ & = & \langle H_1' \wedge G' \wedge B', C_U', C_B', \mathcal{T}', \mathcal{V}\rangle = S_2. \end{array}$$

Da S nach Voraussetzung normalisiert ist, verändert die Anwendung der Normalisierungsfunktion die Komponenten des nach der Anwendung des **Introduce**-Schrittes resultierenden Zustandes nicht. Auf S_1 ist weiterhin der **Simplify**-Schritt anwendbar:

$$\begin{array}{lll} S_1 & \mapsto_{\textbf{Simplify}} & \mathcal{N}(\langle G \wedge B, H_1 \wedge C_U, C_B \wedge H = H', \mathcal{T} \cup T_{(H_1, H' \wedge C_U)}, \mathcal{V}\rangle) \\ & = & \langle G' \wedge B', H_1' \wedge C_U', C_B', \mathcal{T}_1', \mathcal{V}\rangle = S_1' \end{array}$$

Dies folgt aus der obigen Anwendung der Normalisierungsfunktion auf dem nach der Anwendung des **Simplify**-Schrittes auf S resultierenden Zustandes. Auf S_2 ist der **Introduce**-Schritt anwendbar, da H_1' ein benutzerdefiniertes Constraint ist. Man erhält dann

$$\begin{array}{lll} S_2 & \mapsto_{\textbf{Introduce}} & \mathcal{N}(\langle G' \wedge B', H_1' \wedge C_U', C_B', \mathcal{T}' \cup T_{(H_1', C_U')}, \mathcal{V}\rangle) \\ & = & \langle G' \wedge B', H_1' \wedge C_U', C_B', \mathcal{T}_2', \mathcal{V}\rangle = S_2'. \end{array}$$

Dies gilt, da S_2 normalisiert ist.
Man kann leicht sehen, daß $\mathcal{T}_1' = \mathcal{T}_2'$. Daraus folgt dann $S_1' = S_2'$, d.h. S_1 und S_2 sind zusammenführbar.

Introduce vs. Solve, Introduce vs. Propagate und **Introduce vs. Simpagate**: Diese Fälle können analog zum Fall **Introduce** vs. **Simplify** bewiesen werden.

Simplify vs. Simplify: Seien

$$\begin{array}{ccl} R & @ & H_1 \wedge \ldots \wedge H_n \Leftrightarrow C \mid B \\ R' & @ & H_1' \wedge \ldots \wedge H_{n'}' \Leftrightarrow C' \mid B' \end{array}$$

zwei Simplifikationsregeln, die auf den Zustand S anwendbar sind. Es ist auch möglich, daß R und R' identische Regeln (bis auf Variablenumbenennung) sind. Dies kann zu unterschiedlichen Ergebnissen führen, wenn die Regeln auf unterschiedliche Atome angewendet werden.

Wir zeigen jetzt, daß die Anwendung von R bzw. R' auf den Zustand S zu zusammenführbaren Zuständen führt. Es sind zwei Unterfälle zu erkennen, die anlehnend an die Nomenklatur der Termersetzungssysteme mit „Disjoint Peak“ und „Critical Peak“ bezeichnet werden:

„Disjoint Peak“: Kein Constraint H_i des Regelkopfes von R unifiziert mit einem Constraint H_j des Regelkopfes von R'. Dann sind die Regeln nacheinander anwendbar, da sie auf disjunkte Mengen von Constraints im benutzerdefinierten Speicher angewendet werden und die logische

Konsequenz monoton ist. Die Ergebnisse sind wieder identisch.

„Critical Peak“: Um die Zusammenführbarkeit von S_1 und S_2 nachzuweisen, werden wir die Annahme verwenden, daß alle kritischen Paare zusammenführbar sind. Wir werden ein kritisches Paar aus R und R' bilden und es so modifizeren, bis wir S_1 und S_2 konstruieren, unter der Bedingung, daß die Zusammenführbarkeit erhalten bleibt.

Ohne Beschränkung der Allgemeinheit kann man die Ordnung der Kopfconstraints in R und R' ändern. Wir nehmen an, daß die Variablen des CHR-Programms mit den Variablen des aktuellen Zustandes S disjunkt sind. Wir sind jetzt in der Situation, daß die ersten Kopfatome unifizieren, daß also gilt: $CT \models \exists(H_1 \doteq H_1' \wedge \ldots \wedge H_i \doteq H_i')$, wobei $1 \leq i \leq n$ und $i \leq n'$).

Sei $S = \langle G, G_1 \wedge \ldots \wedge G_m, C_B, \mathcal{T}, \mathcal{V}\rangle$ der aktuelle Zustand. R und R' sind auf S anwendbar, falls folgendes gilt: $CT \models \forall\ (C_B \rightarrow \exists \bar{x} C_1)$ und $CT \models \forall\ (C_B \rightarrow \exists \bar{y} C_2)$, wobei C_1 und C_2 die Konjunktionen der Wächter und der Gleichheitsconstraints (durch Matching), d.h.

$$\begin{array}{rcl} C_1 & = & C \wedge G_1 \doteq H_1 \wedge \ldots \wedge G_i \doteq H_i \wedge G_{i+1} \doteq H_{i+1} \wedge \ldots \wedge G_n \doteq H_n, \\ C_2 & = & C' \wedge G_1 \doteq H_1' \wedge \ldots \wedge G_i \doteq H_i' \wedge G_{n+1} \doteq H_{i+1}' \wedge \ldots \wedge G_{n+n'-i} \doteq H_n'. \end{array}$$

Um Konjunktionen von Constraints zu benennen, werden die folgenden Abkürzungen verwendet:

$$\begin{array}{rcl} \bar{H} & \cong & H_1 \wedge \ldots \wedge H_n, \\ \bar{H}' & \cong & H_1' \wedge \ldots \wedge H_{n'}', \\ \bar{G} & \cong & G_1 \wedge \ldots \wedge G_n, \\ \bar{G}' & \cong & G_1 \wedge \ldots \wedge G_i \wedge G_{n+1} \wedge \ldots \wedge G_{n+n'-i}, \\ \bar{H}_\cap & \cong & H_1 \wedge \ldots \wedge H_i, \\ \bar{H}_\cap' & \cong & H_1' \wedge \ldots \wedge H_i', \\ \bar{G}_R & \cong & G_{n+1} \wedge \ldots \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m, \\ \bar{G}_R' & \cong & G_{i+1} \wedge \ldots \wedge G_n \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m. \end{array}$$

$\bar{H}$ und $\bar{H}'$ sind die Köpfe der Regeln R und R', $\bar{G}$ (bzw. $\bar{G}'$) sind, die benutzerdefinierten Constraints des benutzerdefinierten Speichers von S, die mit H (bzw. H') matchen, $\bar{H}_\cap$ und $\bar{H}_\cap'$ sind die gemeinsamen Teile von $\bar{H}$ und $\bar{H}'$ (d.h. die überlappenden Constraints der Regelköpfe) und $\bar{G}_R$ (bzw. $\bar{G}_R'$) repräsentiert den Inhalt des benutzerdefinierten Speichers nach dem Löschen der gematchten Constraints $\bar{G}$ (bzw. $\bar{G}'$).

Wendet man R bzw. R' auf den aktuellen Zustand S an, erhält man die zwei Zustände S_1 und S_2, von denen im folgenden gezeigt werden soll, daß sie zusammenführbar sind:

$$\begin{array}{rl} S & \mapsto_{\textbf{Simplify}} \ \mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) = S_1 \\ S & \mapsto_{\textbf{Simplify}} \ \mathcal{N}(\langle G \wedge B', \bar{G}_R', C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle) = S_2 \end{array}$$

Die Regeln R und R' haben das folgende kritische Paar

$$(\langle B, H_{i+1}' \wedge \ldots \wedge H_{n'}', C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}_\cap', \emptyset, \mathcal{V}'\rangle, \langle B', H_{i+1} \wedge \ldots \wedge H_n, C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}_\cap', \emptyset, \mathcal{V}'\rangle.$$

Wir benutzen die Abkürzung C_{JOIN}, um den Wächter $C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}_\cap'$ darzustellen. Nach Voraussetzung ist dieses kritische Paar zusammenführbar, also gibt es Berechnungssequenzen von der Form

$$\begin{array}{rcl} \mathcal{N}(\langle B, H_{i+1}' \wedge \ldots \wedge H_{n'}', C_{JOIN}, \emptyset, \mathcal{V}'\rangle) & \mapsto^* & S_1', \\ \mathcal{N}(\langle B', H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \emptyset, \mathcal{V}'\rangle) & \mapsto^* & S_2', \end{array}$$

so daß S_1' und S_2' Varianten sind.

Laut Lemma 4.50 kann man $\mathcal{T}$ zu den Tokenspeichern hinzufügen, da $\mathcal{T}$ bezüglich der benutzerdefinierten Speicher überflüssig ist (In den zwei obigen Zuständen stehen nur Variablen, die in den Regeln selber vorkommen. In $\mathcal{T}$ stehen die aktuellen Variablen, die verschieden von den Variablen des CHR-Programmes sind.) und die Zustände

$$\mathcal{N}(\langle B, H'_{i+1} \wedge \ldots \wedge H'_{n'}, C_{JOIN}, \mathcal{T}, \mathcal{V}' \rangle) \text{ und}$$
$$\mathcal{N}(\langle B', H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \mathcal{T}, \mathcal{V}' \rangle)$$

sind ebenfalls zusammenführbar.
Unter Anwendung von Lemma 4.49 b) kann $\mathcal{V}$ zum globalen Variablenspeicher hinzugefügt werden, da $\mathcal{V}$ keine Variablen mit den Zielspeichern, benutzerdefinierten Speichern und vordefinierten Speichern der obigen zwei Zuständen gemeinsam hat (In diesen Komponenten der zwei obigen Zuständen stehen nur Variablen, die in den Regeln selber vorkommen. In $\mathcal{V}$ stehen die aktuellen Variablen, die verschieden von den Variablen des CHR-Programmes sind.):

$$\mathcal{N}(\langle B, H'_{i+1} \wedge \ldots \wedge H'_{n'}, C_{JOIN}, \mathcal{T}, \mathcal{V}' \circ \mathcal{V} \rangle) \text{ und}$$
$$\mathcal{N}(\langle B', H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \mathcal{T}, \mathcal{V}' \circ \mathcal{V} \rangle)$$

sind zusammenführbar.
Nach Lemma 4.48 kann man die vordefinierten Constraints $C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}'$ zum vordefinierten Speicher und die Constraints $G_{n+n'-i+1} \wedge \ldots \wedge G_m$ zum benutzerdefinierten Speicher beider Zustände hinzufügen und erhält dann die zwei zusammenführbaren Zustände

$$\mathcal{N}(\langle G \wedge B, H'G, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}' \circ \mathcal{V} \rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', HG, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}' \circ \mathcal{V} \rangle),$$

wobei $H'G$ für $H'_{i+1} \wedge \ldots \wedge H'_{n'} \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m$ und HG für $H_{i+1} \wedge \ldots \wedge H_n \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m$ steht. Die Anforderungen des Lemmas sind erfüllt, da die Variablen in $\bar{H}$ und $\bar{H}'$ in $\mathcal{V}'$ enthalten sind und $C_B, \bar{G}, \bar{G}'$ und $G_{n+n'-i+1} \wedge \ldots \wedge G_m$ haben keine gemeinsame Variablen mit den Zielspeichern, benutzerdefinierten Speichern und vordefinierten Speichern der vorhergehenden Zustände.
Jetzt kann laut Lemma 4.49 a) $\mathcal{V}'$ vom globalen Variablenspeicher entfernt werden und die Zusammenführbarkeit der Zustände

$$\mathcal{N}(\langle G \wedge B, H'G, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V} \rangle) \text{ und} \tag{4.3}$$
$$\mathcal{N}(\langle G \wedge B', HG, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V} \rangle) \tag{4.4}$$

bleibt erhalten.
Da sowohl R als auch R' auf den Zustand S anwendbar sind, muß folgendes gelten:

$$CT \models \forall\ (C_B \rightarrow \exists \bar{x} \exists \bar{y}(C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap)),$$

wobei $\bar{x}$ (bzw. $\bar{y}$) die frischen Variablen, die in C und $H_\cap$ (bzw. C' und $H'_\cap$) vorkommen (R und R' enthalten disjunkte Variablen). Daraus folgt

$$\begin{aligned} CT \models \quad \forall \quad & (C_B \wedge \exists \bar{x} \exists \bar{y}(\bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}') \\ & \leftrightarrow C_B \wedge \exists \bar{x} \exists \bar{y}(C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}')) \end{aligned}$$

Aus der Eindeutigkeitseigenschaft von $\mathcal{N}$ folgt, daß die Zustände

$$\mathcal{N}(\langle G \wedge B, H'G, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \emptyset, \mathcal{V} \rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', HG, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \emptyset, \mathcal{V} \rangle)$$

identisch zu den Zuständen (4.3) und (4.4) sind und deswegen ebenfalls zusammenführbar.
Das Propagieren der Gleichheit $\bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}'$ in die Komponenten der Zustände resultiert in Zuständen mit einer identischen normaliserten Form. Deswegen kann man $H'_{i+1} \wedge \ldots \wedge H'_{n'}$

durch $G_{n+1} \wedge \ldots \wedge G_{n+n'-i}$ und analog $H_{i+1} \wedge \ldots \wedge H_n$ durch $G_{i+1} \wedge \ldots \wedge G_n$ und bekommen zusammenführbare Zustände:

$$\mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', \bar{G}'_R, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle).$$

Seien $\bar{x}$ die Variablen aus $\bar{H}'$. In $\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle$ sind diese Variablen strikt lokal, und es gilt (da das Constraint $\bar{G}' \doteq \bar{H}'$ bedeutet, daß $\bar{G}'$ eine Instanz von $\bar{H}'$ ist):

$$CT \models \forall\, (\exists \bar{x}(C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}') \leftrightarrow (C_B \wedge \bar{G} \doteq \bar{H})).$$

Aus der Eindeutigkeitseigenschaft von $\mathcal{N}$ und einer analogen Schlußweise für die Variablen, die in $\bar{H}$ vorkommen, folgt

$$\begin{aligned} & \mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle) \\ = \; & \mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) \\ \text{und} \; & \\ & \mathcal{N}(\langle G \wedge B', \bar{G}'_R, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle) \\ = \; & \mathcal{N}(\langle G \wedge B', \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle). \end{aligned}$$

Daraus folgt dann, daß die Zustände

$$\mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle)$$

zusammenführbar sind.

Simplify vs. Propagate: Es seien zwei CHR-Regeln

$$\begin{array}{lcl} R & @ & H_1 \wedge \ldots \wedge H_n \Leftrightarrow C \mid B \\ R' & @ & H'_1 \wedge \ldots \wedge H'_{n'} \Rightarrow C' \mid B' \end{array}$$

auf den Zustand $S = \langle G, G_1 \wedge \ldots \wedge G_m, C_B, \mathcal{T}, \mathcal{V}\rangle$ anwendbar. Man unterscheidet in diesem Fall auch zwei Fälle:

„Disjoint Peak“: Wie im Falle **Simplify** vs. **Simplify** sind die Ergebnisse nach einer Hintereinanderanwendung von R und dann R' auf S oder von R' und dann R identisch.

„Critical Peak“: Wir verwenden hier die gleichen Abkürzungen, die im Falle **Simplify** vs. **Simplify** eingeführt wurden. Da R' auf S anwendbar ist, muß $R'@\bar{G}'$ in $\mathcal{T}$ vorkommen.

Wendet man R bzw. R' auf den aktuellen Zustand S an, erhält man die zwei Zustände S_1 und S_2, von denen im folgenden gezeigt werden soll, daß sie zusammenführbar sind:

$$\begin{aligned} S_1 &= \mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) \\ S_2 &= \mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) \end{aligned}$$

Die Regeln R und R' haben das kritische Paar

$$(\langle B, H'_{i+1} \wedge \ldots \wedge H'_{n'}, C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap, \emptyset, \mathcal{V}'\rangle, \langle B', \bar{H}' \wedge H_{i+1} \wedge \ldots \wedge H_n, C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap, \emptyset, \mathcal{V}'\rangle,$$

das nach Voraussetzung zusammenführbar ist, d.h. es gibt Berechnungssequenzen

$$\begin{array}{rcl} \langle B, H'_{i+1} \wedge \ldots \wedge H'_{n'}, C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap, \emptyset, \mathcal{V}'\rangle & \mapsto^* & S'_1, \\ \langle B', \bar{H}' \wedge H_{i+1} \wedge \ldots \wedge H_n, C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap, \emptyset, \mathcal{V}'\rangle & \mapsto^* & S'_2, \end{array}$$

so daß S'_1 und S'_2 Varianten sind.
Wenn man die überflüssige Tokenmenge $\mathcal{T} \setminus \{R'@\bar{G}'\}$ bezüglich der benutzerdefinierten Speicher der obigen Zustände zu den Tokenspeichern dieser Zustände hinzufügt, dann sind die resultierenden Zustände

$$\langle B, H'_{i+1} \wedge \ldots \wedge H'_{n'}, C_{JOIN}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}'\rangle \text{ und}$$
$$\langle B', \bar{H}' \wedge H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}'\rangle$$

laut Lemma 4.50 ebenfalls zusammenführbar.
Wenn wir die gleiche Argumentation wie im Falle **Simplify** vs. **Simplify** verwenden, erhalten wir zwei zusammenführbare Zustände

$$\mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle).$$

Wir unterscheiden hier zwei Fälle:

- Die Tokenmenge $\{R'@\bar{G}'\}$ ist bezüglich $\bar{G}_R$ überflüssig, dann kann man diese Menge in den Tokenspeicher des ersten Zustandes einfügen und laut Lemma 4.50 sind die Zustände

  $$\mathcal{N}(\langle G \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) \text{ und}$$
  $$\mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle)$$

  ebenfalls zusammenführbar.

- Die Tokenmenge $\{R'@\bar{G}'\}$ ist bezüglich $\bar{G}_R$ nicht überflüssig, d.h. $\bar{G}'$ kommt in $\bar{G}_R$ vor. Dann kann ein **Propagate**-Schritt mit R' auf S_1 durchgeführt werden. Man sieht leicht, daß die Anwendung von R' auf S_1 und R auf S_2 zu einem identischen Zustand führt:

$$\mathcal{N}(\langle G \wedge B \wedge B', \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle)$$

Propagate vs. Propagate: Seien R und R' die Propagationsregeln, die auf den aktuellen Zustand $S = \langle G, G_1 \wedge \ldots \wedge G_m, C_B, \mathcal{T}, \mathcal{V}\rangle$ anwendbar sind:

$$\begin{array}{lcl} R & @ & H_1 \wedge \ldots \wedge H_n \Rightarrow C \mid B \\ R' & @ & H'_1 \wedge \ldots \wedge H'_{n'} \Rightarrow C' \mid B' \end{array}$$

Da sowohl R als auch R' auf S anwendbar sind, müssen die Tokens $R@\bar{G}$ und $R'@\bar{G}'$ im Tokenspeicher $\mathcal{T}$ vorkommen. Die Anwendung von R und R' auf S liefert folgende Zustandsübergänge:

$$\begin{array}{lll} S & \mapsto_{\mathbf{Propagate}} & \mathcal{N}(\langle G \wedge B, \bar{G} \wedge \bar{G}_{REST}, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T} \setminus \{R@\bar{G}\}, \mathcal{V}\rangle) \\ & & \langle G_1 \wedge B_1, \bar{G}_1 \wedge \bar{G}_{REST_1}, C_{B1}, \mathcal{T}_1, \mathcal{V}\rangle = S_1 \\ S & \mapsto_{\mathbf{Propagate}} & \mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_{REST}, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) \\ & & \langle G_2 \wedge B'_2, \bar{G}'_2 \wedge \bar{G}_{REST'_2}, C_{B2}, \mathcal{T}_2, \mathcal{V}\rangle = S_2 \end{array}$$

R' ist auf S_1 anwendbar und R auf S_2. Man erhält dann:

$$\begin{array}{lll} S_1 & \mapsto_{\mathbf{Propagate}} & \mathcal{N}(\langle G_1 \wedge B_1 \wedge B', \bar{G}_1 \wedge \bar{G}_{REST_1}, C_{B1} \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}_1 \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) = S_3 \\ S_2 & \mapsto_{\mathbf{Propagate}} & \mathcal{N}(\langle G_2 \wedge B'_2 \wedge B, \bar{G}'_2 \wedge \bar{G}'_{REST_2}, C_{B2} \wedge \bar{G} \doteq \bar{H}, \mathcal{T}_2 \setminus \{R@\bar{G}\}, \mathcal{V}\rangle) = S_4 \end{array}$$

Nach der Projektionseigenschaft von $\mathcal{N}$ gilt folgendes:

$$\begin{array}{lcl} CT & \models & \forall\, (C_{B1} \leftrightarrow \exists \bar{x}\ (C_B \wedge \bar{G} \doteq \bar{H})) \text{ und} \\ CT & \models & \forall\, (C_{B2} \leftrightarrow \exists \bar{y}\ (C_B \wedge \bar{G}' \doteq \bar{H}')), \end{array}$$

wobei $\bar{x}$ (bzw. $\bar{y}$) die strikt lokalen Variablen von $\langle G_1 \wedge B_1, \bar{G}_1 \wedge \bar{G}_{REST_1}, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}_1, \mathcal{V}\rangle$ (bzw. $\langle G_2 \wedge B_2', \bar{G}_2' \wedge \bar{G}_{REST_2'}, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}_2, \mathcal{V}\rangle$).
Daraus kann man folgern, daß

$$CT \quad \models \quad \exists \bar{z}_1\ (C_{B1} \wedge \bar{G}' \doteq \bar{H}') \leftrightarrow \exists \bar{z}_2\ (C_{B2} \wedge \bar{G} \doteq \bar{H}),$$

gilt, wobei $\bar{z}_1$ (bzw. $\bar{z}_2$) die strikt lokalen Variablen von S_3 (bzw. S_4). Also sind die normalisierten vordefinierten Speicher von S_3 und S_4 wegen der Eindeutigkeitsbedingung identisch. Nach der Propagation der Gleichheit und der Tokenbehandlung sind die restlichen Komponenten ebenfalls identisch. Also ist $S_3 = S_4$.

Simpagate vs. Simplify: Es seien zwei CHR-Regeln

$$\begin{array}{lcl} R & @ & H_1 \wedge \ldots \wedge H_l \backslash H_{l+1} \wedge \ldots \wedge H_n \Leftrightarrow C \mid B \\ R' & @ & H_1' \wedge \ldots \wedge H_{n'}' \Leftrightarrow C' \mid B' \end{array}$$

auf den Zustand S anwendbar. Man unterscheidet in diesem Fall auch zwei Fälle:

„Disjoint Peak“: Wie im Falle **Simplify** vs. **Simplify** sind die Ergebnisse nach einer Hintereinanderanwendung von R und dann R' auf S oder von R' und dann R identisch.

„Critical Peak“: Zunächst führen wir weitere Abkürzungen ein: G_l bezeichnet den Teil der Constraints aus $\bar{G}$, der bei der Anwendung eines **Simpagate**-Schrittes im benutzerdefinierten Speicher bleibt.

$$\begin{array}{lcl} \bar{G}_l & \cong & G_1 \wedge \ldots \wedge G_l \\ \bar{H}_l & \cong & H_1 \wedge \ldots \wedge H_l \end{array}$$

Wendet man R bzw. R' auf den aktuellen Zustand S an, erhält man die zwei Zustände S_1 und S_2, von denen im folgenden gezeigt werden soll, daß sie zusammenführbar sind:

$$\begin{array}{lll} S & \mapsto_{\textbf{Simpagate}} & \mathcal{N}(\langle G \wedge B, \bar{G}_l \wedge \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) = S_1 \\ S & \mapsto_{\textbf{Simplify}} & \mathcal{N}(\langle G \wedge B', \bar{G}_R', C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle) = S_2 \end{array}$$

Die Regeln R und R' haben das folgende kritische Paar

$$(\langle B, \bar{H}_l \wedge H_{i+1}' \wedge \ldots \wedge H_{n'}', C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}_\cap', \emptyset, \mathcal{V}'\rangle, \langle B', H_{i+1} \wedge \ldots \wedge H_n, C \wedge C' \wedge \bar{H}_\cap \doteq \bar{H}_\cap', \emptyset, \mathcal{V}'\rangle,$$

das nach Voraussetzung zusammenführbar ist, also gibt es Berechnungssequenzen von der Form

$$\begin{array}{rcl} \mathcal{N}(\langle B, \bar{H}_l \wedge H_{i+1}' \wedge \ldots \wedge H_{n'}', C_{JOIN}, \emptyset, \mathcal{V}'\rangle) & \mapsto^* & S_1', \\ \mathcal{N}(\langle B', H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \emptyset, \mathcal{V}'\rangle) & \mapsto^* & S_2', \end{array}$$

so daß S_1' und S_2' Varianten sind.
Wenn man die überflüssige Tokenmenge $\mathcal{T}$ bezüglich der benutzerdefinierten Speicher der obigen Zustände zu den Tokenspeichern dieser Zustände hinzufügt, dann sind die resultierenden Zustände

$$\begin{array}{r} \langle B, H_l \wedge H_{i+1}' \wedge \ldots \wedge H_{n'}', C_{JOIN}, \mathcal{T}, \mathcal{V}'\rangle \text{ und} \\ \langle B', H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \mathcal{T}, \mathcal{V}'\rangle \end{array}$$

laut Lemma 4.50 ebenfalls zusammenführbar.
Wenn wir die gleiche Argumentation wie im Falle **Simplify** + **Simplify** verwenden, erhalten wir zwei zusammenführbare Zustände

$$\begin{array}{l} \mathcal{N}(\langle G \wedge B, \bar{G}_l \wedge \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) \text{ und} \\ \mathcal{N}(\langle G \wedge B', \bar{G}_R', C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle). \end{array}$$

Simpagate vs. Propagate: Es seien zwei CHR-Regeln

$$\begin{array}{lll} R & @ & H_1 \wedge \ldots \wedge H_l \backslash H_{l+1} \wedge \ldots \wedge H_n \Leftrightarrow C \mid B \\ R' & @ & H'_1 \wedge \ldots \wedge H'_{n'} \Rightarrow C' \mid B' \end{array}$$

auf den Zustand S anwendbar. Man unterscheidet in diesem Fall auch zwei Fälle:

„Disjoint Peak“: Wie im Falle **Simplify** vs. **Simplify** sind die Ergebnisse nach einer Hintereinanderanwendung von R und dann R' auf S oder von R' und dann R identisch.

„Critical Peak“: Wendet man R bzw. R' auf den aktuellen Zustand S an, erhält man die zwei Zustände S_1 und S_2, von denen im folgenden gezeigt werden soll, daß sie zusammenführbar sind:

$$\begin{array}{lll} S & \mapsto_{\textbf{Simpagate}} & \mathcal{N}(\langle G \wedge B, \bar{G}_l \wedge \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T}, \mathcal{V}\rangle) = S_1 \\ S & \mapsto_{\textbf{Propagate}} & \mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) = S_2 \end{array}$$

Wenn $R'@\bar{G}'$ bezüglich $\bar{G}_l \wedge \bar{G}_R$ nicht überflüssig ist, dann kann man auf S_1 einen **Propagate**-Schritt mit Hilfe der Regel R' durchführen und auf S_2 einen **Simpagate**-Schritt mit Hilfe der Regel R und erhält identische Zustände (Der Beweis ist analog zu dem Fall **Propagate** vs. **Propagate**).
Im folgenden gehen wir davon aus, daß $R'@\bar{G}'$ bezüglich $\bar{G}_l \wedge \bar{G}_R$ überflüssig ist.
Die Regeln R und R' haben das folgende kritische Paar

$$(\langle B, \bar{H}_l \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, C_{JOIN}, \mathcal{T}', \mathcal{V}'\rangle, \langle B', \bar{H}' \wedge H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \emptyset, \mathcal{V}'\rangle,$$

das nach Voraussetzung zusammenführbar ist, wobei $\mathcal{T}'$ entweder leer ist oder aus dem Token $R'@\bar{H}'$ besteht, also gibt es Berechnungssequenzen von der Form

$$\begin{array}{rll} \mathcal{N}(\langle B, \bar{H}_l \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, C_{JOIN}, \mathcal{T}', \mathcal{V}'\rangle) & \mapsto^* & S'_1, \\ \mathcal{N}(\langle B', \bar{H}' \wedge H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \emptyset, \mathcal{V}'\rangle) & \mapsto^* & S'_2, \end{array}$$

so daß S'_1 und S'_2 Varianten sind.
Wenn $\mathcal{T}'$ aus dem Token $R'@\bar{H}'$ besteht, d.h. $R'@\bar{H}'$ ist bezüglich $\bar{H}_l \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}$ nicht überflüssig, dann ist das Token $R'@\bar{G}'$ bezüglich $\bar{G}_l \wedge \bar{G}_R$ ebenfalls nicht überflüssig, da man G_R aus $H'_{i+1} \wedge \ldots \wedge H'_{n'}$ gewinnt, in dem man zusätzliche Constraints einfügt (Siehe Fall **Simplify** vs. **Simplify**).
Also betrachten wir nur den Fall, daß $\mathcal{T}' = \emptyset$ ist. Dann kann man die überflüssige Tokenmenge $\mathcal{T} \setminus \{R'@\bar{G}'\}$ bezüglich der benutzerdefinierten Speicher der obigen Zustände in den Tokenspeicher dieser Zustände einfügen und die resultierenden Zustände

$$\mathcal{N}(\langle B, \bar{H}_l \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, C_{JOIN}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}'\rangle) \text{ und}$$
$$\mathcal{N}(\langle B', \bar{H}' \wedge H_{i+1} \wedge \ldots \wedge H_n, C_{JOIN}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}'\rangle)$$

sind ebenfalls zusammenführbar.
Wenn wir die gleiche Argumentation wie im Falle **Simplify** + **Simplify** verwenden, erhalten wir zwei zusammenführbare Zustände

$$\mathcal{N}(\langle G \wedge B, \bar{G}_l \wedge \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle).$$

Da $R'@\bar{G}'$ bezüglich $\bar{G}_l \wedge \bar{G}_R$ überflüssig ist, kann man laut Lemma 4.50 die Menge $\{R'@\bar{G}'\}$ in den Tokenspeicher des obigen ersten Zustandes einfügen und die Zustände

$$\mathcal{N}(\langle G \wedge B, \bar{G}_l \wedge \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{T} \setminus \{R'@\bar{G}'\}, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle G \wedge B', \bar{G}' \wedge \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{T}, \mathcal{V}\rangle)$$

sind ebenfalls zusammenführbar.

Simpagate vs. Simpagate: Dieser Fall kann analog zum Fall **Simplify** vs. **Simplify** bewiesen werden.

"$\Longleftarrow$": Sei P ein lokal konfluentes CHR-Programm. Wir beweisen durch Widerspruch, daß alle kritischen Paare zusammenführbar sind. Annahme: Es gibt ein kritisches Paar, das nicht zusammenführbar ist. Wir konstruieren dann seinen direkten gemeinsamen Vorgänger (den kritischen Zustand) und verwenden die lokale Konfluenz, um der Annahme zu widersprechen. Im folgenden werden fünf Fälle unterschieden, je nach Art der Regeln, aus denen das kritische Paar hervorgeht:

(1) Sei $(\langle B_1, H_2, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle, \langle B_2, H_1, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle)$ das nicht zusammenführbare kritische Paar, das aus folgenden Simplifikationsregeln stammen:

$$\begin{array}{lcl} R_1 @\ H_1 \wedge H_3 & \Leftrightarrow & C_1 \mid B_1 \\ R_2 @\ H_2 \wedge H_4 & \Leftrightarrow & C_2 \mid B_2, \end{array}$$

wobei H_3 und H_4 unifizierbar sind und nicht leer. Das kritische Paar ist nicht zusammenführbar, d.h. die Zustände

$$\begin{array}{l} \langle B_1, H_2, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und} \\ \langle B_2, H_1, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \end{array}$$

sind nicht zusammenführbar.
Sei $\langle \top, H'_1 \wedge H'_2 \wedge H'_3, C'_1 \wedge C'_2, \emptyset, \mathcal{V}'\rangle$ eine frische Variante des Zustandes $\langle \top, H_1 \wedge H_2 \wedge H_3, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$. Nun kann man auf den Zustand $\langle \top, H'_1 \wedge H'_2 \wedge H'_3, C'_1 \wedge C'_2, \emptyset, \mathcal{V}'\rangle$ sowohl R_1 als auch R_2 anwenden, wobei man als Ergebnis die zwei Zustände

$$\begin{array}{l} \mathcal{N}(\langle B''_1, H'_2, C'_1 \wedge C'_2 \wedge H''_1 \doteq H'_1 \wedge H''_3 \doteq H'_3, \emptyset, \mathcal{V}\rangle) \text{ und} \\ \mathcal{N}(\langle B''_2, H'_1, C'_1 \wedge C'_2 \wedge H''_2 \doteq H'_2 \wedge H''_4 \doteq H'_3, \emptyset, \mathcal{V}\rangle) \end{array}$$

erhält. Die frischen Varianten der Regeln R_1 und R_2 sind $H''_1 \wedge H''_3 \Leftrightarrow C''_1 \mid B''_1$ und $H''_2 \wedge H''_4 \Leftrightarrow C''_2 \mid B''_2$. Wegen der lokalen Konfluenz von P sind diese zwei Zustände zusammenführbar. Die Normalisierungsfunktion $\mathcal{N}$ propagiert die Gleichheiten $H''_1 \doteq H'_1 \wedge H''_3 \doteq H'_3$ bzw. $H''_2 \doteq H'_2 \wedge H''_4 \doteq H'_3$ in den entsprechenden Zielspeicher. Wir können dann B''_1 bzw. B''_2 durch B'_1 bzw. B'_2 ersetzen, ohne das Ergebnis der Normaliserungsfunktion zu beeinflußen.
Wenn die Variablen in B''_1 bzw. B''_2 nicht in $H''_1 \wedge H''_3$ bzw. $H''_2 \wedge H''_4$ vorkommen, d.h. Variablen, deren Wert nicht durch die Gleichheiten $H''_1 \doteq H'_1 \wedge H''_3 \doteq H'_3$ und $H''_2 \doteq H'_2 \wedge H''_4 \doteq H'_3$ bestimmt sind, können sie durch neue Variablen ersetzt werden und die Zusammenführbarkeit bleibt erhalten.
Also sind die Zustände

$$\mathcal{N}(\langle B'_1, H'_2, C'_1 \wedge C'_2 \wedge H''_1 \doteq H'_1 \wedge H''_3 \doteq H'_3, \emptyset, \mathcal{V}'\rangle) \text{ und} \tag{4.5}$$
$$\mathcal{N}(\langle B'_2, H'_1, C'_1 \wedge C'_2 \wedge H''_2 \doteq H'_2 \wedge H''_4 \doteq H'_3, \emptyset, \mathcal{V}'\rangle) \tag{4.6}$$

zusammenführbar.
Die Zustände 4.7 und 4.8 haben wegen der Eindeutigkeitseigenschaft von $\mathcal{N}$ die gleiche Normalform wie die beiden oberen, da die Variablen, die in H''_3 (bzw. H''_4) vorkommen, strikt lokal in dem Zustand 4.5 (bzw. 4.6) sind und folgendes gilt:

$$\begin{array}{lcl} CT & \models & \exists \bar{x}\ (C'_1 \wedge C'_2 \wedge H''_1 \doteq H'_1 \wedge H''_3 \doteq H'_3) \leftrightarrow \exists \bar{y}\ (C'_1 \wedge C'_2) \text{ und} \\ CT & \models & \exists \bar{x}_1\ (C'_1 \wedge C'_2 \wedge H''_2 \doteq H'_2 \wedge H''_4 \doteq H'_3) \leftrightarrow \exists \bar{y}_1\ (C'_1 \wedge C'_2), \end{array}$$

wobei $\bar{x}$ (bzw. $\bar{x}_1$) die strikt lokalen Variablen des Zustandes 4.5 (bzw. 4.6) sind und $\bar{y}$ (bzw. $\bar{y}_1$) die strikt lokalen Variablen des Zustandes 4.7 (bzw. 4.8). Laut Lemma 4.45 sind also folgende Zustände :

$$\langle B_1', H_2', C_1' \wedge C_2', \emptyset, \mathcal{V}' \rangle \text{ und} \tag{4.7}$$
$$\langle B_2', H_1', C_1' \wedge C_2', \emptyset, \mathcal{V}' \rangle, \tag{4.8}$$

zusammenführbar. Dies ist ein Widerspruch zur Annahme, daß die varianten Zustände

$$\langle B_1, H_2, C_1 \wedge C_2, \emptyset, \mathcal{V} \rangle \text{ und}$$
$$\langle B_2, H_1, C_1 \wedge C_2, \emptyset, \mathcal{V} \rangle$$

nicht zusammenführbar sind.

(2) Sei $(\langle B_1, H_1 \wedge H_2 \wedge H_3, G1 \wedge G2, \emptyset, \mathcal{V} \rangle, \langle B_2, H_1, G1 \wedge G2, \emptyset, \mathcal{V} \rangle)$ das nicht-zusammenführbare kritische Paar, das aus einer Simplifikationsregel und einer Propagationsregel stammt:

$$R_1 \ @ \ H_1, H_3 \ \Rightarrow \ C_1 \mid B_1$$
$$R_2 \ @ \ H_2, H_4 \ \Leftrightarrow \ C_2 \mid B_2,$$

wobei H_3 und H_4 unifizierbar sind und nicht leer. Das kritische Paar ist nicht zusammenführbar, d.h. die Zustände

$$\langle B_1, H_1 \wedge H_2 \wedge H_3, C_1 \wedge C_2, \emptyset, \mathcal{V} \rangle \text{ und}$$
$$\langle B_2, H_1, C_1 \wedge C_2, \emptyset, \mathcal{V} \rangle$$

sind nicht zusammenführbar.
Sei $\langle \top, H_1' \wedge H_2' \wedge H_3', C_1' \wedge C_2', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}' \rangle$ eine frische Variante von $\langle \top, H_1 \wedge H_2 \wedge H_3, C_1 \wedge C_2, \{R_1 @ H_1 \wedge H_3\}, \mathcal{V} \rangle$. Die Anwendung von R_1 und R_2 auf den Zustand $\langle \top, H_1' \wedge H_2' \wedge H_3', C_1' \wedge C_2', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}' \rangle$ resultiert in folgenden Zuständen

$$\mathcal{N}(\langle B_1'', H_1' \wedge H_2' \wedge H_3', C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V} \rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2'', H_1', C_1' \wedge C_2' \wedge H_2'' \doteq H_2' \wedge H_4'' \doteq H_3', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V} \rangle),$$

die wegen der lokalen Konfluenz von P zusammenführbar sind. Mit der gleichen Argumentation wie im Falle (1) sind die Zustände

$$\mathcal{N}(\langle B_1', H_1' \wedge H_2' \wedge H_3', C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}' \rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2', H_1', C_1' \wedge C_2' \wedge H_2'' \doteq H_2' \wedge H_4'' \doteq H_3', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}' \rangle)$$

zusammenführbar.
Die Tokenmenge $\{R_1 @ H_1' \wedge H_3'\}$ ist bezüglich H_1' überflüssig, da H_3' nicht leer ist. Also sind die Zustände

$$\mathcal{N}(\langle B_1', H_1' \wedge H_2' \wedge H_3', C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}' \rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2', H_1', C_1' \wedge C_2' \wedge H_2'' \doteq H_2' \wedge H_4'' \doteq H_3', \emptyset, \mathcal{V}' \rangle)$$

laut Lemma 4.50 ebenfalls zusammenführbar. Die folgenden Zustände haben die gleiche Normalform wie die oberen Zustände (mit einer analogen Argumentation zum ersten Fall. Laut Lemma 3.13 sind dann die Zustände

$$\langle B_1', H_1' \wedge H_2' \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}' \rangle \text{ und}$$
$$\langle B_2', H_1', C_1' \wedge C_2', \emptyset, \mathcal{V}' \rangle.$$

zusammenführbar. Dies ist ein Widerspruch zu der Annahme, daß die varianten Zustände

$$\langle B_1, H_1 \wedge H_2 \wedge H_3, C_1 \wedge C_2, \emptyset, \mathcal{V} \rangle \text{ und}$$
$$\langle B_2, H_1, C_1 \wedge C_2, \emptyset, \mathcal{V} \rangle$$

nicht zusammenführbar sind.

3) Sei $(\langle B_1, H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle, \langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle)$ das nicht-zusammenführbare kritische Paar, das aus einer Simplifikationsregel und einer Simpagationsregel stammt:

$$R_1 \ @ \ H_1, H_3 \ \Leftrightarrow \ C_1 \mid B_1$$
$$R_2 \ @ \ H_2 \backslash H_4 \ \Leftrightarrow \ C_2 \mid B_2,$$

wobei H_3 und H_5 unifizierbar sind, H_5 ein Konjunkt von $H_2 \wedge H_4$ und $H_2 \wedge H_4 = H_2^c \wedge H_4^c \wedge H_5$. Das kritische Paar ist nicht zusammenführbar, d.h. die Zustände

$$\langle B_1, H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$$

sind nicht zusammenführbar.
Sei $\langle \top, H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle$ eine frische Variante von $\langle \top, H_1 \wedge H_2^c \wedge H_4^c \wedge H_3, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$. Die Anwendung von R_1 und R_2 auf den Zustand $\langle \top, H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle$ resultiert in folgenden Zuständen

$$\mathcal{N}(\langle B_1'', H_2'^c \wedge H_4'^c, C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2'', H_1' \wedge H_2', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_3'), \emptyset, \mathcal{V}\rangle),$$

die wegen der lokalen Konfluenz von P zusammenführbar sind. Nach der Propagierung der Gleichheiten und mit der gleichen Argumentation wie im Falle (1) sind die Zustände

$$\mathcal{N}(\langle B_1', H_2'^c \wedge H_4'^c, C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}'\rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2', H_1' \wedge H_2', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_3'), \emptyset, \mathcal{V}'\rangle)$$

zusammenführbar.
Die folgenden Zustände haben die gleiche Normalform wie die oberen Zustände. Laut Lemma 3.13 sind dann die Zustände

$$\langle B_1', H_2'^c \wedge H_4'^c, C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle \text{ und}$$
$$\langle B_2', H_1' \wedge H_2', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle.$$

zusammenführbar.
Dies ist ein Widerspruch zu der Annahme, daß die varianten Zustände

$$\langle B_1, H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$$

nicht zusammenführbar sind.

4) Sei $(\langle B_1, H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle, \langle B_2, H_1^c \wedge H_3^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle)$ das nicht-zusammenführbare kritische Paar, das aus zwei Simpagationsregeln stammt:

$$R_1 \ @ \ H_1 \backslash H_3 \ \Leftrightarrow \ C_1 \mid B_1$$
$$R_2 \ @ \ H_2 \backslash H_4 \ \Leftrightarrow \ C_2 \mid B_2,$$

wobei H_5 und H_6 unifizierbar sind, H_5 ein Konjunkt von $H_1 \wedge H_3$, H_6 ein Konjunkt von $H_2 \wedge H_4$, $H_1 \wedge H_3 = H_1^c \wedge H_3^c \wedge H_5$ und $H_2 \wedge H_4 = H_2^c \wedge H_4^c \wedge H_6$. Das kritische Paar ist nicht zusammenführbar, d.h. die Zustände

$$\langle B_1, H_1 \wedge H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_4 \wedge H_1^c \wedge H_3^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$$

sind nicht zusammenführbar.
Sei $\langle \top, H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle$ eine frische Variante von $\langle \top, H_1 \wedge H_2^c \wedge H_4^c \wedge$

$H_3, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$. Die Anwendung von R_1 und R_2 auf den Zustand $\langle \top, H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle$ resultiert in folgenden Zuständen

$$\mathcal{N}(\langle B_1'', H_1' \wedge H_2'^c \wedge H_4'^c, C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2'', H_1'^c \wedge H_3'^c \wedge H_4', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_5'), \emptyset, \mathcal{V}\rangle),$$

die wegen der lokalen Konfluenz von P zusammenführbar sind. Nach der Propagierung der Gleichheiten und mit der gleichen Argumentation wie im Falle (1) sind die Zustände

$$\mathcal{N}(\langle B_1', H_1' \wedge H_2'^c \wedge H_4'^c, C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}'\rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2', H_1'^c \wedge H_3'^c \wedge H_4', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_5'), \emptyset, \mathcal{V}'\rangle)$$

zusammenführbar.
Die folgenden Zustände haben die gleiche Normalform wie die oberen Zustände. Laut Lemma 3.13 sind dann die Zustände

$$\langle B_1', H_1' \wedge H_2'^c \wedge H_4'^c, C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle \text{ und}$$
$$\langle B_2', H_1'^c \wedge H_3'^c \wedge H_4', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle.$$

zusammenführbar.
Dies ist ein Widerspruch zu der Annahme, daß die varianten Zustände

$$\langle B_1, H_1 \wedge H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_1^c \wedge H_3^c \wedge H_4, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle$$

nicht zusammenführbar sind.

5) Sei $(\langle B_1, H_1 \wedge H_3 \wedge H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle, \langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \mathcal{T}, \mathcal{V}\rangle)$ das nicht-zusammenführbare kritische Paar, das aus einer Propagationsregel und einer Simpagationsregel stammt:

$$R_1 \ @ \ H_1, H_3 \ \Rightarrow \ C_1 \mid B_1$$
$$R_2 \ @ \ H_2 \backslash H_4 \ \Leftrightarrow \ C_2 \mid B_2,$$

wobei H_3 und H_5 unifizierbar sind, H_5 ein Konjunkt von $H_2 \wedge H_4$ und $H_2 \wedge H_4 = H_2^c \wedge H_4^c \wedge H_5$. Das kritische Paar ist nicht zusammenführbar, d.h. die Zustände

$$\langle B_1, H_1 \wedge H_3 \wedge H_2^c \wedge H_4^c, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \mathcal{T}, \mathcal{V}\rangle$$

sind nicht zusammenführbar, wobei $\mathcal{T}$ aus dem Token $\{R_1 @ H_1 \wedge H_3\}$ besteht, wenn er bezüglich $H_1 \wedge H_2$ nicht überflüssig ist, ansonsten ist $\mathcal{T}$ leer.
Sei $\langle \top, H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}'\rangle$ eine frische Variante von $\langle \top, H_1 \wedge H_2^c \wedge H_4^c \wedge H_3, C_1 \wedge C_2, \{R_1 @ H_1 \wedge H_3\}, \mathcal{V}\rangle$. Die Anwendung von R_1 und R_2 auf den Zustand $\langle \top, H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}'\rangle$ resultiert in folgenden Zuständen

$$\mathcal{N}(\langle B_1'', H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}\rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2'', H_1' \wedge H_2', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_3'), \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}\rangle),$$

die wegen der lokalen Konfluenz von P zusammenführbar sind. Nach der Propagation der Gleichheiten und mit der gleichen Argumentation wie im Falle (1) sind die Zustände

$$\mathcal{N}(\langle B_1', H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}'\rangle) \text{ und} \tag{4.9}$$
$$\mathcal{N}(\langle B_2', H_1' \wedge H_2', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_3'), \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}'\rangle) \tag{4.10}$$

zusammenführbar.
Die folgenden Zuständen haben die gleiche Normalform wie die Zustände 4.9 und 4.10

$$\langle B_1', H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle \text{ und}$$
$$\langle B_2', H_1' \wedge H_2', C_1' \wedge C_2', \{R_1 @ H_1' \wedge H_3'\}, \mathcal{V}'\rangle.$$

Man unterscheidet jetzt zwei Fälle:

– $\{R_1@H_1' \wedge H_3'\}$ ist bezüglich $H_1' \wedge H_2'$ überflüssig, dann sind die Zustände

$$\mathcal{N}(\langle B_1', H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \emptyset, \mathcal{V}'\rangle) \text{ und}$$
$$\mathcal{N}(\langle B_2', H_1' \wedge H_2', C_1' \wedge C_2' \wedge (H_2'' \wedge H_4'') \doteq (H_2'^c \wedge H_4'^c \wedge H_3'), \emptyset, \mathcal{V}'\rangle)$$

laut Lemma 4.50 ebenfalls zusammenführbar.
Dies ist ein Widerspruch zu der Annahme, daß die varianten Zustände

$$\langle B_1, H_1 \wedge H_2^c \wedge H_4^c \wedge H_3, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \mathcal{T}, \mathcal{V}\rangle$$

nicht zusammenführbar sind, wobei $\mathcal{T} = \emptyset$, da $\{R_1@H_1 \wedge H_3\}$ bezüglich $H_1 \wedge H_2$ überflüssig ist ($\{R_1@H_1' \wedge H_3'\}$ ist bezüglich $H_1' \wedge H_2'$ überflüssig.).

– $\{R_1@H_1' \wedge H_3'\}$ ist bezüglich $H_1' \wedge H_2'$ nicht überflüssig, dann besteht auch $\mathcal{T}$ aus dem Token $R_1@H_1 \wedge H_3$. Also sind die Zustände

$$\langle B_1', H_1' \wedge H_2'^c \wedge H_4'^c \wedge H_3', C_1' \wedge C_2', \emptyset, \mathcal{V}'\rangle \text{ und}$$
$$\langle B_2', H_1' \wedge H_2', C_1' \wedge C_2', \{R_1@H_1' \wedge H_3'\}, \mathcal{V}'\rangle$$

zusammenführbar. Dies ist ein Widerspruch zu der Annahme, daß die varianten Zustände

$$\langle B_1, H_1 \wedge H_2^c \wedge H_4^c \wedge H_3, C_1 \wedge C_2, \emptyset, \mathcal{V}\rangle \text{ und}$$
$$\langle B_2, H_1 \wedge H_2, C_1 \wedge C_2, \{R_1@H_1 \wedge H_3\}, \mathcal{V}\rangle$$

nicht zusammenführbar sind. □

4.4 Der Konfluenztester für CHR-Programme

Das Korollar 4.15 legt ein Verfahren nahe, das aus der syntaktischen Untersuchung eines terminierenden CHR-Programmes die Konfluenz entscheiden kann. Die Idee vom Konfluenztester ist, daß alle kritischen Paare eines CHR-Programmes auf ihre Zusammenführbarkeit untersucht werden. Sind alle Paare zusammenführbar, so ist das Programm konfluent, im anderen Fall nicht. Ein Vorteil dieses Konfluenztesters ist, daß nicht nur die Frage beantwortet wird, ob ein gegebenes CHR-Programm konfluent ist oder nicht. Falls ein CHR-Programm nicht konfluent ist, identifiziert der Konfluenztester anhand der nichtzusammenführbaren kritischen Paare sofort die problematischen Regeln. In diesem Abschnitt wird die Implementierung des Verfahrens beschrieben.

Die hier vorgestellte prototypische Implementierung vom Konfluenztester setzt auf der Prolog-Plattform ECL^{i}PS^e [ACD+94] auf. Diese Implementierung erweitert den in [Mar96] vorgestellten Konfluenztester auf Behandlung der Propagationsregeln.

4.4.1 Konfluenz von Constraintsymbolen

Wenn ein terminierendes CHR-Programm nicht konfluent ist, dann hat das Programm kritische Paare, die nicht zusammenführbar sind. Diese Paare lokalisieren dann den Grund der Nicht-Konfluenz.

Bei größeren CHR-Programmen kann man Schwachpunkte genauer lokalisieren und verbessern, wenn bei der Analyse nicht global das gesamte Programm auf Konfluenz getestet wird, sondern die Konfluenz von einzelnen Constraintsymbolen selektiv untersucht wird. Bei Constraintsymbolen in einem terminierenden Programm unterscheidet man daher zwei Arten von Konfluenz:

Definition 4.51 Ein Constraintsymbol p heißt *partiell konfluent*, wenn die kritischen Paare aller Regeln, die p im Kopf enthalten, zusammenführbar sind.
Ein Constraintsymbol p heißt *total konfluent*, wenn p partiell konfluent ist, und wenn alle Constraintsymbole q partiell konfluent sind, wobei q ein Constraintsymbol ist, auf das sich p abstützt.
Eine Menge C von Constraintsymbolen heißt *total (partiell) konfluent*, wenn alle Constraintsymbole in C total (partiell) konfluent sind.

Dazu ist folgende rekursive Definition nötig:

Definition 4.52 Ein Constraintsymbol p *stützt sich auf Constraintsymbol* q *direkt ab*, wenn es eine Regel gibt, in der p im Kopf und q im Rumpf auftritt.
Ein Constraintsymbol p *stützt sich auf Constraint* q *ab*, wenn sich p direkt auf q abstützt, oder wenn es im Rumpf einer Regel, in deren Kopf p auftritt, ein Constraintsymbol q' gibt, das sich auf q abstützt.

Der folgende Satz zeigt Zusammenhänge zwischen den bisher definierten Begriffen auf und rechtfertigt damit die Definitionen 4.51 und 4.52.

Satz 4.53 Für jedes terminierende CHR-Programm P und die Menge C der Constraintsymbole, die in P definiert werden (also in mindestens einem Regelkopf vorkommen), gilt:

1. C ist total konfluent, wenn C partiell konfluent ist.

2. P ist konfluent, wenn C total konfluent ist.

Beweis:

1. Die erste Aussage folgt unmittelbar aus Definition 4.51.

2. Wenn C total konfluent ist, dann sind alle Constraintsymbole aus C total konfluent. Nach Definition sind dann die kritischen Paare aller Regeln zusammenführbar, da C alle Constraintsymbole enthält, die in P definiert sind. Also ist P lokal konfluent. Da P terminierend ist, ist es auch konfluent.

Mit diesem Satz ist die Herangehensweise bei der Analyse von CHR-Programmen die folgende: Mit dem Konfluenztester werden Constraintsymbole identifiziert, die nicht partiell konfluent sind. Die verantwortlichen kritischen Paare geben einen genauen Hinweis auf den Schwachpunkt.

4.4.2 Implementierung

Zustandsrepräsentation

Die Zustandsrepräsentation der Implementierung orientiert sich an derjenigen der operationalen Semantik. Ein Zustand wird mit einem Fakt `state/5` dargestellt; die Argumente entsprechen genau den Elementen des Zustandstupels der operationalen Semantik. Zur Darstellung des Zielspeichers und der Constraintspeicher werden allerdings Listen benutzt, da Listen von Prolog besser unterstützt werden als Konjunktionen.

Die Entscheidung der partiellen Konfluenz

`partial` entscheidet die partielle Konfluenz für ein gegebenes benutzerdefiniertes Constraintsymbol. `partial(Name/Arity)` untersucht alle kritischen Paare für das Constraintsymbol mit Namen `Name` und Stelligkeit `Arity` und gibt diejenigen kritischen Paare aus, die nicht zusammenführbar sind.

`partial` stützt sich im wesentlichen auf `joinable`, das die Zusammenführbarkeit für zwei Zustände entscheidet. `joinable` benutzt `expand`, das die Regeln des CHR-Programms auf einen Zustand anwendet, wie es der CHR-Interpreter tun würde, mit dem Unterschied, daß `expand` wiedererfüllbar ist. Zum Vergleich der Endzustände von Berechnungen stützt sich `joinable` auf `equivalent_states`. `expand` stützt sich auf `solve`, `introduce`, `simplify`, `propagate` und `simpagate`, die die in der operationalen Semantik definierten Zustandsübergänge aus Abschnitt 4.3.1 implementieren.

Die erste Klausel von `partial` holt zunächst ein kritisches Paar für das spezifizierte Constraint. Sind schon alle kritischen Paare behandelt worden, dann terminiert `partial` erfolgreich mit der zweiten Klausel.

```
partial(Name/Arity) :-

    % get a critical pair for the specified constraint
    critical_pair(Name1, Name2, Shared_atoms,
                  Guard,  B1, H1, B2, H2, Global_vars),

    % prove that the critical pair is not joinable
    not joinable(state(B1, H1, [], Guard, Global_vars),
                 state(B2, H2, [], Guard, Global_vars)),

    % display the critical pair
    display_critical_pair(Name1, Name2, Shared_atoms,
                          Guard, B1, H1, B2, H2, Global_vars),

    % examine next critical pair
    fail.

partial(_).
```

Die Entscheidung der totalen Konfluenz

Das Prädikat `total` eruiert zuerst alle Prädikate, auf die sich das angegebene Prädikat stützt. Danach wird `partial` für all diese Prädikate und das angegebene Prädikat aufgerufen.

```
total(Pred/Arity):-

    % gather all base predicates
    based_upon(Pred/Arity, [], BasePreds),
    union([Pred/Arity], BasePreds, Preds),

    % test all base predicates including Pred itself
    checklist(partial, Preds).
```

Die Entscheidung der Zusammenführbarkeit

`joinable(S, T)` ist genau dann erfolgreich, wenn `S` $\downarrow$ `T`. Zwei Zustände `S` und `T` sind trivialerweise zusammenführbar, wenn sie syntaktisch gleich sind. Diesen Fall erledigt die erste Klausel.

Eine Implementierung von `joinable` muß berücksichtigen, daß durch das Erzeugen einer Berechnung für einen Zustand neue Bindungen erzeugt werden können, die sich auf den anderen Zustand übertragen. Für eine korrekte Implementierung von `joinable` ohne die oben erwähnten Seiteneffekte ist es daher notwendig, von beiden Zuständen zunächst Varianten mit frischen Variablen zu bilden. Deswegen erstellt die zweite Klausel zunächst Kopien von `S` und `T` mit `copy_term`, um dann mit `expand` endliche Berechnungen für die Kopien zu erzeugen. Die Ergebnisse der Berechnungen werden mit `variants` verglichen. Sind die Ergebnisse keine Varianten voneinander, dann erfolgt Rücksetzen nach `expand`, so daß im schlechtesten Fall, wenn `S` und `T` nicht zusammenführbar sind, die Ergebnisse aller Berechnungen für die Kopie von `S` mit den Ergebnissen aller Berechnungen für die Kopie von `T` verglichen werden. Gibt es keine weiteren Berechnungen für die Kopien, dann scheitert `joinable`.

```
joinable(S, T) :-
    S == T,
    !.

joinable(S, T) :-
    % avoid side effects
```

```
    copy_term(S, U),
    copy_term(T, V),

    % decide joinability
    expand(U, W),
    expand(V, X),
    variants(W, X).
```

Der Metainterpreter

Die meisten Constraintlöser, die in CHR geschrieben sind, verwenden vordefinierte Prolog-Prädikate im Wächter oder im Rumpf der Regeln. Zum Beispiel findet man in der CHR-Bibliothek einen Constraintlöser mit folgenden Regeln:

```
transitivity1 @ X < Y ∧ Y < Z  ⇒  X ≠ Y ∧ Y ≠ Z | X < Z.
transitivity2 @ X ≤ Y ∧ Y < Z  ⇒  X ≠ Y ∧ Y ≠ Z | X < Z.
transitivity3 @ X < Y ∧ Y ≤ Z  ⇒  X ≠ Y ∧ Y ≠ Z | X < Z.
```

Die Anwendungen des Prolog-Prädikats $\neq$ sind dabei notwendig, um Nichtterminierung zu verhindern.

Für den Konfluenztester stellen Anwendungen von Prolog-Prädikaten ein Problem dar. In der Umgebung, in der ein Konfluenztest stattfindet, sind in aller Regel viel weniger Variablen gebunden als in realen Anwendungen des betrachteten Constraintlösers. Die Argumente für Prolog-Prädikate, die während eines Konfluenztests ausgewertet werden müssen, sind deshalb oft nur unzureichend instanziiert. Eine naive Auswertung von Prolog-Zielen, die während eines Konfluenztests auftreten, kann zu Prolog-Fehlermeldungen führen, die Terminierung des Konfluenztests gefährden oder ein nicht korrektes Ergebnis zur Folge haben. So führt z.B. die Anfrage `1 < Z` in konventionellen Prolog-Systemen zu einer Fehlermeldung. Die Anfrage `checklist(integer, X)` terminiert nicht. Die Anfrage `var(X)` reduziert zu `true`, obwohl gar nicht sicher ist, daß `X` für eine Variable steht. Generell muß davon ausgegangen werden, daß eine Variable für einen beliebigen Term steht. So kann `A == A` zwar zu *true* reduziert werden, aber `A == B` kann nicht zu *false* reduziert werden, da `A` und `B` in der realen Anwendung auch für den gleichen Term stehen können. Es ist also nicht möglich, Prolog-Pädikate direkt von Prolog auswerten zu lassen; zur Sicherstellung von Korrektheit und Terminierung ist es notwendig, eine Metainstanz einzuschalten, die in Prolog-Zielen vorkommende Variablen als Metavariablen betrachtet.

Der Konfluenztester bemüht zur Auswertung von Prolog-Prädikaten einen Metainterpreter, der viele vordefinierte Prädikate korrekt behandelt. Auch benutzerdefinierte Prädikate werden korrekt behandelt, doch ist hier die Terminierung nicht in allen Fällen sichergestellt. Da der Metainterpreter Ziele isoliert betrachtet, ist er aber nicht vollständig, d.h. Widersprüche, wie z.B. `A < B` $\wedge$ `A > B`, werden nicht erkannt. Wegen dieser Schwäche legt `joinable` die verzögerten Ziele dem Benutzer zur interaktiven Vereinfachung vor, so daß unerfüllbare kritische Paare sicher erkannt werden können. Eine ausführliche Beschreibung des Metainterpreters findet sich in [Mar96].

Kapitel 5

Vervollständigung von CHR-Programmen

Für ein terminierendes Termersetzungssystem ist die Konfluenz eine entscheidbare Eigenschaft. Oft ist allerdings ein vorgegebenes System nicht konfluent, obwohl ein äquivalentes konfluentes Termersetzungssystem existiert, d.h. ein konfluentes System, das die gleiche Äquivalenzrelation erzeugt. Die Vervollständigung nach Knuth und Bendix [KB70] versucht ein solches Regelsystem zu erzeugen, ausgehend von einem terminierenden, aber nicht konfluenten System. Das Vervollständigungsverfahren von Knuth und Bendix kann auch verwendet werden, um ein Gleichungssystem in ein äquivalentes Regelsystem zu transformieren, das terminierend und konfluent ist.

Dieses Verfahren hat sich als wichtig erwiesen. Der erste Grund ist, daß dieses Verfahren in vielen Fällen anwendbar ist und ein terminierendes, konfluentes Regelsystem liefert. Ein weiterer Grund für die Wichtigkeit des Vervollständigungverfahrens von Knuth und Bendix besteht darin, daß man das Verfahren in abgewandelter Form für andere Probleme vewenden kann: Es findet unter anderem Verwendung bei induktiven Beweisen [KM86] und für das Theorembeweisen in der Prädikatenlogik [HD83].

Dieser Abschnitt stellt eine spezialisierte Form der Knuth-Bendix-Vervollständigung vor, um ein nicht-konfluentes CHR-Programm zu einem äquivalenten konfluenten zu transformieren. Diese Äquivalenz gilt aber nur für die Klasse der bereichsbeschränkten CHR-Programme, wo alle Variablen, die im Rumpf und Wächter einer Regel vorkommen, auch in ihrem Kopf vorkommen müssen. Dies ist keine starke Einschränkung, da die meisten existierenden (25) Constraintlöser diese Eigenschaft erfüllen.

Die Idee ist dabei, lokale Konfluenz dadurch herzustellen, daß nicht-zusammenführbare kritische Paare als neue Regeln mit in das CHR-Programm aufgenommen werden. Natürlich setzt dieses Vervollständigungsverfahren eine Ordnung voraus, bezüglich der die Regeln gerichtet werden, damit es erfolgreich mit einem terminierenden konfluenten CHR-Programm hält. Dadurch kommen wir jetzt zu der zweiten wichtigen Eigenschaft von Constraintlösern und zwar der Terminierung, die im nächsten Abschnitt kurz besprochen wird.

5.1 Terminierungsordnungen von CHR-Programmen

In den letzten Abschnitten haben wir gezeigt, daß die Konfluenz entscheidbar ist, wenn man sich auf terminierende CHR-Programme beschränkt: Zum einen kann man nur bei terminierenden CHR-Programmen aus der lokalen Konfluenz die Konfluenz folgern. Zum anderen stellt nur die Terminierung sicher, daß die Zusammenführbarkeit der kritischen Paare entscheidbar ist. Außerdem ist die Terminierung unabhängig von der Behandlung der Konfluenz eine notwendige Eigenschaft von Constraintlösern, da man bei der Vereinfachung von Constraints eine Antwort in endlicher

Zeit erwartet. Die Terminierung ist aber in vielen Fällen sehr schwierig sicherzustellen und vor allem nachzuweisen.

Der erste Schritt ist in Abschnitt 4.3.1 getan. Dort wurde eine neue operationale Semantik von CHR vorgestellt, die die triviale Nichtterminierung von Propagationsregeln verhindert. In [Frü97] sind einige Kriterien angegeben, unter denen ein CHR-Programm terminierend ist.

Im folgenden gehen wir von einer Ordnung $\gg$ auf Konjunktionen von Constraints aus, die auf einer Ordnung auf Constraintatomen $>$ beruht. Um Constraintatomen vergleichen zu können, braucht man eine Präzedenz auf der Menge der Constraintsymbolen $>_{P_c}$ und eine Ordnung auf Termen $>_T$. Es reicht aus, diese Ordnungen auf benutzerdefinierten Constraints zu definieren, da nach Voraussetzung ein vordefinierter Constraintlöser terminierend und konfluent ist.

5.2 Vervollständigung à la Knuth-Bendix

Bevor wir das Vervollständigungsverfahren vorstellen, geben wir die Definitionen der Bereichsbeschränkung und der semantischen Äquivalenz von CHR-Programmen an:

Definition 5.1 Ein CHR-Regel ist *bereichsbeschränkt*, wenn alle Variablen, die in ihrem Rumpf und Wächter vorkommen auch in ihrem Kopf vorkommen. Ein CHR-Programm ist *bereichsbeschränkt*, wenn alle seine Regeln bereichsbeschränkt sind.

Beispiel 5.2 Das in Beispiel 3.1 vorgestellte CHR-Programm

```
reflexivitaet @ X ≤ X ⇔ true.
antisymmetrie @ X ≤ Y ∧ Y ≤ X ⇔ X=Y.
transitivitaet @ X ≤ Y ∧ Y ≤ Z ⇒ X ≤ Z.
```

ist bereichsbeschränkt.
Die CHR-Regel

```
durchschnitt @ X::L1 ∧ X::L2 ⇔
       durchschnitt(L1,L2,L3) ∧ X::L3.
```

aus Beispiel 4.1 ist nicht bereichsbeschränkt, da die Variable L3 im Rumpf aber nicht im Kopf vorkommt.

Definition 5.3 Seien P_1, P_2 zwei CHR-Programme und CT die zugehörige Constrainttheorie. P_1 und P_2 sind *semantisch äquivalent*, falls ihre logischen Leseweisen logisch äquivalent sind:

$$CT \models \mathcal{P}_1 \leftrightarrow \mathcal{P}_2$$

Wie oben erwähnt, beschäftigt sich die Vervollständigung mit folgendem Problem:

Eingabe: Ein nicht-konfluentes, terminierendes und bereichsbeschränktes CHR-Programm P mit Terminierungsordnung $\gg$
Gesucht: Ein konfluentes CHR-Programm P', das zu P äquivalent ist.

Wir stellen das Verfahren durch Inferenzregeln dar (Abbildung 5.1), orientiert an [BD86]. Dabei steht K für eine Menge von kritischen Paaren und P für eine Menge von CHR-Regeln.

Den Kern des Verfahrens bilden die Inferenzregeln *Generieren* und *Orientieren*. Die Regel *Generieren* erlaubt, kritische Paare bezüglich P in K aufzunehmen. Mit *Orientieren* können solche Paare zu CHR-Regeln transformiert werden, wenn sie sich mit $\gg$ orientieren lassen.

Mit den Reduktionsregeln *Löschen* und *Simplifizieren* können die Paare von K vereinfacht werden. Dabei erlaubt die Reduktionsregel *Löschen* das Streichen von „harmlosen" Paaren, d.h. die Paare, die zusammenführbar sind. Die Mehrfachanwendung von *Simplifizieren* berechnet zu jedem kritischen Paar (S_1, S_2) einen Endzustand T_1 (bzw. T_2) einer Berechnung ausgehend von S_1 bzw. S_2 und ersetzt das kritische Paar (S_1, S_2) durch (T_1, T_2). Dadurch können kritische Paare vereinfacht werden.

Generieren:

$$\frac{\begin{array}{c}(K, P)\\ (S_1, S_2) \text{ ist ein kritisches Paar}\end{array}}{(K \cup \{(S_1, S_2), P\})}$$

Orientieren:

$$\frac{\begin{array}{c}(K \cup \{(S_1, S_2)\}, P)\\ R = \mathit{orient}_{\gg}(S_1, S_2)\end{array}}{(K, P \cup R)}$$

Löschen:

$$\frac{\begin{array}{c}(K \cup \{(S_1, S_2)\}, P)\\ S_1 \downarrow S_2\end{array}}{(K, P)}$$

Simplifizieren:

$$\frac{\begin{array}{c}(K \cup \{(S_1, S_2)\}, P)\\ S_1 \mapsto S_1'\end{array}}{(K \cup \{(S_1', S_2)\}, P)}$$

$$\frac{\begin{array}{c}(K \cup \{(S_1, S_2)\}, P)\\ S_2 \mapsto S_2'\end{array}}{(K \cup \{(S_1, S_2')\}, P)}$$

Abbildung 5.1: Inferenzregeln der Vervollständigung

Definition 5.4 Sei P ein terminierendes CHR-Programm. Ein naives Vervollständigungsverfahren kann wie folgt definiert werden:

- Beginne mit dem Paar $(\emptyset, P)$ und wende die Reduktionsregeln *Generieren, Orientieren, Löschen* und *Simplifizieren* fortgesetzt an.
- Ist ein Paar $(\emptyset, P')$ erreicht, wo P' lokal konfluent ist, d.h. alle möglichen kritischen Paare sind zusammenführbar, so beende die Ableitung.

Dieses Inferenzsystem verwendet keine Kontrollstrategie zur Anwendung der Reduktionsregeln. Man kann beweisen, daß jede *faire* Kontrollstrategie zu einem terminierenden und konfluenten P' führt (oder erfolglos abbricht oder unendlich lange läuft). Unter fairer Strategie versteht man, daß

- jedes kritische Paar, das bildbar ist, gebildet wird und
- jedes kritische Paar irgendwann aus K gelöscht wird.

Es treten aber einige Probleme auf: Erstens kann man abhängig von der Kontrollstrategie überflüssige Regeln in das Programm aufnehmen, wenn man zusammenführbare kritische Paare orientiert. Diese Regeln können im Gegensatz zu der Knuth-Bendix-Vervollständigung nicht eliminiert werden. Zweitens bereitet die Orientierung von Paaren einige Schwierigkeiten, da der Ziel- und Tokenspeicher bei einem kritischen Paar nicht immer leer sind, und das führt dazu, daß Propagationsregeln anwendbar sein können und diese Information nicht in den Regeln ausgedrückt werden kann. Die Idee ist jetzt, ein konkretes Vervollständigungsverfahren anzugeben, das aufgrund des Inferenzsystems und der Fairneßbedingung korrekt ist. Es gibt natürlich mehrere Heuristiken, um die Effizienz zu steigern. Eine Möglichkeit ist, die kritischen Paare so weit wie möglich zu vereinfachen, weil sonst zu viel Information erzeugt wird, die verarbeitet werden muß. Dies geschieht im folgenden Vervollständigungsalgorithmus, der eine Spezialisierung der Inferenzregeln aus Abbildung 5.1 darstellt.

Definition 5.5 Sei P ein terminierendes CHR-Programm. Das Vervollständigungsverfahren wird wie folgt definiert:

1. Setze $i := 0$ und fange mit der Menge der CHR-Regeln P_0 und deren nicht-zusammenführbaren kritischen Paaren K_0 an.

2. Falls $K_i = \emptyset$, stoppe mit $P' = P_i$.

3. Wähle fair ein Paar (S_1, S_2) aus K_i, und wende die Reduktionsregel *Simplifizieren* auf (K_i, P_i) solange an, bis Endzustände (T_1, T_2) erreicht sind. Richte das Paar (T_1, T_2) mit $\gg$ durch die Reduktionsregel *Orientieren* zu einer Menge von Regeln R. Setze $P_{i+1} = P_i \cup R$. Ansonsten brich erfolglos ab.

4. Bilde alle kritischen Paaren zwischen den Regeln in R und zwischen einer Regel aus R und allen Regeln aus P_i durch die Reduktionsregel *Generieren*. Nimm diese kritischen Paare in K_i auf und dann elimimiere alle zusammenführbaren Paare durch die Reduktionsregel *Löschen*. Es entsteht K_{i+1}.

5. Setze $i := i + 1$ und gehe nach 2.

Wie bei Termersetzungssysteme kann dieses Vervollständigungsverfahren nicht immer erfolgreich sein. Wir unterscheiden mehrere Möglichkeiten:

- Der Algorithmus stoppt mit einem Programm P', dann ist P' konfluent, terminierend und äquivalent zu P.

- Der Algorithmus bricht erfolglos ab, wenn ein Paar in K nicht in CHR-Regeln transformiert werden kann. Dies kann erfolgen, entweder wenn eine Orientierung die Terminierung gefährden würde, oder wenn eine Orientierung nicht möglich ist, weil das Paar ausschließlich aus vordefinierten Constraints besteht.

- Der Algorithmus terminiert nicht, weil neue Regeln neue kritische Paare erzeugen, die wiederum neue Regeln erfordern und so fort. Dann ist $P' = \cup P_i$ konfluent, terminierend und äquivalent zu P.

Da nach dem oben vorgestellten Vervollständigungsverfahren nur Endzustände verglichen werden, werden wir *orient* ausschließlich für diesen Fall definieren. Endzustände sind nach Definition in Normalform. Es gibt zwei Arten von Endzuständen:

- Erfolgreiche Endzustände sind von der Form $\langle \top, C_U, C_B, \emptyset, \mathcal{V} \rangle$.

- Erfolglose Endzustände sind Zustände mit einem inkonsistenten vordefinierten Speicher. Sie werden mit $\langle \top, \top, \mathit{false}, \emptyset, \mathcal{V} \rangle$ idenitifiziert.

Die Idee von *orient* ist, Regeln in das CHR-Programm einzufügen, damit die zwei Endzutände zusammenführbar werden. Da die Terminierung aller P_i mit der vorgegebenen Terminierungsordnung $\gg$ gewährleistet wird, wie sie in Abschnitt 5.1 definiert ist, verwendet *orient* diese Ordnung, um Endzustände zu richten. Da bei Endzuständen nur der benutzerdefinierte und der vordefinierte Speicher relevant sind, werden wir einfachheitshalber *orient* nicht für ganze Zustände definieren, sondern nur für benutzerdefinierte und vordefinierte Constraints. Für den Fall $C_{U1} \neq \top$ und $C_{U1} \gg C_{U2}$ (der Fall $C_{U2} \neq \top$ und $C_{U2} \gg C_{U1}$ ist analog) bekommt man
$\mathit{orient}_{\gg}((C_{U1}, C_{B1}), (C_{U2}, C_{B2})) :=$

$$\begin{cases} \{C_{U1} \Leftrightarrow C_{B1} \mid C_{U2} \wedge C_{B2},\ C_{U2} \Rightarrow C_{B2} \mid C_{B1}\}, & \text{falls } C_{U2} \neq \top \\ \{C_{U1} \Leftrightarrow C_{B1} \mid C_{B2}\} \qquad , & \text{falls } C_{U2} = \top \text{ und } CT \models C_{B1} \leftrightarrow C_{B2} \end{cases}$$

Für den Fall, daß beide Zustände entweder leere oder unvergleichbare benutzerdefinierte Speicher haben, ist die Funktion *orient* undefiniert.

Man kann auf die Propagationsregeln verzichten, wenn ihr Rumpf ausschließlich aus *true* besteht, da eine solche Propagationsregel keine neue Information einfügt. Das folgende Beispiel zeigt, daß man nicht immer auf die Propagationsregeln verzichten kann, da es i.a. nicht ausreicht, wie bei Termersetzungssystemen nur die Simplifikationsregel einzufügen, um ein nicht-zusammenführbares kritisches Paar zusammenzuführen.

Beispiel 5.6 Sei P das folgende CHR-Programm, wobei `p`,`q` und `r` benutzerdefinierte Constraintsymbole und $\geq, \leq$ vordefinierte Constraintsymbole sind.

$$\texttt{r1 @ p(X,Y)} \Leftrightarrow \texttt{X} \geq \texttt{Y} \wedge \texttt{q(X,Y)}.$$
$$\texttt{r2 @ p(X,Y)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \wedge \texttt{r(X,Y)}.$$

P ist nicht konfluent, da das kritische Paar, das aus den Regel `r1` und `r2` entsteht

$$(\langle \texttt{q(X,Y)} \wedge \texttt{X} \geq \texttt{Y}, \top, \texttt{true}, \emptyset, [\texttt{X},\texttt{Y}]\rangle, \langle \texttt{r(X,Y)} \wedge \texttt{X} \leq \texttt{Y}, \top, \texttt{true}, \emptyset, [\texttt{X},\texttt{Y}]\rangle)$$

nicht zusammenführbar ist. Nach der Berechnung der Endzustände lautet das Paar, das man orientieren soll: $(\texttt{q(X,Y)}, \texttt{X} \geq \texttt{Y})$ und $(\texttt{r(X,Y)}, \texttt{X} \leq \texttt{Y})$. Sei $\texttt{r} >_{P_c} \texttt{q}$, dann liefert die Funktion *orient* angewandt auf das Paar folgende CHR-Regeln als Ergebnis:

$$\texttt{r3 @ r(X,Y)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \mid \texttt{q(X,Y)} \wedge \texttt{X} \geq \texttt{Y}.$$
$$\texttt{r4 @ q(X,Y)} \Rightarrow \texttt{X} \geq \texttt{Y} \mid \texttt{X} \leq \texttt{Y}.$$

Die folgenden Berechnungssequenzen zeigen, daß es nötig ist, die Propagationsregel `r4` in das Programm mitaufzunehmen, damit das obige nicht-zusammenführbare kritische Paar zusammengeführt werden kann:

$$\begin{array}{ll}
 & \langle \texttt{q(X,Y)} \wedge \texttt{X} \geq \texttt{Y}, \top, \texttt{true}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Solve}} & \langle \texttt{q(X,Y)}, \top, \texttt{X} \geq \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Introduce}} & \langle \top, \texttt{q(X,Y)}, \texttt{X} \geq \texttt{Y}, \{\texttt{r4@q(X,Y)}\}, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Propagate}} & \langle \texttt{X} \leq \texttt{Y}, \texttt{q(X,Y)}, \texttt{X} \geq \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Solve}} & \langle \top, \texttt{q(X,Y)}, \texttt{X} = \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle
\end{array}$$

Ohne die Propagationsregel hätte man den zweiten Schritt nicht durchführen können und der Endzustand wäre dann $\langle \top, \texttt{q(X,Y)}, \texttt{X} \geq \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle$, der aber keine Variante vom Endzustand der folgenden Berechnugssequenz wäre:

$$\begin{array}{ll}
 & \langle \texttt{r(X,Y)} \wedge \texttt{X} \leq \texttt{Y}, \top, \texttt{true}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Solve}} & \langle \texttt{r(X,Y)}, \top, \texttt{X} \leq \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Introduce}} & \langle \top, \texttt{r(X,Y)}, \texttt{X} \leq \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Simplify}} & \langle \texttt{q(X,Y)} \wedge \texttt{X} \geq \texttt{Y}, \top, \texttt{X} \leq \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Introduce}} & \langle \texttt{X} \geq \texttt{Y}, \texttt{q(X,Y)}, \texttt{X} \leq \texttt{Y}, \{\texttt{r4@q(X,Y)}\}, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Solve}} & \langle \top, \texttt{q(X,Y)}, \texttt{X} = \texttt{Y}, \{\texttt{r4@q(X,Y)}\}, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Propagate}} & \langle \texttt{X} \leq \texttt{Y}, \texttt{q(X,Y)}, \texttt{X} = \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle \\
\mapsto_{\textbf{Solve}} & \langle \top, \texttt{q(X,Y)}, \texttt{X} = \texttt{Y}, \emptyset, [\texttt{X},\texttt{Y}]\rangle
\end{array}$$

Das folgende Beispiel soll die Vorgehensweise des Verfahrens an einem existierenden Constraintlöser verdeutlichen.

Beispiel 5.7 Sei P das folgende CHR-Programm, das einen Ausschnitt des Booleschen Constraintlösers [Frü95, FA97b] darstellt:

```
and1 @ and(X,X,Z) ⇔ X=Z.
and2 @ and(X,Y,X) ⇔ imp(X,Y).
and3 @ and(X,Y,Y) ⇔ imp(Y,X).
and4 @ and(X,Y,Z) \ and(X,Y,Z1) ⇔ Z=Z1.

imp1 @ imp(X,X) ⇔ true.
imp2 @ imp(X,Y) ∧ imp(Y,X) ⇔ X=Y.
```

Wir wählen die folgende Terminierungsordnung:

$C_1 \gg C_2$ gdw. C_2 ein striktes Konjunkt von C_1 ist
oder falls C_1, C_2 Constraintatome sind und $C_1 > C_2$

Dabei ist $>$ wie folgt definiert: $\mathtt{and(X,Y,Z)} > \mathtt{imp(A,B)}$. Es ergibt sich folgende Vervollständigungsfolge, wobei die zu bearbeitenden Paaren unterstrichen sind. Dabei ist $\mathcal{V} = [\mathtt{X,Y,Z}]$.

$$
\begin{array}{ll}
P_0 = & P \\
K_0 = & \{\underline{(\langle \mathtt{X = Z, and(X,Y,X), true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{imp(X,Y), and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle)}, \\
& (\langle \mathtt{Z = X, and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{imp(X,Y), and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle), \\
& (\langle \mathtt{Y = Z, and(X,Y,Y), true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{imp(Y,X), and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle), \\
& (\langle \mathtt{Z = Y, and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{imp(Y,X), and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle)\}
\end{array}
$$

$$
\begin{array}{ll}
P_1 = & P \cup \{\mathtt{imp(X,Y)} \wedge \mathtt{and(X,Y,Z)} \Leftrightarrow \mathtt{imp(X,Y)} \wedge \mathtt{X = Z}\} \\
K_1 = & \{\underline{(\langle \mathtt{Y = Z, and(X,Y,Y), true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{imp(Y,X), and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle)}, \\
& (\langle \mathtt{Z = Y, and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{imp(Y,X), and(X,Y,Z), true}, \emptyset, \mathcal{V}\rangle) \\
& (\langle \mathtt{X = X} \wedge \mathtt{imp(X,Y)}, \top, \mathtt{true}, \mathcal{V}\rangle, \langle \mathtt{imp(X,Y), imp(X,Y), true}, \mathcal{V}\rangle)\}
\end{array}
$$

$$
\begin{array}{ll}
P_2 = & P_1 \cup \{\mathtt{imp(Y,X)} \wedge \mathtt{and(X,Y,Z)} \Leftrightarrow \mathtt{imp(Y,X)} \wedge \mathtt{Y = Z}\} \\
K_2 = & \{\underline{(\langle \mathtt{X = X} \wedge \mathtt{imp(X,Y)}, \top, \mathtt{true}, \mathcal{V}\rangle, \langle \mathtt{imp(X,Y), imp(X,Y), true}, \mathcal{V}\rangle)}\}
\end{array}
$$

$$
\begin{array}{ll}
P_3 = & P_2 \cup \{\mathtt{imp(X,Y)} \wedge \mathtt{imp(X,Y)} \Leftrightarrow \mathtt{imp(X,Y)}\} \\
K_3 = & \emptyset
\end{array}
$$

Das unterstrichene kritische Paar von K_0 stammt aus dem Unifizieren des Kopfes der Regel `and2` mit dem ersten Constraintatom des Kopfes der Regel `and4`. Der kritische Zustand dieser Regeln lautet:

$$\langle \top, \mathtt{and(X,Y,X)} \wedge \mathtt{and(X,Y,Z)}, \mathtt{true}, \emptyset, [\mathtt{X,Y,Z}]\rangle$$

Das zweite kritische Paar von K_0 stammt aus dem Unifizieren des Kopfes der Regel `and2` mit dem zweiten Constraintatom des Kopfes der Regel `and4`. Dieses kritische Paar wird im zweiten Schritt gelöscht, da es nach dem Einfügen der Regel `r1` zusammenführbar wird. Außerdem kommt im zweiten Schritt ein nicht-zusammenführbares kritisches Paar hinzu, das aus dem Unifizieren des Kopfes der Regel `and2` und dem zweiten Constraintatom des Kopfes der im ersten Schritt hinzugefügten Regel stammt.

Das Ergebnis der Ableitung ist $P' = P_3$:

```
and1 @ and(X,X,Z)  ⇔  X=Z.
and2 @ and(X,Y,X)  ⇔  imp(X,Y).
and3 @ and(X,Y,Y)  ⇔  imp(Y,X).
and4 @ and(X,Y,Z) \ and(X,Y,Z1)  ⇔  Z=Z1.

imp1 @ imp(X,X)  ⇔  true.
imp2 @ imp(X,Y) ∧ imp(Y,X)  ⇔  X=Y.

r1 @ imp(X,Y) ∧ and(X,Y,Z)  ⇔  imp(X,Y) ∧ X=Z.
r2 @ imp(X,Y) ∧ and(Y,X,Z)  ⇔  imp(X,Y) ∧ X=Z.
r3 @ imp(X,Y) ∧ imp(X,Y)  ⇔  imp(X,Y).
```

P' ist terminierend und konfluent.

Wir geben im folgenden einige Eigenschaften des Vervollständigungsverfahrens an. Satz 5.8 besagt, daß der Vervollständigungsalgorithmus korrekt ist: Hält das Vervollständigungsverfahren mit $(K_n, P_n) = (\emptyset, P')$, so ist P' terminierend und konfluent und äquivalent zu P. Dies gilt auch für $P = \cup P_i$, wenn der Algorithmus nicht terminiert.

Satz 5.8 Startet man den Vervollständigungsalgorithmus mit einem bereichsbeschränkten CHR-Programm P und einer Terminierungsordnung $\gg$, dann gilt (falls kein Abbruch vorkommt), daß P zu P' äquivalent ist und P' konfluent und terminierend ist.
Beweis:

- P' ist terminierend. Alle Regeln in P_0 respektieren die Terminierungsordnung, und beim Übergang $P_{i+1} = P_i \cup R$ gilt dies für die neu gebildeten CHR-Regeln in R. Also gilt es für alle P_i und insbesondere für das letzte $P_i = P'$.

- P' ist konfluent. Es reicht zu zeigen, daß alle kritischen Paare des Programms P' zusammenführbar sind. Sei (S_1, S_2) ein kritisches Paar für R_1 und R_2, wobei R_1 zur Zeit i entstanden ist und R_2 zur Zeit j und $j > i$. Dann ist das Paar (S_1, S_2) in K_j, und es wird wegen der Fairneßbedingung zu einem Zeitpunkt $k > j$ entfernt. Dann ist (S_1, S_2) in P_{k+1}, also auch in P' zusammenführbar.

- P' ist äquivalent zu P. Es reicht zu zeigen, daß P_i äquivalent zu P_{i+1} für ein beliebiges i ist. Sei (S_1, S_2) in K_i. T_1 (bzw. T_2) ist der Endzustand einer Berechnung ausgehend von S_1 (bzw. S_2). T_l hat dann die Form $\langle \top, C_{Ul}, C_{Bl}, \emptyset, \mathcal{V} \rangle$, wobei $l = 1, 2$. Dann ist $P_{i+1} = P_i \cup R$ und es reicht zu zeigen, daß $\mathcal{P}_i \cup CT \models F$, wobei $F \in \mathcal{R}$ und R die deklarative Bedeutung von R ist. Wir unterscheiden hier zwei Fälle:

  - $R = \{C_{Ul} \Leftrightarrow C_{Bl} \mid C_{Um} \wedge C_{Bm}, C_{Um} \Rightarrow C_{Bm} \mid C_{Bl}\}$, wobei $l = 1, m = 2$ oder $l = 2, m = 1$.

    Da (S_1, S_2) ein kritisches Paar ist, das aus der Anwendung zweier CHR-Regeln auf einen gemeinsamen Zustand entsteht, ist die logische Beschreibung von S_1 zu der von S_2 bezüglich $\mathcal{P}_i \cup CT$ laut Korollar 3.21 äquivalent. Es folgt dann $\mathcal{P}_i \cup CT \models \forall(\exists \bar{x}(C_{Ul} \wedge C_{Bl}) \leftrightarrow \exists \bar{y}(C_{Um} \wedge C_{Bm}))$, wobei $\bar{x}$ (bzw. $\bar{y}$) die lokalen Variablen in T_1 (bzw. T_2) sind. Da P bereichsbeschränkt ist, kommen in den Zuständen keine lokalen Variablen vor und es gilt

    $$\mathcal{P}_i \cup CT \models \forall((C_{Ul} \wedge C_{Bl}) \leftrightarrow (C_{Um} \wedge C_{Bm})). \tag{5.1}$$

    Aus 5.1 folgt dann $\mathcal{P}_i \cup CT \models \forall((C_{Ul} \wedge C_{Bl}) \leftrightarrow (C_{Um} \wedge C_{Bm} \wedge C_{Bl}))$. Daraus folgt, daß $\mathcal{P}_i \cup CT \models \forall(C_{Bl} \rightarrow (C_{Ul} \leftrightarrow (C_{Um} \wedge C_{Bm})))$ gilt.

    Aus 5.1 folgt auch $\mathcal{P}_i \cup CT \models \forall((C_{Um} \wedge C_{Bm}) \rightarrow (C_{Bm} \wedge C_{Bl}))$. Daraus folgt dann, daß $\mathcal{P}_i \cup CT \models \forall(C_{Bm} \rightarrow (C_{Um} \rightarrow C_{Bl})$ gilt.

  - $R = \{C_{Ul} \Leftrightarrow C_{Bl} | C_{Bm}\}$: Dieser Fall ist trivial. □

Der Satz gilt nur für bereichsbeschränkte CHR-Programme, wie das folgende Beispiel zeigt:

Beispiel 5.9 Gegeben sei das CHR-Programm P

```
r1 @ p ⇔ q(X).
r2 @ p ⇔ r(Y).
```

Das Programm ist nicht konfluent. Das Vervollständigungsverfahren fügt die Regel

```
r3 @ q(X) ⇔ r(Y).
```

in P hinzu, wenn $\mathtt{q} > \mathtt{r}$. $P' = \{\mathtt{r1}, \mathtt{r2}, \mathtt{r3}\}$. P und P' sind nicht äquivalent, da $\{\forall \mathtt{X} \mathtt{q(X)} \leftrightarrow \exists \mathtt{Y} \mathtt{r(Y)}\}$ aus $\mathcal{P} = \{\mathtt{p} \leftrightarrow \exists \mathtt{X} \mathtt{q(X)}, \mathtt{p} \leftrightarrow \exists \mathtt{Y} \mathtt{r(Y)}\}$ nicht folgt.

Eine weitere Eigenschaft des Vervollständigungsalgorithmus ist die folgende:

Satz 5.10 Sei P ein CHR-Programm und CT eine vollständige Theorie. Wenn das Vervollständigungsverfahren erfolglos abbricht, weil die zugehörigen Endzustände eines nicht-zusammenführbaren kritischen Paares ausschließlich aus vordefinierten Constraints bestehen, ist $\mathcal{P}$ inkonsistent.
Beweis: Seien C_{B1} und C_{B2} die vordefinierten Constraints. Nach Voraussetzung gilt, daß $\exists \bar{x}_1 C_{B1}$ und $\exists \bar{x}_2 C_{B2}$ herleitbare Constraints eines kritischen Zustandes sind. Da nach Korollar 3.21 alle herleitbaren Constraints eines Zieles logisch äquivalent sind, gilt folgendes

$$\mathcal{P} \cup CT \models \forall\, (\exists \bar{x}_1 C_{B1} \leftrightarrow \exists \bar{x}_2 C_{B2}),$$

wobei $\bar{x}_1, \bar{x}_2$ die lokalen Variablen der Endzustände sind.
Annahme: $\mathcal{P}$ ist konsistent. Dann ist $\mathcal{P} \cup CT$ ebenfalls konsistent. Da CT eine vollständige Theorie, gilt dann $CT \models \forall\ (\exists \bar{x}_1 C_{B1} \leftrightarrow \exists \bar{x}_2 C_{B2})$ (C_{B1} und C_{B2} bestehen ausschließlich aus vordefinierten Constraints). Nach der Eindeutigkeitsbedingung von $\mathcal{N}$ gilt $C_{B1} = C_{B2}$. Dies ist ein Widersprich zur Voraussetzung, daß das kritische Paar nicht zusammenführbar ist. Also ist die Annahme falsch. □

Beispiel 5.11 Sei P das CHR-Programm aus Beispiel 4.3, das das Constraintsymbol `max` mit einem Tippfehler in der zweiten Regel implementiert.

```
r1 @ max(X,Y,Z) ⇔ X≤Y | Z = Y.
r2 @ max(X,Y,Z) ⇔ Y≤X | Y = X.
```

Ein kritisches Paar für die Regeln `r1` und `r2` ist

$$(\langle \mathtt{Z} = \mathtt{Y}, \top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}, \emptyset, [\mathtt{X},\mathtt{Y},\mathtt{Z}]\rangle, \langle \mathtt{Y} = \mathtt{X}, \top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}, \emptyset, [\mathtt{X},\mathtt{Y},\mathtt{Z}]\rangle).$$

Das Vervollständigungsverfahren bricht erfolglos ab, da die Endzustände, die aus dem kritischen Paar hervorgehen ausschließlich aus vordefinierten Constraints bestehen.

$$(\langle \top, \top, \mathtt{X} = \mathtt{Y} \wedge \mathtt{Z} = \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y},\mathtt{Z}]\rangle, \langle \top, \top, \mathtt{X} = \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y},\mathtt{Z}]\rangle)$$

Die logische Leseweise des CHR-Programms P mit einer Constrainttheorie, die die Constraintsymbole $\leq$ und = definiert, ist nicht konsistent. Dies kann informell mit Hilfe des Zieles `max(1,1,0)` erklärt werden, das durch `1=0` mit der ersten Regel ersetzt wird. Dieses Constraint ist äquivalent zu *false*. Wendet man aber die zweite Regel auf das gleiche Ziel an, bekommt man das Constraint `1=1`, das zu *true* äquivalent ist.

5.3 Der Vervollständiger für CHR-Programme

Das Prädikat `complete` implementiert das in Definition 5.5 vorgestellte Verfahren. Es untersucht alle kritischen Paare auf Zusammenführbarkeit. Für diejenigen kritischen Paare, die nicht zusammenführbar sind, werden zwei Endzustände berechnet. Aus diesen werden dann Vorschläge gebildet und mit einem Auswahlvorschlag dem Benutzer präsentiert. Dieser kann einen der Vorschläge einfügen oder das nächste kritische Paar untersuchen. Wenn sich der Benutzer für einen Vorschlag entscheidet, dann werden die Regeln dieses Vorschlages eingefügt und es werden alle kritischen Paare gebildet, die sich mit den neuen Regeln ergeben. Auch diese neuen kritischen Paare werden im weiteren Verlauf der interaktiven Vervollständigung noch betrachtet.

`complete` versucht, in allen Fälle die CHR-Regeln so zu bilden, daß die Terminierung nicht gefährdet wird. Zum Beispiel wird statt der Simplifikationsregel $H_1 \Leftrightarrow H_1 \wedge H_2$ die Propagationsregel $H_1 \Rightarrow H_2$ gebildet, d.h. wenn alle Constraints des Kopfes zunächst auch im Rumpf vorkämen, dann wird eine Propagationsregel gebildet, die im Rumpf kein Constraint des Kopfs hat. `complete` implementiert also das oben vorgestellte Verfahren mit einer Erweiterung, die theoretisch noch zu behandeln ist.

Die Orientierung von Paaren braucht eine Terminierungsordnung. Dazu verwendet `complete` eine Variante der rekursiven Pfad-Ordnung (RPO) [Der82]. Die RPO erweitert eine Ordnung $>$ auf Operationssymbolen zu einer Ordnung auf Termen: $s >_{RPO} t$ gdw.

entweder	$s = f(s_1,\ldots,s_m)$ und $s_i >_{RPO} t$ oder $s_i = t$ für ein i mit $1 \leq i \leq m$
oder	$s = f(s_1,\ldots,s_m)$ und $t = g(t_1,\ldots,t_n)$ und $f > g$
	und für alle j mit $1 \leq j \leq n$ ist $s >_{RPO} t_j$
oder	$s = f(s_1,\ldots,s_m)$ und $t = g(t_1,\ldots,t_m)$
	und $\{s_1,\ldots,s_m\} \gg_{RPO} \{t_1,\ldots,t_m\}$,

wobei für Multimengen X und Y die Beziehung $X \gg_{RPO} Y$ gilt, falls $X \neq Y$ und für alle $y \in Y - X$ ein $x \in X - Y$ existiert mit $x > y$. $\gg_{RPO}$ heißt Multimengenordnung.

RPO betrachtet die Konjunktion als Operationssymbol. Für CHR macht das keinen Sinn, wenn man die Constraintspeicher als Multimengen auffaßt. Deshalb setzt der Vervollständiger direkt auf $\gg_{RPO}$ auf. Übertragen auf CHR bedeutet $C_1 \gg_{RPO} C_2$, daß es für jedes neue Constraint c_2 auf der rechten Seite der Regel $C_1 \Leftrightarrow C_2$ ein Constraint c_1 mit $c_1 >_{RPO} c_2$ auf der linken Seite gibt, das rechts nicht mehr vorkommt.

Die RPO benötigt nur eine Ordnung auf den Constraintsymbolen. Eine solche Ordnung wird mit dem Prädikat `(>os)/2` vorgegeben. Der Tester selbst definiert:

```
% built-in constraints are minimal elements
F/A '>os' G/B :-
        is_userdef_constraint_symbol(F/A),
        is_built_in_predicate_symbol(G/B).
```

Vordefinierte Constraints sind also immer kleiner als benutzerdefinierte Constraints; sie sind aber selbst unvergleichbar. Der Anwender kann diese Ordnung durch Klauseln für `>os` in seinem Constraintlöser selbst erweitern. Der Tester vervollständigt die entstehende Relation zu einer partiellen Ordnung auf den Constraintsymbolen.

Für das CHR-Programm aus Beispiel 5.7 schlägt `complete` die Regeln

```
r1 @ imp(X,Y) ∧ and(X,Y,Z)  ⇔  true | imp(X,Y) ∧ Z = X.
r2 @ imp(X,Y) ∧ and(Y,X,Z)  ⇔  true | imp(X,Y) ∧ Z = X.
r3 @ imp(X,Y) ∧ imp(X,Y)  ⇔  true | imp(X,Y).
```

vor. `complete` schlägt zusätzlich Propagationsregeln vor, die eigentlich keine neue Information einfügen, deswegen werden sie hier nicht explizit angegeben. Mit diesen Regeln ist das CHR-Programm konfluent. Hier die Ausgabe der Vervollständigung für das erste untersuchte kritische Paar:

```
The CRITICAL PAIR

(⟨X = Z,and(X,Y,X), true,∅,[X,Y,Z]⟩,⟨imp(X,Y),and(X,Y,Z), true,∅,[X,Y,Z]⟩)

has been derived from the rules

and2 @ and(X,Y,X)  ⇔  imp(X,Y)
and4 @ and(X,Y,Z) \ and(X,Y,Z1)  ⇔  Z = Z1

by sharing

and(X,Y,X)

This critical pair is UNJOINABLE!

Recomputing final states

(⟨⊤,imp(X,Y),X = Z,∅,[X,Y,Z]⟩,⟨⊤,imp(X,Y) ∧ and(X,Y,Z), true,∅,[X,Y,Z]⟩)
```

```
The rules resulting from the final states are:

(1)     r1 @ imp(X,Y) ∧ and(X,Y,Z) ⇔ true | imp(X,Y) ∧ Z = X
        prop_r0 @ imp(X,Y) ⇒ Z = X | true

The RPO given suggests that rule set (1) will not violate termination.
Type (1.) to add the first rule set or (go.) to
examine the next critical pair!
```

Kapitel 6

Anwendungen der Konfluenz und Vervollständigung

6.1 Konfluenz und die semantische Analyse von CHR

6.1.1 Konfluenz und Konsistenz von CHR-Programmen

Im letzten Kapitel haben wir gesehen, daß bei der operationalen Analyse von CHR-Programmen die Konfluenz sicherstellt, daß bei jeder möglichen Auswahlstrategie dieselbe Antwort berechnet wird, auch wenn sich die Berechnungen selber voneinander unterscheiden. Insbesondere ist die Antwort auch unabhängig von der Reihenfolge, in der die Constraints dem Löser übergeben werden. Dies ist gerade bei nebenläufigen Anwendungen von Interesse. In diesem Kapitel zeigen wir, daß die Konfluenz bei der deklarativen Analyse von CHR-Programmen garantiert, daß die deklarative Semantik eines CHR-Programmes gemeinsam mit der die vordefinierten Constraints beschreibenden Theorie eine konsistente Theorie ist.

Im folgenden wird vorausgesetzt, daß das CHR-Programm bereichsbeschränkt ist und die Constrainttheorie grundvollständig ist. Die Annahme, daß das CHR-Programm bereichsbeschränkt ist, macht den Beweis des Satzes anschaulicher. Allerdings muß für allgemeine CHR-Programme eine neue Beweistechnik gesucht werden. Die Voraussetzung, daß die Constrainttheorie grundvollständig ist, ist keine starke Restriktion, da die meisten Constrainttheorien diese Eigenschaft erfüllen (Abschnitt 2.1).

Definition 6.1 Sei P ein CHR-Programm. Die *Berechnungs-Äquivalenz* $\leftrightarrow_P^*$ ist wie folgt definiert: $S_1 \leftrightarrow_P S_2$ gdw $S_1 \mapsto S_2$ oder $S_1 \mapsfrom S_2$. Es gilt $S \leftrightarrow_P^* S'$ gdw es eine Sequenz $S_1, \ldots S_n$ gibt, sodaß gilt: S_1 ist S, S_n ist S' und $S_i \leftrightarrow_P S_{i+1}$ für alle $1 \leq i \leq n$. Wir schreiben $\leftrightarrow$ anstatt $\leftrightarrow_P$, wenn das Programm P festgelegt ist.

Man kann leicht sehen, daß es für jede Berechnungs-Äquivalenz von der Form $S_1 \leftrightarrow^* S_2$ eine Sequenz $S_1, T_1, S_2, T_2, \ldots, T_{n-1}, S_n$ mit folgender Gestalt gibt (Diese Sequenz wird in Abbildung 6.1 illustriert):

$$S \;{}^*\!\!\mapsfrom S_1 \mapsto^* T_1 \;{}^*\!\!\mapsfrom S_2 \mapsto^* \ldots \mapsto^* T_{n-1} \;{}^*\!\!\mapsfrom S_n \mapsto^* S'.$$

$$\begin{tikzcd}
 & S_1 \arrow[dl, "*"'] \arrow[dr, "*"] & & S_2 \arrow[dl, "*"'] \arrow[dr, "*"] & & \cdots & & S_n \arrow[dl, "*"'] \arrow[dr, "*"] & \\
S & & T_1 & & T_2 & \cdots & T_{n-1} & & S'
\end{tikzcd}$$

Abbildung 6.1: Berechnungs-Äquivalenz von S und S'

Lemma 6.2 Wenn P konfluent ist, dann gilt $\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \leftrightarrow_P^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ nicht.
Beweis: Durch Induktion über die Länge der Berechnungs-Äquivalenz n, daß es keine Zustände $S_1, T_1, S_2, \ldots, T_{n-1}, S_n$ gibt, sodaß

$$\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \;{}^*\!\!\leftarrow S_1 \mapsto^* T_1 \;{}^*\!\!\leftarrow S_2 \mapsto^* \ldots \mapsto^* T_{n-1} \;{}^*\!\!\leftarrow S_n \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle$$

Induktionsanfang: $\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \;{}^*\!\!\leftarrow S_1 \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ kann nicht existieren, da P konfluent ist und $\langle \top, \top, true, \emptyset, \mathcal{V}\rangle$ und $\langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ unterschiedliche Endzustände (keine Varianten) sind.
Induktionsschritt: Wir nehmen an, daß die Induktionsvoraussetzung für n gilt. Dann existiert keine Sequenz der Form

$$\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \;{}^*\!\!\leftarrow S_1 \mapsto^* T_1 \;{}^*\!\!\leftarrow S_2 \mapsto^* \ldots \mapsto^* T_{n-1} \;{}^*\!\!\leftarrow S_n \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle.$$

Wir beweisen die Behauptung für $n+1$ durch Widerspruch:
Annahme: Es gibt eine Sequenz von der Form

$$\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \;{}^*\!\!\leftarrow S_1 \mapsto^* T_1 \;{}^*\!\!\leftarrow S_2 \mapsto^* T_2 \;{}^*\!\!\leftarrow \ldots \;{}^*\!\!\leftarrow S_n \mapsto^* T_n \;{}^*\!\!\leftarrow S_{n+1} \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle.$$

Da P konfluent ist, sind die Zustände $\langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ und T_n zusammenführbar. Da aber $\langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ ein Endzustand ist, gibt es eine Sequenz ausgehend von T_n, die zu einem Endzustand von der Form $\langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ führt ($T_n \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle$). Also gilt $S_n \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle$. Es gibt also eine Sequenz von der Form

$$\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \;{}^*\!\!\leftarrow S_1 \mapsto^* T_1 \;{}^*\!\!\leftarrow S_2 \mapsto^* \ldots \mapsto^* T_{n-1} \;{}^*\!\!\leftarrow S_n \mapsto^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle.$$

Dies ist aber ein Widerspruch zu der Induktionsvoraussetzung. □

Lemma 6.3 Sei P ein CHR-Programm und CT eine grundvollständige Constrainttheorie. Wenn $\langle \top, \top, true, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, false, \emptyset, \mathcal{V}\rangle$ nicht gilt, dann ist $\mathcal{P} \cup CT$ konsistent.
Beweis: Wir beweisen die Konsistenz, indem wir eine Interpretation definieren, die ein Modell für $\mathcal{P} \cup CT$ ist.

Sei I_0 wie folgt definiert:

$$\begin{aligned} I_0 \;:=\; \{&\{G_1\theta, \ldots, G_n\theta\} \mid \langle (G_1 \wedge \ldots \wedge G_n)\theta, \top, true, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, true, \emptyset, \mathcal{V}\rangle, \text{ wobei} \\ &\theta \text{ eine Variablenbelegung für die Variablen von } G_1, \ldots, G_n \text{ ist.}\} \end{aligned}$$

Sei $I := (\bigcup I_0)$, wobei $\bigcup M$ die Vereinigung aller Elemente von M ist.
Wegen der Konsistenz und Grundvollständigkeit hat CT ein einziges Herbrandmodel

$$CM := \{C \mid CT \models C \text{ und } C \text{ ist grund}\}$$

Sei $\mathcal{I} = I \cup CM$. $false \notin \mathcal{I}$, da $\langle false, \top, true, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, true, \emptyset, \mathcal{V}\rangle$ nicht gilt. $\mathcal{I}$ ist dann eine Herbrandinterpretation von $\mathcal{P} \cup CT$. Wir zeigen jetzt, daß $\mathcal{I} \models \mathcal{P}$.

1. Fall: Sei $(H_1 \wedge \ldots \wedge H_n \Leftrightarrow C_1 \wedge \ldots \wedge C_j \mid B_1 \wedge \ldots \wedge B_m)$ eine Simplifikationsregel aus P. Wir beweisen dann, daß $\mathcal{I} \models \forall \bar{x}\; ((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow \exists \bar{y}\; (B_1 \wedge \ldots \wedge B_m)))$ gilt. Da die Regeln bereichsbeschränkt sind, reicht es zu zeigen, daß $\mathcal{I} \models \forall \bar{x}\; ((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m))$ gilt.

Um zu zeigen, daß $\mathcal{I} \models \forall \bar{x}\; (C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m)$ gilt (dies ist äquivalent zu $\mathcal{I} \models \forall \bar{x}\; ((H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j) \leftrightarrow (B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j)))$, müssen wir zeigen, daß $\mathcal{I}, \theta \models (H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \leftrightarrow B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j)$ für eine beliebige Variablenbelegung θ.

$$\begin{array}{ll} & \mathcal{I}, \theta \models H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \\ \text{gdw} & \{H_1\theta, \ldots, H_n\theta, C_1\theta, \ldots, C_j\theta\} \subseteq \mathcal{I} \\ \text{gdw} & \langle (H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \wedge G_1 \wedge \ldots \wedge G_l)\theta, \top, true, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, true, \emptyset, \mathcal{V}\rangle \\ \text{gdw} & \langle (B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j \wedge G_1 \wedge \ldots \wedge G_l)\theta, \top, true, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, true, \emptyset, \mathcal{V}\rangle \\ \text{gdw} & \{B_1\theta, \ldots, B_m\theta, C_1\theta, \ldots, C_j\theta\} \subseteq \mathcal{I} \\ \text{gdw} & \mathcal{I}, \theta \models B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j. \end{array}$$

Es gilt dann $\mathcal{I} \models \forall\,((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m))$.

2. Fall: Sei $(H_1 \wedge \ldots \wedge H_n \Rightarrow C_1 \wedge \ldots \wedge C_j \mid B_1 \wedge \ldots \wedge B_m)$ eine Propagationsregel aus P. Wir beweisen dann, daß $\mathcal{I} \models \forall\bar{x}\,((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \rightarrow \exists\bar{y}\,(B_1 \wedge \ldots \wedge B_m)))$ gilt.

Mit der gleichen Argumentation wie im Falle der Simplifikationsregeln müssen wir zeigen, daß $\mathcal{I}, \theta \models (H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \rightarrow B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j)$ für eine beliebige Variablenbelegung θ gilt.

$$
\begin{array}{ll}
 & \mathcal{I}, \theta \models H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \\
\text{dann} & \mathcal{I} \models H_1\theta \wedge \ldots \wedge H_n\theta \wedge C_1\theta \wedge \ldots \wedge C_j\theta \\
\text{dann} & \{H_1\theta, \ldots, H_n\theta, C_1\theta, \ldots, C_j\theta\} \subseteq \mathcal{I} \\
\text{dann} & \langle (H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \wedge G_1 \wedge \ldots \wedge G_l)\theta, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \\
\text{dann} & \langle (B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j \wedge H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_l)\theta, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \\
 & \langle \top, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \\
\text{dann} & \{B_1\theta, \ldots, B_m\theta, C_1\theta, \ldots, C_j\theta, G_1\theta, \ldots, G_l\theta\} \subseteq \mathcal{I} \\
\text{dann} & \{B_1\theta, \ldots, B_m\theta, C_1\theta, \ldots, C_j\theta\} \subseteq \mathcal{I} \\
\text{dann} & \mathcal{I}, \theta \models B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j
\end{array}
$$

Es gilt dann $\mathcal{I} \models \forall\bar{x}\,((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \rightarrow \exists\,(B_1 \wedge \ldots \wedge B_m)))$.

3. Fall Sei $(H_1 \wedge \ldots \wedge H_i \backslash H_{i+1} \wedge \ldots \wedge H_n \Leftrightarrow C_1 \wedge \ldots \wedge C_j \mid B_1 \wedge \ldots \wedge B_m)$ eine Simpagationsregel aus P. Wir beweisen dann, daß $\mathcal{I} \models \forall\bar{x}\,((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow \exists\bar{y}\,(H_1 \wedge \ldots \wedge H_i \wedge B_1 \wedge \ldots \wedge B_m)))$ gilt. Da die Regeln bereichsbeschränkt sind, reicht es zu zeigen, daß $\mathcal{I} \models \forall\bar{x}\,((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow H_1 \wedge \ldots \wedge H_i \wedge B_1 \wedge \ldots \wedge B_m))$ gilt.

Zu zeigen ist, daß $\mathcal{I}, \theta \models (H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \leftrightarrow H_1 \wedge \ldots \wedge H_i \wedge B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j)$ für eine beliebige Variablenbelegung θ.

$$
\begin{array}{ll}
 & \mathcal{I}, \theta \models H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \\
\text{gdw} & \{H_1\theta, \ldots, H_n\theta, C_1\theta, \ldots, C_j\theta\} \subseteq \mathcal{I} \\
\text{gdw} & \langle (H_1 \wedge \ldots \wedge H_n \wedge C_1 \wedge \ldots \wedge C_j \wedge G_1 \wedge \ldots \wedge G_l)\theta, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \\
\text{gdw} & \langle (H_1 \wedge \ldots H_i \wedge B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j \wedge G_1 \wedge \ldots \wedge G_l)\theta, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \leftrightarrow^* \\
 & \langle \top, \top, \mathit{true}, \emptyset, \mathcal{V}\rangle \\
\text{gdw} & \{H_1\theta, \ldots, H_i\theta, B_1\theta, \ldots, B_m\theta, C_1\theta, \ldots, C_j\theta\} \subseteq \mathcal{I} \\
\text{gdw} & \mathcal{I}, \theta \models H_1 \wedge \ldots \wedge H_i \wedge B_1 \wedge \ldots \wedge B_m \wedge C_1 \wedge \ldots \wedge C_j.
\end{array}
$$

Es gilt dann $\mathcal{I} \models \forall\,((C_1 \wedge \ldots \wedge C_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow H_1 \wedge \ldots \wedge H_i \wedge B_1 \wedge \ldots \wedge B_m))$.

$\mathcal{I}$ ist dann eine Interpretation, die alle Formeln aus $\mathcal{P}$ erfüllt. Es gilt dann $\mathcal{I} \models \mathcal{P}$. Daraus folgt, daß $\mathcal{I}$ ein Modell für $\mathcal{P} \cup CT$ ist, da $CM \subseteq \mathcal{I}$. □

Der folgende Satz folgt dann unmittelbar aus Lemma 6.2 und Lemma 6.3.

Satz 6.4 Sei P ein bereichsbeschränktes CHR-Programm und CT eine grundvollständige Constrainttheorie. Wenn P konfluent ist, dann ist $\mathcal{P} \cup CT$ konsistent.

6.1.2 Konfluenz und die Korrektheit und Vollständigkeit von CHR

Wir haben in dieser Arbeit versucht, zwischen erfolgreichen und erfolglosen Ableitungen nicht zu unterscheiden, da wir mit CHR daran interessiert sind, Constraints zu vereinfachen. Die Antwort *false* ist also eine befriedigende Antwort, die auf die Inkonsistenz der Constraints hindeutet. In diesem Abschnitt werden wir aber zwischen erfolgreichen und erfolglosen Ableitungen unterscheiden, um einen Zusammenhang zu den Ergebnissen von NCLP-Sprachen herzustellen.

Der folgende Satz gibt ein besseres Vollständigkeitsergebnis als Satz 3.22 an, wenn ein CHR-Programm konfluent und terminierend ist.

Satz 6.5 Sei P ein terminierendes und konfluentes CHR-Programm und G ein Ziel, dann sind folgende Aussagen äquivalent:

a) $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$.

b) G hat eine Ableitung mit einem Antwortconstraint C', so daß $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C')$ gilt.

c) Jede Ableitung von G hat ein Antwortconstraint C', so daß $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C')$ gilt.

Beweis: "a) $\Rightarrow$ b)" gilt trivialerweise laut Vollständigkeitssatz 3.22.
"b) $\Rightarrow$ c)" folgt unmittelbar wegen Konfluenz und Terminierung.
"c) $\Rightarrow$ a)" gilt laut Korrektheitssatz 3.20. $\square$

Satz 6.7 gibt eine Bedingung für die Existenz von endlich gescheiterten Ableitungen vorausgesetzt, daß die Ziele die folgende Eigenschaft haben.

Definition 6.6 Ein Ziel ist *datenausreichend*, falls es eine endliche Ableitung mit einem Endzustand hat, dessen benutzerdefinierter Speicher leer ist.

Diese Eigenschaft garantiert, daß es eine Ableitung von einem Ziel mit einem Antwortconstraint bestehend aus ausschließlich vordefinierten Constraints gibt. Diese Eigenschaft ist die gleiche, wie die in [Mah87] vorausgesetzte (siehe Definition 2.19), aber diese Definition ist expliziter[1].

Satz 6.7 Sei P ein bereichbeschränktes und konfluentes CHR-Programm, CT eine grundvollständige Theorie und G ein datenausreichendes Ziel. Wenn $\mathcal{P} \cup CT \models \neg\exists G$ gilt, dann hat G eine endlich gescheiterte Ableitung.
Beweis: G hat eine Ableitung mit einem Antwortconstraint C bestehend aus ausschließlich vordefinierten Constraints, da G nach Voraussetzung datenausreichend ist.
Laut Satz 3.20 gilt folgendes:

$$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G).$$

Aus $\mathcal{P} \cup CT \models \neg\exists G$ folgt $\mathcal{P} \cup CT \models \forall\ (\mathit{false} \leftrightarrow G)$. Daher gilt

$$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow \mathit{false}).$$

Da P konfluent ist, ist $\mathcal{P}$ konsistent[2] (Satz 6.4). Außerdem definiert P keine vordefinierten Constraints und C besteht ausschließlich aus vordefinierten Constraints. Daher gilt folgendes:

$$CT \models \forall\ (C \leftrightarrow \mathit{false}).$$

Da C ausschließlich aus vordefinierten Constraints besteht, muß C *false* sein. $\square$

Das folgende Korollar ist ein Korrektheits- und Vollständigkeitsergebnis vom „Endlichen Scheitern". Es ist eine unmittelbare Konsequenz aus den Sätzen 6.5, 6.4 and 6.7.

Korollar 6.8 Sei P ein bereichbeschränktes, terminierendes und konfluentes CHR-Programm, CT eine grundvollständige Theorie und G ein datenausreichendes Ziel, dann sind folgende Aussagen äquivalent:

a) $\mathcal{P} \cup CT \models \neg\exists G$

b) G hat eine endlich gescheiterte Ableitung.

c) Jede Ableitung von G scheitert endlich.

[1] Kommunikation mit M. Maher, Email, Januar 1997.
[2] Der Beweis kann folglich für konsistente Programme durchgeführt werden.

Diese Ergebnisse sind ähnlich wie die für ALPS [Mah87], wenn auch ein ALPS-Programm eine unterschiedliche deklarative Semantik hat (basierend auf Clarkscher Vervollständigung) und eine unterschiedliche operationale Semantik.

Als Zusammenfassung von diesem Abschnitt geben wir tabellarisch in Abbildung 6.2 einen Vergleich von den verschiedenen Korrektheits- und Vollständigkeitssätzen für erfolgreiche Ableitungen (EA) und endliches Scheitern (ES) von NCLP-Sprachen,[3] wie sie in [JM94] vorgestellt wurden, und CHR. Sowohl für deterministische NCLP-Programme als auch für konfluente CHR-Programme kann man Korrektheits- und Vollständigkeitsergebnisse angeben. Die Klasse der konfluenten Programme ist aber echt größer als die der deterministischen Programme, wie in Abschnitt 8.2 gezeigt wird.

	NCLP	CHR	Determ. NCLP	konfl. CHR
Korrektheit (EA)	Ja	Ja	Ja	Ja
Vollständigkeit (EA)	Nein	Ja	Ja	Ja
Korrektheit (ES)	Ja	Ja	Ja	Ja
Vollständigkeit (ES)	Nein	Nein	Ja	Ja

Abbildung 6.2: Korrektheits- und Vollständigkeitsergebnisse von NCLP und CHR

6.2 Konfluenz und Kombination von CHR-Programmen

Die in diesem Abschnitt zugrundeliegende Problematik ist die gleichzeitige Verwendung zweier (oder mehreren) Constraintlöser. Einige Constraints sind möglicherweise in beiden Constraintlösern definiert, und der Benutzer muß wissen, ob die zwei Definitionen der Constraints verträglich sind. Ein naheliegender Ansatz ist es, zu untersuchen, ob die Vereinigung der zwei Programme konfluent ist. Ist das der Fall, so sind die zwei Implementierungen verträglich in dem Sinne, daß sie bei der gleichen Eingabe ein identisches Ergebnis erzeugen.

Bei diesem Ansatz geht man davon aus, daß die einzelnen Löser, die man untersucht, terminierend und konfluent sind. Damit ist leider noch nicht die Terminierung der Kombination der Löser sichergestellt, die zusätzlich zur Konfluenz untersucht werden muß, wie das folgende einfache Beispiel zeigt.

Beispiel 6.9 Sei P_1 das folgende CHR-Programm, das das Constraint `p` mit Hilfe von `q` definiert:

```
r1 @ p ⇔ q.
```

und sei P_2 das CHR-Programm, das das Constraint `q` mit Hilfe von `p` definiert.

```
r2 @ q ⇔ p.
```

Beide CHR-Programme sind terminierend, aber ihre Kombination nicht mehr.

Wir werden die Verträglichkeit von CHR-Programmen (Constraintlösern) anhand eines Beispieles illustrieren:

Beispiel 6.10 Dieses Beipiel geht von dem Constraint `range` aus, das in zwei verschiedenen Constraintlösern definiert ist. `range(X,Min,Max)` bedeutet, daß `X` zwischen `Min` und `Max` liegen muß. Jede der zwei folgenden Definitionen von `range` enthält Redundanzen, die man weglassen könnte. Um das Beispiel nicht zu trivial zu gestalten, wurden die vorgestellten Implementierungen ausgewählt:

```
range(X,Min,Max) ⇔ maximum(Min,Max,Max) ∧ maximum(X,Min,X) ∧ maximum(X,Max,Max).
```

[3] Die deklarative Semantik von diesen Sprachen ist verschieden von der von CHR, sie basiert auf Clarkscher Vervollständigung.

und

```
range(_X,Min,Max) ⇔  Max < Min  | false.
range(X,Min,Max)  ⇔  Min ≤ Max | Min ≤ X ∧ X ≤ Max.
```

Um die Verträglichkeit zu prüfen, werden die zwei Löser in einem Löser integriert. Nun wird mit `partial(range)` die partielle Konfluenz des Constraintsymboles `range` überprüft. Der Konfluenztester untersucht dann drei nicht-triviale kritische Paare, die aus der Kombination der drei Regeln entstehen. Interessant sind die Paare, die aus Regeln unterschiedlicher Löser entstehen, da die Löser allein nach Voraussetzung konfluent sind. Der Konfluenztester liefert das folgende Ergebnis:

```
[eclipse 2]: partial(range).

No (more) unjoinable critical pairs.
yes.
```

Alle kritischen Paare sind also zusammenführbar, wenn man voraussetzt, daß die zur Verfügung gestellten Constraintsymbole vollständig implementiert sind. Man sieht auch sofort, daß die Terminierung bei der Kombination der Löser erhalten bleibt. Damit ist die Verträglichkeit sichergestellt.

6.3 Vervollständigung zur Erweiterung von Constraintlösern

Die Konfluenz eines CHR-Programmes kann u.a. verletzt werden, wenn ein Benutzer vergißt, einige Fälle zu betrachten. Wir werden in diesem Abschnitt anhand eines Beispieles sehen, wie das Vervollständigungsverfahren sinnvolle CHR-Regeln vorschlägt, die diese Fälle abdecken können.

Beispiel 6.11 Gegeben sei das CHR-Programm aus Beispiel 4.42 mit einer zusätzlichen Regel `r5`, die den Zusammenhang zwischen dem Constraintsymbol $\leq$ und dem Constraintsymbol $<$ beschreibt, wobei $\neq$ ein vordefiniertes Constraintsymbol ist.

```
r1 @ X ≤ X ⇔ true.
r2 @ X ≤ Y ∧ Y ≤ X ⇔ X = Y.
r3 @ X ≤ Y ∧ Y ≤ Z ⇒ X ≤ Z.
r4 @ X ≤ Y \ X ≤ Y ⇔ true.
r5 @ X ≤ Y ⇔ X ≠ Y | X < Y.
```

Mit folgender Terminierungsordnung

$C_1 \gg C_2$ gdw. C_2 ein striktes Konjunkt von C_1 oder $C_1 = \mathtt{X} \leq \mathtt{Y}$ und $C_2 = \mathtt{X} < \mathtt{Y}$,

liefert das Vervollständigungsverfahren u.a. folgende Regeln:

```
r6 @ X < Y ∧ Y < X  ⇔ X ≠ Y | false.
r7 @ X < Y ∧ X < Y  ⇔  X ≠ Y | X < Y.
```

Die Regel `r6` stammt aus den Regeln `r2` und `r5`. Der kritische Zustand der beiden Regeln ist $\langle \top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}, \mathtt{X} \neq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle$. Das kritische Paar stammt also aus dem Unifizieren des ersten Constraintatoms des Kopfes der Regel `r2` und dem Kopf der Regel `r5`:

$$(\langle \mathtt{X} = \mathtt{Y}, \top, \mathtt{X} \neq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle, \langle \mathtt{X} < \mathtt{Y}, \mathtt{Y} \leq \mathtt{X}, \mathtt{X} \neq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle).$$

Die Regel `r7` stammt aus den Regeln `r4` und `r5`: Der kritische Zustand der beiden Regeln lautet $\langle \top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \leq \mathtt{Y}, \mathtt{X} \neq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle$. Das kritische Paar ist

$$(\langle \top, \mathtt{X} \leq \mathtt{Y}, \mathtt{X} \neq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle, \langle \mathtt{X} < \mathtt{Y}, \mathtt{Y} \leq \mathtt{X}, \mathtt{X} \neq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle).$$

Obwohl die Wächter der CHR-Regeln `r6` und `r7` redundant sind, sind sie sinnvolle CHR-Regeln, um die der Constraintlöser erweitert werden sollte. Regel `r6` drückt die Aymmetrie von $<$ aus. Regel `r7` drückt die Idempotenz von $\wedge$ für das Constraintsymbol $<$ aus.

Eine Regel, die die Irreflexivität von $<$ ausdrückt, kann nicht generiert werden, da Regel `r5` verlangt, daß $\mathtt{X} \neq \mathtt{Y}$ gelten soll. Wenn wir aber die Regel

$$\mathtt{r8}\ @\ \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} < \mathtt{X} \Leftrightarrow \mathtt{false}.$$

einfügen, liefert das Vervollständigungsverfahren folgende Simplifilationsregel

$$\mathtt{r9}\ @\ \mathtt{X} < \mathtt{X} \Leftrightarrow \mathtt{false}.$$

Mit der Irreflexivitätsregel `r9` kann das kritische Paar

$$(\langle \mathtt{X} < \mathtt{X}, \top, \mathtt{true}, \emptyset, [\mathtt{X}]\rangle, \langle \mathtt{false}, \top, \mathtt{true}, \emptyset, [\mathtt{X}]\rangle),$$

das aus `r1` und `r8` stammt, zusammengeführt werden.

6.4 Vervollständigung zur Spezialisierung von Constraintlösern

Wir werden in diesem Abschnitt einen Mechanismus vorstellen, der Constraintlöser für gegebene Ziele spezialisieren kann. Dieser Mechanismus basiert auf einer Variante des Vervollständigungsverfahrens, die *lineare Vervollständigung* [DJ84, BH92]. Sie ist der linearen Resolution [KK71] ähnlich. Man beschränkt sich auf Vervollständigungsfolgen folgender Art: Die jeweils vorangegangene abgeleitete CHR-Regel ist gleichzeitig die Regel, die beim nächsten Schritt der Vervollständigungsfolge beteiligt ist.

Wir werden uns im folgenden auf die Klasse der CHR-Programme beschränken, die bei ihrer Vervollständigung das Einfügen von Propagationsregeln nicht benötigen. Wir gehen dann von einem konfluenten CHR-Programm aus und fügen eine *Anfrage-Regel* ein. Diese ist eine Simplifikationsregel, deren Kopf aus dem Ziel besteht und deren Rumpf aus einem benutzerdefinierten Constraint `answer`, das als Argumente die Variablen des Zieles hat. Das lineare Vervollständigungsverfahren besteht darin, daß man, startend mit der Anfrage-Regel, solange durch Anwendung der linearen Vervollständigung neue CHR-Regeln bildet und hinzufügt, bis gegebenenfalls keine nicht-zusammenführbaren kritischen Paare gebildet werden können. Die neu hinzugefügten CHR-Regeln, die keine nicht-zusammenführbaren kritischen Paare bilden, bilden dann das Ergebnis der linearen Vervollständigung. Als Ergebnis erhält man dann einen speziellen Constraintlöser für das vorgegebene Ziel. Wir illustrieren dieses Verfahren anhand einiger Beispiele. Beim folgenden Beispiel ist das Ziel, wofür einen Constraintlöser spezialisiert wird, eine Darstellung einer endlichen Menge von Antworten.

Beispiel 6.12 Gegeben sei das CHR-Programm P, das `member` implementiert:

```
r1 @ member(E,[]) ⇔ false.
r2 @ member(E,[E|_]) ⇔ true.
r3 @ member(E,[H|L]) ⇔ E ≠ H | member(E,L).
```

Das Ziel `member(X,[1,2,3])` kann mit dem obigen Programm nicht reduziert werden. Das entsprechende Programm in Prolog

```
member(E,[E|_]).
member(E,[H|L]) :- member(E,L).
```

würde für die Anfrage `member(X,[1,2,3])` die drei Lösungen `X=1;X=2;X=3` liefern und dann scheitern, wenn man nach weiteren Lösungen fragt.

Die lineare Vervollständigung fügt zuerst die Anfrage-Regel

```
r4 @ member(A,[1,2,3]) ⇔ answer(A).
```

in das CHR-Programm P hinzu und liefert folgendes Ergebnis

```
a1 @ answer(1) ⇔ true.
a2 @ answer(2) ⇔ true.
a3 @ answer(3) ⇔ true.
a4 @ answer(A) ⇔ A ≠ 1 ∧ A ≠ 2 ∧ A ≠ 3 | false.
```

Im folgenden geben wir in Form eines Baumes an, wie diese Regeln generiert wurden:

$$
\begin{tikzcd}
 & \mathtt{r4@member(A,[1,2,3])} \Leftrightarrow \mathtt{answer(A)} \arrow[dl, "\mathtt{r4+r2}"'] \arrow[d, "\mathtt{r4+r3}"] \\
\mathtt{a1@answer(1)} \Leftrightarrow \mathtt{true} & \mathtt{r5@member(A,[2,3])} \Leftrightarrow \mathtt{A} \neq 1 | \mathtt{answer(A)} \arrow[dl, "\mathtt{r5+r2}"'] \arrow[d, "\mathtt{r5+r3}"] \\
\mathtt{a2@answer(2)} \Leftrightarrow \mathtt{true} & \mathtt{r6@member(A,[3])} \Leftrightarrow \mathtt{A} \neq 1 \wedge \mathtt{A} \neq 2 | \mathtt{answer(A)} \arrow[dl, "\mathtt{r6+r2}"'] \arrow[d, "\mathtt{r6+r3}"] \\
\mathtt{a3@answer(3)} \Leftrightarrow \mathtt{true} & \mathtt{a4@answer(A)} \Leftrightarrow \mathtt{A} \neq 1 \wedge \mathtt{A} \neq 2 \wedge \mathtt{A} \neq 3 | \mathtt{false}
\end{tikzcd}
$$

Das Einfügen von Regel `r4` in das CHR-Programm, das `member` definiert, führt dazu, daß diese Regel sowohl mit Regel `r2` als auch mit Regel `r3` nicht-zusammenführbare kritische Paare bildet. Das Vervollständigungsverfahren schlägt also die Regeln `a1` und `r5` vor, um diese kritischen Paare zusammenzuführen. Mit Regel `a1` kann man keine nicht-zusammenführbaren kritischen Paare bilden, deswegen endet die lineare Vervollständigung in diesem Zweig mit dem Einfügen der Regel `a1`. Im rechten Zweig, wo Regel `r5` eingefügt wurde, fügt die lineare Vervollständigung weitere Regeln hinzu, da Regel `r5` mit den ursprünglichen Regeln des CHR-Programms P nicht-zusammenführbare kritische Paare bildet. Dies wird dann solange durchgeführt, bis aus den neu erhaltenen Regeln und dem ursprünglichen Programm keine nicht-zusammenführbaren kritischen Paare gebildet werden können.

Die CHR-Regeln `a1`,`a2` und `a3` drücken die Antworten `X=1;X=2;X=3` aus und die vierte Regel `a4` drückt aus, daß das Ziel `member(X,[1,2,3])` keine weitere Lösung besitzt.

Ein Ziel kann aber auch eine unendliche Menge von Lösungen darstellen. Im folgenden Beispiel wird gezeigt, wie das lineare Vervollständigungsverfahren sich bei einem solchen Fall verhält.

Beispiel 6.13 Gegeben sei folgendes CHR-Programm P

```
r0 @ append([],L,L) ⇔ true.
r1 @ append([X|L1],Y,[X|L2]) ⇔ append(L1,Y,L2).
```

Dieses Programm ist konfluent. Das Ziel `append(X,[b|Y],[a,b,c|Z])` kann mit dem CHR-Programm nicht reduziert werden, da es mit keinem Kopf der obigen CHR-Regeln matcht.

`append` kann in Prolog wie folgt implementiert werden:

```
append([],L,L).
append([X|L1],Y,[X|L2]) :- append(L1,Y,L2).
```

Prolog generiert bei der Auswertung des Zieles `append(X,[b|Y],[a,b,c|Z])` eine unendliche Menge von Antworten:

```
X = [a], Y = [c|Z], Z = Z

X = [a,b,c], Y = Y, Z = [b|Y]

X = [a,b,c,X1], Y = Y, Z = [X1,b|Y]

X = [a,b,c,X1,X2], Y = Y, Z = [X1,X2,b|Y]
...
```

Das lineare Vervollständigungverfahren liefert nach dem Hinzufügen der Anfrage-Regel

```
r2 @ append(X,[b|Y],[a,b,c|Z]) ⇔ answer(X,Y,Z).
```

in P folgendes Ergebnis:

```
a1 @ answer([a],[c|E],E) ⇔ true.
a2 @ answer([a,b,c],G,[b|G]) ⇔ true.
a3 @ answer([a,b,c,H|I],J,[H|K]) ⇔ answer([a,b,c|I],J,K).
```

Die Regeln `a1` und `a2` entsprechen den ersten beiden von Prolog gelieferten Antworten. Regel `a3` drückt die (unendlich vielen) Antworten der Anfrage `append(X,[b|Y],[a,b,c|Z])` aus, deren `X` von der Form `[a,b,c|L]` ist, wobei die Liste `L` nicht leer ist.

Im folgenden geben wir in Form eines Baumes an, wie diese Regeln durch das lineare Vervollständigungsverfahren generiert wurden:

$$
\begin{tikzcd}
 & \mathtt{r2@append(X,[b|Y],[a,b,c|Z])} \Leftrightarrow \mathtt{answer(X,Y,Z)} \arrow[d, "\mathtt{r2+r1}"] \\
 & \mathtt{r3@append(L1,[b|Y],[b,c|Z])} \Leftrightarrow \mathtt{answer([a|L1],Y,Z)} \arrow[dl, "\mathtt{r3+r0}"'] \arrow[d, "\mathtt{r3+r1}"] \\
\mathtt{a1@answer([a],[c|Z],Z)} \Leftrightarrow \mathtt{true} & \mathtt{r4@append(L2,[b|Y],[c|Z])} \Leftrightarrow \mathtt{answer([a,b|L2],Y,Z)} \arrow[d, "\mathtt{r4+r1}"] \\
 & \mathtt{r5@append(L3,[b|Y],Z)} \Leftrightarrow \mathtt{answer([a,b,c|L3],Y,Z)} \arrow[dl, "\mathtt{r5+r0}"'] \arrow[d, "\mathtt{r5+r1}"] \\
\mathtt{a2@answer([a,b,c],Y,[b|Y])} \Leftrightarrow \mathtt{true} & \mathtt{r6@append(L4,[b|Y],L2)} \Leftrightarrow \mathtt{answer([a,b,c,X'|L4],Y,[X'|L2])} \arrow[d, "\mathtt{r6+r5}"] \\
 & \mathtt{a3@answer([a,b,c,X'|L4],Y,[X'|L2])} \Leftrightarrow \mathtt{answer([a,b,c|L4],Y,L2)}
\end{tikzcd}
$$

Kapitel 7

Constraint Handling Rules modulo Permutativität

7.1 Konfluenz modulo Permutativität

Das im Kapitel 5 vorgestellte Vervollständigungsverfahren erlaubt es in vielen Fällen, ein terminierendes, nicht-konfluentes CHR-Programm in ein äquivalentes Programm zu transformieren, das terminierend und konfluent ist. Dieses Vorgehen ist nicht immer erfolgreich, wie das folgende Beispiel zeigt:

Beispiel 7.1 Gegeben sei folgendes CHR-Programm P, das die logische Negation (`neg/2`) implementiert. P stellt einen Ausschnitt des Booleschen Constraintlösers dar.

```
r1 @ neg(0,X)  ⇔  X=1.
r2 @ neg(X,0)  ⇔  X=1.
r3 @ neg(1,X)  ⇔  X=0.
r4 @ neg(X,1)  ⇔  X=0.
r5 @ neg(X,X)  ⇔  false.
r6 @ neg(X,Y) \ neg(Y,Z)  ⇔  X=Z.
r7 @ neg(X,Y) \ neg(Z,Y)  ⇔  X=Z.
r8 @ neg(Y,X) \ neg(Y,Z)  ⇔  X=Z.
```

P ist nicht konfluent, da folgende kritischen Paare nicht zusammenführbar sind:

$$\begin{aligned}K = \{&(\langle \mathtt{X} = \mathtt{X}, \mathtt{neg(X,Y)}, \mathtt{true}, \emptyset, \mathcal{V}\rangle, \langle \mathtt{Y} = \mathtt{Y}, \mathtt{neg(Y,X)}, \mathtt{true}, \emptyset, \mathcal{V}\rangle),\\ &(\langle \mathtt{X} = \mathtt{Z}, \mathtt{neg(X,Y)} \wedge \mathtt{neg(W,Z)}, \mathtt{true}, \emptyset, \mathcal{V}'\rangle, \langle \mathtt{Y} = \mathtt{W}, \mathtt{neg(X,Y)} \wedge \mathtt{neg(Y,Z)}, \mathtt{true}, \emptyset, \mathcal{V}'\rangle).\\ &(\langle \mathtt{X} = \mathtt{Z}, \mathtt{neg(X,Y)} \wedge \mathtt{neg(W,Z)}, \mathtt{true}, \emptyset, \mathcal{V}'\rangle, \langle \mathtt{W} = \mathtt{Y}, \mathtt{neg(X,Y)} \wedge \mathtt{neg(W,Z)}, \mathtt{true}, \emptyset, \mathcal{V}'\rangle)\}\end{aligned}$$

Dabei ist $\mathcal{V} = [\mathtt{X}, \mathtt{Y}]$ und $\mathcal{V}' = [\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{W}]$. Das erste kritische Paar stammt aus dem Unifizieren des Kopfes der Regel `r6` mit sich selbst. Die beiden anderen entstehen aus dem Unifizieren eines Constraintatoms des Kopfes der Regel `r6` mit einem des Kopfes der Regel `r7`.

Wie bei Termersetzungssystemen tritt hier der Fall ein, daß man ein nicht-zusammenführbares kritisches Paar hat, das prinzipiell nicht richtbar ist. Dieses Problem tritt in Termersetzungssystemen auf, wenn eine Funktion z.B. die Kommutativitätseigenschaft hat. Da man in CHR daran interessiert ist, Constraints zu definieren, verursachen symmetrische Constraints dieselben Probleme. Im folgenden wird gezeigt, daß der Versuch, eine Formulierung einer Regel zu finden, die die Symmetrie ausdrückt, in einem CHR-Programm resultiert, das nicht mehr terminiert.

Das Hinzufügen einer Simplifikationsregel, die die Symmetrie des Constraints `neg` ausdrückt,

```
r9 @ neg(X,Y)  ⇔  neg(Y,X).
```

führt zu einem nicht-terminierenden Constraintlöser.

CHR stellt gegenüber Termersetzungssystemen einen erweiterten Formalismus zur Verfügung: man hat noch die Möglichkeit, die Symmetrie mittels einer Propagationsregel auszudrücken:

```
neg(X,Y) ⇒ neg(Y,X).
```

Leider führt diese Regel auch zu einem nicht-terminierenden Löser. Die neu eingefügte Propagationsregel löst eine Oszillation zwischen den zwei Zuständen `neg(X,Y)` und `neg(X,Y),neg(Y,X)` aus, da ihre Wirkung zu der Wirkung der folgenden bereits im Booleschen Constraintlöser enthaltenen Regel konträr ist:

```
r6 @ neg(X,Y) \ neg(Y,Z) ⇔ X = Z.
```

Zum Beispiel kann ausgehend von $\langle \top, \mathtt{neg(X,Y)} \wedge \mathtt{neg(Y,X)}, \mathtt{true}, \top, \{\mathtt{r9@neg(X,Y)}\}, [\mathtt{X},\mathtt{Y}]\rangle$ mittels der in Abschnitt 4.3.1 eingeführten operationalen Semantik eine unendliche Ableitung erzeugt werden:

$$
\begin{array}{ll}
 & \langle \top, \mathtt{neg(X,Y)} \wedge \mathtt{neg(Y,X)}, \mathtt{true}, \{\mathtt{r9@neg(X,Y)}\}, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Simpagate}} & \langle \top, \mathtt{neg(X,Y)}, \mathtt{true}, \{\mathtt{r9@neg(X,Y)}\}, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Propagate}} & \langle \mathtt{neg(Y,X)}, \mathtt{neg(X,Y)}, \mathtt{true}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Introduce}} & \langle \top, \mathtt{neg(X,Y)} \wedge \mathtt{neg(Y,X)}, \mathtt{true}, \{\mathtt{r9@neg(Y,X)}\}, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Simpagate}} & \langle \top, \mathtt{neg(Y,X)}, \mathtt{true}, \{\mathtt{r9@neg(Y,X)}\}, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Propagate}} & \langle \mathtt{neg(X,Y)}, \mathtt{neg(Y,X)}, \mathtt{true}, \emptyset, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Introduce}} & \langle \top, \mathtt{neg(X,Y)} \wedge \mathtt{neg(Y,X)}, \mathtt{true}, \{\mathtt{r9@neg(X,Y)}\}, [\mathtt{X},\mathtt{Y}]\rangle \\
\mapsto_{\mathbf{Simpagate}} & \cdots
\end{array}
$$

In [Meu96] wurde eine Möglichkeit vorgeschlagen, wie man einen konfluenten und terminierenden Constraintlöser für `neg/2` bekommen kann. Dies erforderte aber eine (manuelle) Veränderung des CHR-Programms, die zu einem unübersichtlichen und ineffizienten konfluenten Constraintlöser führt.

Die geschilderten Probleme tauchen ständig auf; nicht nur, wenn ein zweistelliges Constraintsymbol symmetrisch ist, sondern auch dann, wenn für ein Constraintsymbol mit größerer Stelligkeit Argumente vertauscht werden dürfen. Wir wollen deswegen nicht von Symmetrie, sondern allgemeiner von *Permutativität* sprechen.

Ein Ansatz zur Lösung der beschriebenen Probleme besteht darin, die Berücksichtigung der Permutativität von Constraintsymbolen direkt in das Auswertungsverfahren einzubauen, sodaß die Terminierungseigenschaften des Programms unverändert bleiben. Die Idee für diesen Ansatz stammt aus dem Gebiet der Termersetzungssysteme, wo z.B. kommutative Funktionssymbole ähnliche Schwierigkeiten bereiten. Dort werden ausgezeichnete Gleichungen nicht in Regeln umgeformt, sondern als Erzeugende einer Äquivalenzrelation auf den Termen betrachtet, die dann in den Ableitungsprozess eingeht [Hue80, ENP92, Pla93].

Wie muß die operationale Semantik zur Berücksichtigung der Permutativität geändert werden? Nehmen wir an, das betrachtete Programm enthält eine Menge von Regeln der Form $c(x_1,\ldots,x_n) \approx c(x_{\pi(1)},\ldots,x_{\pi(n)})$, wobei c ein Constraintsymbol ist, die (x_i) Variablen sind und π eine Permutation von $1,\ldots,n$ ist. Intuitiv bedeutet eine solche Regel, daß die Argumente für das Constraint in der angegebenen Weise vertauscht werden können, ohne daß sich dabei die Bedeutung verändert. Wir erhalten damit eine Äquivalenzrelation auf Constraintatomen:

Definition 7.2 Wir definieren zuerst eine Relation R auf Constraintatomen:

$$
\begin{gathered}
c(y_1,\ldots,y_n)\ R\ c(z_1,\ldots,z_n) \\
\leftrightarrow \\
\exists (c(x_1,\ldots,x_n) \approx c(x_{\pi(1)},\ldots,x_{\pi(n)})) \in P \wedge \forall i.y_i = z_{\pi(i)},
\end{gathered}
$$

wobei P das Programm ist. Wir definieren $\approx$ nun als die reflexive, symmetrische und transitive Hülle von R. Wir bezeichnen $\approx$ als *Gleichheit modulo Permutativität.*

Zu einer geeigneten operationalen Semantik gelangen wir, indem wir die syntaktische Gleichheit $\doteq$ durch $\dot{\approx}$, die *syntaktische Gleichheit modulo Permutativität*, ersetzen. Wir erhalten $\dot{\approx}$ durch Erweiterung der CET um die Regel

$$c(y_1, \ldots, y_n) \dot{\approx} c(z_1, \ldots, z_n)$$
$$\leftrightarrow$$
$$\exists (c(x_1, \ldots, x_n) \approx c(x_{\pi(1)}, \ldots, x_{\pi(n)})) \in P \wedge \forall i. y_i \doteq z_{\pi(i)},$$

und um Regeln, die Reflexivität, Symmetrie und Transitivität für $\dot{\approx}$ fordern. Damit ergibt sich auch eine Erweiterung der Constrainttheorie, die wir mit $CT_{\approx}$ bezeichnen.

Mit $\dot{\approx}$ erhalten wir auch neue Begriffe: *Matching modulo Permutativität*, kurz auch *Matching modulo* $\approx$ oder $\approx$*-Matching* und $\approx$*-Ableitung*. Mit $\approx$-Matching gibt es für Regeln u.U. mehr Möglichkeiten zu feuern:

Beispiel 7.3 Gegeben sei die erste Regel des CHR-Programms aus Beispiel 7.1:

```
r1 @ neg(0,X) ⇔ X=1.
```

Der Zustand $\langle \top, \texttt{neg(X,0)}, \texttt{true}, \emptyset, [\texttt{X}] \rangle$ kann mit der operationalen Semantik aus Abschnitt 4.3.1 nicht reduziert werden, da `neg(X,0)` nicht mit dem Kopf der Regel `r1` matcht. Aber wenn wir das Constraintsymbol `neg` als symmetrisch definieren und $\approx$-Matching verwenden, dann kann auch Regel `r1` angewendet werden und die Ableitung resultiert in dem Endzustand $\langle \top, \top, \texttt{X} = \texttt{1}, \emptyset, [\texttt{X}] \rangle$.

Der Vorteil von $\approx$-Matching liegt darin, daß man sich die Regeln für die „symmetrischen" Fälle sparen kann. Dadurch werden Constraintlöser kürzer, die Gefährdung der Konfluenz durch Vergessen von Regeln entfällt bzw. die Konfluenz wird überhaupt erst ermöglicht. Allerdings ist für $\approx$-Matching die resultierende Substitution nicht mehr eindeutig bestimmt; dieselbe Regel kann auf dasselbe Constraint u.U. auf verschiedene Weisen angewendet werden, wenn für mindestens ein im Constraint vorkommenden Constraintsymbol Permutationen definiert sind. In diesem Fall muß sich der vordefinierte Constraintlöser für eine Substitution entscheiden.

Beispiel 7.4 Gegeben sei das CHR-Programm

```
r @ p(X,Y) ⇒ X = 1 ∧ Y = 0.
```

Ist das Constraintsymbol `p` als symmetrisch definiert, dann kann die Berechnung für den Zustand $\langle \top, \texttt{p(1,0)}, \texttt{true}, \{\texttt{r@p(1,0)}\}, [] \rangle$ scheitern oder mit dem Zustand $\langle \top, \texttt{p(1,0)}, \texttt{true}, \emptyset, [] \rangle$ terminieren.

Das Beispiel zeigt auch, daß man kritische Paare von Propagationsregeln mit Propagationsregeln betrachten muß, wenn man den Zusammenhang zwischen lokaler Konfluenz und Zusammenführbarkeit der kritischen Paare auch für die neue operationale Semantik etablieren will. Außerdem muß man bei der Bildung der kritischen Paare der Tatsache Rechnung tragen, daß $\approx$-Matching nicht eindeutig ist. Wir führen dazu die $\approx$-Unifikation ein:

Definition 7.5 Zwei Constraintatome C_1 und C_2 heißen $\approx$-unifizierbar, wenn es eine Substitution σ gibt mit $C_1\sigma \approx C_2\sigma$. Dann heißt σ ein $\approx$-Unifikator von C_1 und C_2. Die Menge aller $\approx$-Unifikatoren von C_1 und C_2 wird mit $U_{\approx}(C_1, C_2)$ bezeichnet.

Die Lösungsmenge eines Theorie-Unifikationsproblems ist im allgemeinen unendlich und man möchte sie durch eine möglichst kleine, repräsentative Teilmenge ersetzen. Üblicherweise betrachtet man deswegen Mengen *allgemeinster* Unifikatoren. Die folgende Definition überträgt diesen Begriff auf die $\approx$-Unifikation:

Definition 7.6 Die Menge $\mu U_{\approx}(C_1, C_2)$ der *allgemeinsten* $\approx$-Unifikatoren für Constraints C_1 und C_2 ist durch die folgenden Eigenschaften definiert:

- Korrektheit: $\mu U_{\approx}(C_1, C_2) \subseteq U_{\approx}(C_1, C_2)$, d.h. jeder allgemeinste Unifikator ist auch eine Lösung für das Unifikationsproblem.
- Vollständigkeit: Für jedes $\tau \in U_{\approx}(C_1, C_2)$ gibt es ein $\sigma \in \mu U_{\approx}(C_1, C_2)$ und eine Substitution λ mit $C_1\tau \approx C_1\sigma\lambda$ und $C_2\tau \approx C_2\sigma\lambda$, d.h. jeder beliebige *equiv*-Unifikator ist Instanz eines allgemeinsten $\approx$-Unifikators.
- Minimalität: Für alle $\sigma, \tau \in \mu U_{\approx}(C_1, C_2)$ gilt: wenn es eine Substitution λ gibt, sodaß $C_1\tau \approx C_1\sigma\lambda$ und $C_2\tau \approx C_2\sigma\lambda$ ist, dann ist $\sigma = \tau$, d.h. verschiedene allgemeinste $\approx$-Unifikatoren sind keine Instanzen voneinander.

Die Menge $\mu U_{\approx}(c(x_1, \ldots, x_n), c(y_1, \ldots, y_n)$ ist maximal einelementig, falls für c keine Permutation der Argumente definiert ist; in diesem Fall entspricht $\approx$-Unifikation der normalen Unifikation. Mit jeder Permutation ergibt sich aber eine neue Möglichkeit zur Unifikation, die selbst wieder maximal einen allgemeinsten Unifikator liefern kann. Für Konjunktionen von Constraintatomen $C_1 \wedge \ldots \wedge C_n$ und $C'_1 \wedge \ldots \wedge C'_n$ ist die Kardinalität von $\mu U_{\approx}(C_1 \wedge \ldots \wedge C_n, C'_1 \wedge \ldots \wedge C'_n)$ kleiner oder gleich dem Produkt der Kardinalitäten der Mengen $\mu U_{\approx}(C_i, C'_i)$ für $1 \leq i \leq n$.

Im folgenden werden wir die $\approx$-Unifikatoren als Gleichheitsconstraints wie im CLP-Kalkül betrachten. Wir gehen davon aus, daß ein Algorithmus zur Verfügung steht, der folgendes leistet: Gegeben seien zwei Constraints C_1 und C_2. Wenn $C_1 \dot{\approx} C_2$ in $CT_{\approx}$ erfüllbar ist (d.h. C_1, C_2 sind $\approx$-unifizierbar), dann werden für $C_1 \dot{\approx} C_2$ alle (Normal-)Formen $X_1 \doteq t_1 \wedge \ldots \wedge X_n \doteq t_n$ berechnet, wobei die Variablen $X_1, \ldots, X_n$ paarweise verschieden sind und nicht in den Termen $t_1, \ldots, t_n$ vorkommen, ansonsten soll $C_1 \dot{\approx} C_2$ zu *false* reduziert werden.

Nun können wir den Begriff des kritischen Paares anpassen:

Definition 7.7 ($\approx$-Kritisches Paar) Sei R_1 eine CHR-Regel mit Wächter G, Rumpf B und Kopf H und sei R_2 eine CHR-Regel mit Wächter G', Rumpf B' und Kopf H'. Seien $\{H_i \mid 1 \leq i \leq n\}$ und $\{H'_i \mid 1 \leq i \leq m\}$ die Mengen der Atome in H bzw. H'.
Für jede Normalform von $H_{i_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_1}, \ldots, H'_{j_k}$, wobei $H_{i_1}, \ldots, H_{i_k}$ ein oder mehrere Constraintatome des Kopfs von R_1 sind und $H'_{j_1}, \ldots, H'_{j_k}$ ein oder mehrere Constraintatome des Kopfs von R_2 sind, heißt das Paar

$$(\langle B, \mathrm{survive}(R_1) \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}, \bar{G}, \mathcal{T}_1, \mathcal{V}\rangle, \langle B', \mathrm{survive}(R_2) \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n}, \bar{G}, \mathcal{T}_2, \mathcal{V}\rangle)$$

ein $\approx$-*kritisches Paar* von R_1 und R_2, falls

- R_1 und R_2 Simplifikationsregeln sind oder
- R_1 und R_2 Propagationsregeln sind, die Varianten voneinander sind, oder
- R_1 und R_2 Simpagationsregeln oder
- R_1 eine Simplifikationsregel und R_2 eine Propagationsregel oder
- R_1 eine Simplifikationsregel und R_2 eine Simpagationsregel oder
- R_1 eine Simpagationsregel und R_2 eine Propagationsregel

$\bar{G} = G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}$, $\{i_1, \ldots, i_n\}$ und $\{j_1, \ldots, j_m\}$ sind dabei Permutationen von $\{1, \ldots, n\}$ bzw. $\{1, \ldots, m\}$ und $1 \leq k \leq \min(n, m)$. $\mathcal{V}$ ist die Liste der Variablen in $H_1, \ldots, H_n$, $H'_1, \ldots, H'_m$. Der Tokenspeicher $\mathcal{T}_2$ ist immer leer. Der Tokenspeicher $\mathcal{T}_1$ ist leer für die ersten fünf Fälle. Wenn R_1 eine Simpagationsregel und R_2 eine Propagationsregel sind, dann ist $\mathcal{T}_1$ leer, wenn $\{R_2@H'\}$ bezüglich $\mathrm{survive}(R_1) \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}$ überflüssig ist, ansonsten ist $\mathcal{T}_1 = \{R_2@H'\}$.

Für den Satz der lokalen $\approx$-Konfluenz brauchen wir noch die Begriffe $\approx$-*Zusammenführbarkeit* und *(lokale)* $\approx$-*Konfluenz* (Abbildung 7.1). Diese ergeben sich mit der $\approx$-*Varianz.* $\mapsto$ bezeichnet im folgenden einen Ableitungsschritt mit $\approx$-Matching.

Definition 7.8 Ein Constraintatom C heißt $\approx$-Instanz von D, falls es eine Substitution σ gibt, sodaß $C \approx D\sigma$. C heißt $\approx$-Variante von D, falls C $\approx$-Instanz von D ist und falls D $\approx$-Instanz von C ist.

Definition 7.9 Sei P ein CHR-Programm. S_1 und S_2 sind $\approx$-zusammenführbar (in Zeichen $S_1 \downarrow_\approx S_2$), falls es T_1, T_2 gibt, so daß, $S_1 \mapsto^* T_1$ und $S_2 \mapsto^* T_2$ und T_1, T_2 $\approx$-Varianten sind.
P ist *lokal* $\approx$-konfluent, falls gilt: Aus $S \mapsto S_1$ und $S \mapsto S_2$, folgt $S_1 \downarrow_\approx S_2$.
P ist $\approx$-konfluent, falls gilt: Aus $S \mapsto^* S_1$ und $S \mapsto^* S_2$, folgt $S_1 \downarrow_\approx S_2$.

$$
\text{(A)}\quad
\begin{tikzcd}
 & S \arrow[dl, "*"'] \arrow[dr, "*"] & \\
S_1 \arrow[d, "*"] & & S_2 \arrow[d, "*"'] \\
S_1' & \approx & S_2'
\end{tikzcd}
\qquad\qquad
\text{(B)}\quad
\begin{tikzcd}
 & S \arrow[dl] \arrow[dr] & \\
S_1 \arrow[d, "*"] & & S_2 \arrow[d, "*"'] \\
S_1' & \approx & S_2'
\end{tikzcd}
$$

Abbildung 7.1: $\approx$-Konfluenz (A) und Lokale $\approx$-Konfluenz (B)

Satz 7.10 Sei P ein CHR-Programm. P ist genau dann lokal $\approx$-konfluent, wenn alle seine $\approx$-kritischen Paare $\approx$-zusammenführbar sind.
Beweisidee: "$\Longrightarrow$": Der Fall **Solve vs. Solve** ist unproblematisch. Im Fall **Solve vs. Simplify** behält das Argument für die Anwendbarkeit von **Simplify** nach **Solve** seine Gültigkeit. Allerdings besteht die Möglichkeit, daß in den Berechnungszweigen unterschiedliche Substitutionen für die Anwendung der Simplifikationsregel gewählt werden. Insbesondere ist es aber auch möglich, daß in beiden Zweigen dieselbe Substitution gewählt wird, wodurch die Zusammenführbarkeit erhalten bleibt. Mit der gleichen Argumentation können auch die übrigen Fälle bewiesen werden. Nur für **Propagate vs. Propagate** muß ein zweiter Fall betrachtet werden: Wenn die Regeln Varianten voneinander sind, dann muß man analog zu **Simplify vs. Simplify** unter Bezugnahme auf die Zusammenführbarkeit aller $\approx$-kritischen Paare zeigen, daß die Nachfolgezustände des kritischen Zustandes zusammenführbar sind.

"$\Longleftarrow$": Die Argumentation bleibt in allen Fällen gültig. Allerdings muß berücksichtigt werden, daß es auch kritische Paare von Propagationsregeln mit sich selbst gibt. In diesem Fall kann aber wie im Fall für zwei Simplifikationsregel argumentiert werden. □

7.2 Vervollständigung modulo Permutativität

Beispiel 7.11 Gegeben sei ein Ausschnitt des Booleschen Constraintlösers, der die Konjunktion `and/3` und die Implikation `imp/2` implementiert:

```
and1 @ and(0,X,Y) ⇔ Y=0.
and2 @ and(X,0,Y) ⇔ Y=0.
and3 @ and(1,X,Y) ⇔ Y=X.
and4 @ and(X,1,Y) ⇔ Y=X.
and5 @ and(X,Y,1) ⇔ X=1,Y=1.
and6 @ and(X,X,Z) ⇔ X=Z.
and7 @ and(X,Y,X) ⇔ imp(X,Y).
and8 @ and(X,Y,Y) ⇔ imp(Y,X).
```

```
and9 @ and(X,Y,A) \ and(X,Y,B)  ⇔  A=B.
and10 @ and(X,Y,A) \ and(Y,X,B)  ⇔  A=B.

imp1 @ imp(0,X)  ⇔  true.
imp2 @ imp(X,0)  ⇔  X=0.
imp3 @ imp(1,X)  ⇔  X=1.
imp4 @ imp(X,1)  ⇔  true.
imp5 @ imp(X,X)  ⇔  true.
imp6 @ imp(X,Y),imp(Y,X)  ⇔  X=Y.
```

Dieses CHR-Programm ist nicht konfluent, da einige kritische Paare nicht zusammenführbar sind:

1. $(\langle \top, \mathtt{and(B,A,C)} \wedge \mathtt{imp(A,B)}, \mathtt{true}, \emptyset, [\mathtt{A,B,C}]\rangle, \langle \mathtt{A} = \mathtt{C}, \mathtt{and(A,B,A)}, \mathtt{true}, \emptyset, [\mathtt{A,B,C}]\rangle)$

   Dieses kritische Paar stammt aus dem Unifizieren des Kopfes der Regel `and7` mit dem ersten Constraintatom des Kopfes der Regel `and10`.

2. $(\langle \top, \mathtt{and(B,A,C)} \wedge \mathtt{imp(B,A)}, \mathtt{true}, \emptyset, [\mathtt{A,B,C}]\rangle, \langle \mathtt{B} = \mathtt{C}, \mathtt{and(A,B,B)}, \mathtt{true}, \emptyset, [\mathtt{A,B,C}]\rangle)$

   Dieses kritische Paar stammt aus dem Unifizieren des Kopfes der Regel `and8` mit dem ersten Constraintatom des Kopfes der Regel `and10`.

3. $(\langle \mathtt{C} = \mathtt{D}, \mathtt{and(A,B,C)}, \mathtt{true}, \emptyset, [\mathtt{A,B,C,D}]\rangle, \langle \mathtt{D} = \mathtt{C}, \mathtt{and(B,A,D)}, \mathtt{true}, \emptyset, [\mathtt{A,B,C,D}]\rangle)$

   Dieses kritische Paar stammt aus dem Unifizieren des Kopfes der Regel `and10` mit sich selbst.

Wenn man das Vervollständigungsverfahren aus Abschnitt 5 auf `and/3` und `imp/2` anwendet, dann werden für die ersten beiden kritischen Paare folgende Regeln vorgeschlagen:

```
r0 @ imp(A,B) \ and(B,A,C)  ⇔  true | C = A.
r1 @ imp(A,B) \ and(A,B,C)  ⇔  true | C = A.
```

Dabei ist die erste Regel der symmetrische Fall der zweiten. Für das letzte kritische Paar kann man keine CHR-Regel angeben, die die Terminierung des Lösers einhält. Man kann dann durch die Vervollständigung dieses Lösers nicht zu einem äquivalenten konfluenten und terminierenden Löser transformieren.

Man verallgemeinert das Vervollständigungsverfahren zur Vervollständigung modulo Permutativität, indem man in den Inferenzregeln aus Abschnitt 5 $\approx$-kritische Paare zugrundelegt und die $\approx$-Zusammenführbarkeit für die Zusammenführbarkeit einsetzt.

Wenn man das Vervollständigungsverfahren modulo Permutativität auf obiges Beispiel anwendet, dann erhält man mehr nicht-zusammenführbare kritische Paare. Aber durch das Hinzufügen der ersten Regel `r0` werden alle kritischen Paare zusammenführbar und man bekommt daher einen Constraintlöser, der modulo Permutativität konfluent ist.

7.3 Implementierung

Wie oben erwähnt, hat man oftmals das Problem, daß kritische Paare nicht zusammenführbar sind, da zum einen semantisch äquivalente, aber syntaktisch verschiedene Constraints nicht identifiziert werden können (z.B. `and(A,B,C)` und `and(B,A,C)` oder `neg(A,B)` und `neg(B,A)` oder `maximum(A,B,C)` und `maximum(B,A,C)`), es zum anderen aber auch nicht möglich ist, Simplifikationsregeln der Form `neg(A,B)` ⇔ `neg(B,A)` hinzuzufügen, weil das die Terminierung kosten würde.

Um dieses Problem zu lösen, wurde für den Anwender die Möglichkeit geschaffen, eine Äquivalenz auf Constraints zu definieren. Damit werden dynamisch alle Regeln gebildet, die sich durch Anwendung der Äquivalenz auf die Köpfe der Regeln des betrachteten Lösers ergeben.

Diese Regeln werden bei der Berechnung des Tokenspeichers, beim Matching und bei der Bildung kritischer Paare berücksichtigt. Das Verfahren arbeitet im Grunde wie ein Präprozessor.

Da die Form, in der ein Constraint vorliegt, nun für die Anwendung von Regeln auf das Constraint nicht mehr bedeutend ist, ist es nicht notwendig und auch nicht sinnvoll, die Äquivalenz auf die rechten Seiten der Regeln des betrachteten Lösers anzuwenden. Allerdings kann es nun passieren, daß keine syntaktisch übereinstimmenden Endzustände mehr erzeugt werden können, obwohl semantisch äquivalente Endzustände vorhanden sind. Also muß die Äquivalenz auch beim abschließenden Varianztest in `joinable` angewendet werden. Dies entspricht also der oben vorgestellten Konfluenz modulo Permutativität.

Für den Programmierer ergeben sich noch weitere Vorteile: Durch die Äquivalenz abgedeckte Regeln können weggelassen werden. So können z.B. in dem Booleschen Constraintlöser 23 von 66 Regeln gestrichen werden. Die Anzahl der vom Konfluenztester betrachteten Regeln aber reduziert sich nicht. Im folgenden geben wir den vereinfachten Booleschen Constraintlöser an:

```
and1 @ and(0,X,Y) ⇔ Y=0.
and3 @ and(1,X,Y) ⇔ Y=X.
and5 @ and(X,Y,1) ⇔ X=1 ∧ Y=1.
and6 @ and(X,X,Z) ⇔ X=Z.
and7 @ and(X,Y,X) ⇔ imp(X,Y).
and8 @ and(X,Y,Y) ⇔ imp(Y,X).
and9 @ and(X,Y,A) \ and(X,Y,B) ⇔ A=B.

or1 @ or(0,X,Y) ⇔ Y=X.
or3 @ or(X,Y,0) ⇔ X=0,Y=0.
or4 @ or(1,X,Y) ⇔ Y=1.
or6 @ or(X,X,Z) ⇔ X=Z.
or7 @ or(X,Y,X) ⇔ imp(Y,X).
or8 @ or(X,Y,Y) ⇔ imp(X,Y).
or10 @ or(X,Y,A) \ or(X,Y,B) ⇔ A=B.

exor1 @ exor(0,X,Y) ⇔ X=Y.
exor3 @ exor(X,Y,0) ⇔ X=Y.
exor4 @ exor(1,X,Y) ⇔ neg(X,Y).
exor6 @ exor(X,Y,1) ⇔ neg(X,Y).
exor7 @ exor(X,X,Y) ⇔ Y=0.
exor8 @ exor(X,Y,X) ⇔ Y=0.
exor10 @ exor(X,Y,A) \ exor(X,Y,B) ⇔ A=B.

neg1 @ neg(0,X) ⇔ X=1.
neg3 @ neg(1,X) ⇔ X=0.
neg5 @ neg(X,X) ⇔ fail.
neg_neg1 @ neg(X,Y) \ neg(Y,Z) ⇔ X=Z.
neg_neg2 @ neg(X,Y) \ neg(Z,Y) ⇔ X=Z.
neg_neg3 @ neg(Y,X) \ neg(Y,Z) ⇔ X=Z.

neg_and1 @ neg(X,Y) \ and(X,Y,Z) ⇔ Z=0.
neg_and3 @ neg(X,Z) ∧ and(X,Y,Z) ⇔ X=1 ∧ Y=0 ∧ Z=0.
neg_and5 @ neg(Y,Z) ∧ and(X,Y,Z) ⇔ X=0 ∧ Y=1 ∧ Z=0.

neg_or1 @ neg(X,Y) \ or(X,Y,Z) ⇔ Z=1.
neg_or3 @ neg(X,Z) ∧ or(X,Y,Z) ⇔ X=0 ∧ Y=1 ∧ Z=1.
neg_or5 @ neg(Y,Z) ∧ or(X,Y,Z) ⇔ X=1 ∧ Y=0 ∧ Z=1.

neg_exor1 @ neg(X,Y) \ exor(X,Y,Z) ⇔ Z=1.
```

```
neg_exor3 @ neg(X,Z) \ exor(X,Y,Z) ⇔ Y=1.
neg_exor5 @ neg(Y,Z) \ exor(X,Y,Z) ⇔ X=1.

neg_imp1 @ neg(X,Y) ∧ imp(X,Y) ⇔ X=0 ∧ Y=1.

imp1 @ imp(0,X) ⇔ true.
imp2 @ imp(X,0) ⇔ X=0.
imp3 @ imp(1,X) ⇔ X=1.
imp4 @ imp(X,1) ⇔ true.
imp5 @ imp(X,X) ⇔ true.
imp6 @ imp(X,Y) ∧ imp(Y,X) ⇔ X=Y.
```

Mit dem Prädikat $\approx$ kann der Anwender nun eine Äquivalenz auf seinen Constraints erzeugen. Für den Booleschen Löser kann man dann schreiben:

```
and_permutativity @ and(A,B,C) ≈ and(B,A,C).
or_permutativity @ or(A,B,C) ≈ or(B,A,C).
exor_permutativity @ exor(A,B,C) ≈ exor(B,A,C).
neg_symmetry @ neg(A,B) ≈ neg(B,A).
```

Der Konfluenztester erweitert die so entstandene Relation zu einer Äquivalenzrelation auf Constraints. Interessant sind die Auswirkungen auf die Vervollständigung auf den Booleschen Constraintlöser.

Ohne Äquivalenz und mit dem ursprünglichen Booleschen Constraintlöser (66 Regeln) können einige kritische Paare, die durch eine fehlende Permutativitätsregel hervorgerufen werden, nicht gerichtet werden (Siehe Beispiel 7.1). Also scheitert das Prädikat `complete` aus Abschnitt 5.3. Einige nicht-zusammenführbare kritische Paare können aber trotzdem zusammengeführt werden, in dem man die folgenden Regeln in den Constraintlöser hinzufügt:

```
r0 @ imp(A,B)  ∧ and(B,A,C) ⇔ imp(A,B) ∧ A=C.
r1 @ imp(A,B) ∧ and(A,B,C) ⇔ imp(A,B) ∧ A=C.
r2 @ imp(A,B) ∧ or(B,A,C) ⇔ imp(A,B) ∧ B=C.
r3 @ imp(A,B) ∧ or(A,B,C) ⇔ imp(A,B) ∧ B=C.
r4 @ imp(A,B) ∧ imp(A,B) ⇔ imp(A,B).
```

Dabei sind `r1` und `r3` nur die symmetrischen Fälle von `r0` bzw. `r2`.

Mit der oben angegebenen Äquivalenz und mit dem vereinfachten Constraintlöser sind nur noch drei Regeln zur Vervollständigung vom Booleschen Constraintlöser notwendig. Die symmetrischen Fälle werden nicht vorgeschlagen.

```
r1 @ imp(A,B) ∧ and(A,B,C) ⇔ imp(A,B) ∧ A=C.
r3 @ imp(A,B) ∧ or(A,B,C) ⇔ imp(A,B) ∧ B=C.
r4 @ imp(A,B) ∧ imp(A,B) ⇔ imp(A,B).
```

Wenn man die Regeln `r1`,`r3` und `r4` in den vereinfachten Constraintlöser hinzufügt, erhält man einen Booleschen Constraintlöser, der modulo Permutativität konfluent ist.

Kapitel 8

Vergleich mit verwandten Bereichen

Die Konfluenz ist eine wichtige Eigenschaft, die in verschiedenen Bereichen untersucht wurde. In manchen Gebieten wie (idealisiertes) Prolog und λ-Kalkül ist diese Eigenschaft automatisch gewährleistet: Bei Prolog wird die Konfluenz durch das „Switching Lemma" garantiert ([Llo87]), das aussagt, daß die Auswahl der Literale die Menge der Antworten nicht beeinflußt. Die Konfluenz des λ-Kalküls wird im Church-Rosser-Satz verkörpert, der besagt, daß verschiedene Berechnungssequenzen ausgehend von demselben Term zu einem gemeinsamen Term zusammengeführt werden kann. In anderen Gebieten ist die Konfluenz eine wünschenswerte Eigenschaft. Für nebenläufige logische Sprachen (bzw. regelbasierte Systeme wie Termersetzungssysteme) sind die konfluenten Programme (bzw. Systeme) eine echte Teilklasse der syntaktisch korrekten Programme (bzw. Systeme).

In keinem dieser Bereiche existiert ein Konstrukt, das der Propagationsregel entspricht. Das ist der Hauptgrund, der die Übertragung der Konfluenzergebnisse aus diesen Bereichen auf CHR erschwert. In diesem Kapitel werden wir einen Vergleich zu den Bereichen ziehen, die mit Constraint Handling Rules verwandt sind. Wie man in früheren Kapiteln gesehen hat, erinnert der CHR-Ansatz und seine Konfluenzergebnisse sehr stark an Überlegungen für Termersetzungssysteme. Deswegen werden im nächsten Abschnitt die Zusammenhänge zwischen beiden Ansätzen erklärt. Anschließend wird ein Vergleich mit anderen nebenläufigen Constraintsprachen durchgeführt, da CHR auch eine nebenläufige Sprache ist.

8.1 CHR und Termersetzungssysteme

Ein Termersetzungssystem ist ein gerichtetes Gleichungssystem, es definiert also in natürlicher Weise eine Reduktionsrelation auf der Menge der Terme. Eine Erweiterung davon sind die bedingten Termersetzungssysteme (Conditional Term Rewriting Systems, CTRS) [Kap84, DOS88, Der93]), bei denen die Anwendung einer Regel einer Bedingung (ähnlich wie die Wächter bei CHR) unterworfen ist.

Wegen der strukturellen Ähnlichkeit von SiR-Programmen und den bedingten Termersetzungssystemen war der erste Ansatz der vorliegenden Arbeit, SiR-Programme in bedingte Termersetzungssysteme einzubetten ([AFMM95]). Die Idee war, zu zeigen, daß jedes SiR-Programm auch als bedingtes Termersetzungssystem aufgefaßt werden kann. Dann könnte man alle Ergebnisse über Konfluenz von bedingten Termersetzungssystemen auf SiR-Programme übertragen. Die naheliegende Idee dieser Einbettung war, jeden Zustand als Term aufzufassen. Auf diese Terme werden dann Termersetzungsregeln angewendet, die direkt aus den SiR-Regeln, die das Programm konstituieren, hervorgehen. Für die vordefinierten Constraints kann die Existenz von Termersetzungsregeln angenommen werden, die das Verhalten der Normalisierungsfunktion repräsentieren.

Es zeigte sich jedoch bei dieser Untersuchung, daß es einige fundamentale Unterschiede zwischen den beiden Formalismen gibt, die es unmöglich machen, CHR-Programme einfach als bedingte Termersetzungssysteme aufzufassen:

Globales Wissen: Die Ersetzung von Subtermen bei bedingten Termersetzungssystemen geschieht rein lokal: Ob eine Termersetzungsregel auf einen Subterm angewendet wird, ist völlig unabhängig von der Identität des umgebenden Termes. Bei CHR-Programmen ist das nicht der Fall: Ob eine CHR-Regel anwendbar ist, ist abhängig von den Constraints im vordefinierten Constraintspeicher. Sieht man den ganzen Zustand als Term, verstößt dies gegen die Lokalität von bedingten Termersetzungssystemen. Es gibt eine Möglichkeit, zusätzliche Termersetzungsregeln einzubauen, die die vordefinierten Constraints durch den ganzen Zustand propagieren, so daß jedem Constraint im benutzerdefinierten Speicher der Inhalt des aktuellen vordefinierten Constraintspeichers zugeordnet wird. Dann müssen aber auch Regeln eingebaut werden, die neu hinzugefügte vordefinierte Constraints an einer Stelle sammeln und kombinieren. Sobald aber diese zwei Sorten von Regeln gefordert werden, terminiert das bedingte Termersetzungssystem nicht mehr, und es gibt keine vernünftige Möglichkeit, die Terminierung durch eine Steuerung herzustellen, wenn man nur lokalen Zugriff auf Subterme hat.

Lokale Variablen im Rumpf: Die Konfluenzergebnisse von (bedingten) Termersetzungssystemen fordern, daß eine (bedingte) Termersetzungsregel von der Form $c \mid l \rightarrow r$ die Variablenbedingung erfüllt, daß alle Variablen von r auch in l vorkommen. Diese Bedingung bewirkt, daß von einer Gleichung jeweils höchstens eine Instanz existieren kann, deren linke Seite mit einem gegebenen Unterterm eines aktuellen Terms übereinstimmt. Diese Instanz läßt sich durch „matching" zwischen der linken Seite der Termersetzungsregel und dem Unterterm berechnen. Wenn man diese Bedingung verletzt, dann reicht matching nicht aus. Man sollte dann mit Hilfe der Unifikation („narrowing") alle möglichen Substitutionen, die c erfüllen, aufzählen. Es ist unglücklicherweise im allgemeinen nicht entscheidbar, ob eine solche Substitution existiert. Die Konfluenzergebnisse für CHR-Programme brauchen diese Restriktion nicht.

Wenn man zusätzlich Propagationsregeln betrachtet, dann hat man das Problem, daß man bei einer naiven Übersetzung der Propagationsregeln in Simplifikationsregeln nicht-terminierende Programme erhält: Es gibt keine Möglichkeit, die Anwendungsbedingung der Propagationsregel so zu formalisieren, daß die Lokalität erhalten bleibt.

8.2 CHR und Nebenläufige Constraintsprachen

CHR ist eine committed choice Sprache, die nebenläufige Constraint-Logikprogrammiersprachen (kurz NCLP-Sprachen) um Propagationsregeln und mehrköpfige Regeln erweitert. Man könnte sich fragen, wieso wir uns mit der Konfluenz ärgern anstatt den Determinisumusbegriff von Maher [Mah87] und Saraswat [Sar93] auf SiR-Programme anzupassen. Auf den ersten Blick sieht es so aus, daß jedes konfluente SiR-Programm in ein äquivalentes deterministisches Programm übersetzt werden kann. Dies ist aber nicht der Fall, da meistens das resultierende determistische Programm „operational schwächer" als sein konfluentes Gegenstück ist. Die Klasse der konfluenten Programme ist also echt größer als die Klasse der deterministischen Programme:

- Das Constraintsystem muß unter der Negation abgeschlossen sein, damit ein NCLP-Programm in ein Programm transformiert wird, das keine überlappenden Wächter hat.

  Beispiel 8.1 Sei P das Programm, das das Constraintsymbol `max` implementiert:

```
max(X,Y,Z) ⇔ X≤Y | Z=Y.
max(X,Y,Z) ⇔ Y≤X | Z=X.
```

Dieses Programm kann nicht in ein äquivalentes deterministisches Programm transformiert werden, wenn $\doteq$ und $\leq$ die einzigen vordefinierten Constraintsymbole sind, die zur Verfügung gestellt werden.

- Konfluente Programme sind meistens effizienter als deterministische Programme, da eine Regel früher angewendet werden kann, weil die Wächter wegen der Überlappung öfter erfüllt werden können.

  Beispiel 8.2 Eine deterministische Version von `max` aus Beispiel 8.1 kann wie folgt aussehen

  ```
  max(X,Y,Z)  ⇔  X<Y | Z=Y.
  max(X,Y,Z)  ⇔  Y≤X | Z=X.
  ```

  Das Ziel `max(A,B,C) ∧ A≤B` kann nicht reduziert werden, da keine Regel anwendbar ist. In der konfluenten Version kann die erste Regel angewendet werden und das Antwortconstraint lautet dann `A≤B ∧ A=C`.

- Im Gegensatz zu den meisten NCLP-Sprachen erlaubt SiR-Programme mehrköpfige Regeln (Konjunktionen im Kopf der Regeln). Wir erweitern für unseren Formalismus den Determinismusbegriff, so daß für ein gegebenes Ziel eine einzige Regel angewendet werden kann. Wir können in eine Situation geraten, daß zwei Regeln auf verschiedene, aber überlappende Constraints angewendet werden können. Im allgemeinen ist es nicht möglich, die Auswahl einer der Regeln zu verhindern und so das Programm deterministisch zu machen, indem zusätzliche Constraints zu den Wächtern eingefügt werden.

  Beispiel 8.3 Gegeben sei ein Ausschnitt des Booleschen Constraintlösers:

  ```
  neg(X,Y) ∧ imp(X,Y)  ⇔  X=0 ∧ Y=1.
  neg(X,Y) ∧ or(X,Z,Y)  ⇔  X=0 ∧ Y=1 ∧ Z=1.
  ```

  Sei `neg(A,B)∧imp(A,B)∧or(A,C,B)` ein Ziel . Beide Regeln sind anwendbar. Um sicherzustellen, daß nur die erste Regel angewendet werden kann, mußte die Bedingung, daß das Constraint `or(A,C,B)` im aktuellen Zustand nicht existiert, zu dem Wächter der ersten Regel eingefügt werden. Eine solche Bedingung kann nicht durch Constraints ausgedrückt werden, da sie metalogisch ist und außerdem vom Kontext abhängig.

Wenn man zusätzlich die Propagationsregeln miteinbeziehen will, dann treten ähnliche Probleme wie bei Termersetzungssystemen auf: Eine naive Übersetzung von Propagationsregeln in NCLP-Klauseln, indem man den Kopf der Klausel im Rumpf wiederholt, führt zu einem nicht terminierenden Programm.

Beispiel 8.4 Gegeben sei folgende Propagationsregel, die besagt, daß das Maximum zweier Zahlen größer oder gleich den beiden Zahlen ist.

```
r @ maximum(X,Y,Z)  ⇒ X ≤ Z ∧ Y ≤ Z.
```

Eine naive Übersetzung von `r` als NCLP-Klausel liefert `maximum(X,Y,Z) ← maximum(X,Y,Z) ∧ X ≤ Z ∧ Y ≤ Z`. Diese Klausel ist nach der operationalen Semantik von NCLP-Sprachen immer wieder anwendbar.

In den letzten Jahren gab es etliche Untersuchungen zur Konfluenz im Bereich von nebenläufigen Constraintsprachen [dBGMP94, FGMP95, MO95]. Diese Arbeiten basieren auf dem vereinheitlichten Modell (cc-Sprachfamilie) von Saraswat [Sar92, Sar93].

Auf den ersten Blick fällt auf, daß cc-Sprachen einem wesentlich operationaleren Paradigma entstammen als z.B. CHR, das ja eine deklarative Sprache ist. Im Gegensatz zu der in Abschnitt 2.5

vorgestellten Syntax und Semantik von NCLP-Sprachen, die sich an der Logik orientieren, basiert die cc-Sprachfamilie auf der Prozeßalgebra. Die zwei in diesem Zusammenhang grundlegenden Konnektoren im cc-Formalismus sind der Paralleloperator $A_1 \parallel A_2$ und die Disjunktion („guarded choice") $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$. Bei der operationalen Auswertung dieser Konstrukte sieht man aber die Ähnlichkeit zu CHR und NCLP-Sprachen, die in Abschnitt 2.5 vorgestellt wurden: Der Paralleloperator entspricht der Konjunktion $\wedge$, die Disjunktion hat eine Entsprechung in dem Nichtdeterminismus bei der Regelauswahl, wobei die $ask(c_i)$ den Wächtern und die A_i den Regelrümpfen entsprechen.

In [FGMP95] wird der Begriff der *konfluenten* cc-*Programme* eingeführt. Dort ist der Konfluenzbegriff jedoch schwächer als in dieser Arbeit: Ein cc-Programm heißt konfluent, wenn jede Selektionsstrategie bei der Bearbeitung des Paralleloperators zu derselben Menge von *beobachtbaren Antworten* (observables) führt. Die Menge der beobachtbaren Antworten ist die Menge aller Antworten, die durch verschiedene Strategien bei der Bearbeitung der Disjunktion zustande kommen können. Der Nichtdeterminismus, der ungefähr der Regelauswahl bei CHR entspricht, wird also von dem Konfluenzbegriff in [FGMP95] ignoriert, indem die gesamten Mengen der beobachtbaren Antworten verglichen werden. Diese Definition wird auch in den Arbeiten von [MO95, CFMW97] verwendet.

Für die konfluenten cc-Programme können die Autoren von [FGMP95] eine einfache und natürliche denotationale Semantik für cc-Programme geben. Eine hinreichende Bedingung für die Konfluenz wird durch Angabe einer Teilklasse der konfluenten Programme gegeben, der *unabhängigen* cc-*Programme* (admissible cc programs). Dies sind die cc-Programme, bei denen für die Wächter einer Disjunktion gilt: Sie sind identisch oder alle Paare von Agenten, die sich aus der Konjunktion der Wächter und den zwei Regelrümpfen ergeben, sind kongruent. Zwei Agenten (Ziele) heißen *kongruent*, wenn sie unter jedem Kontext zu derselben Menge von beobachtbaren Antworten führen. Hier erkennt man eine sehr starke Parallelität zu der vorliegenden Arbeit: Die Paare von Agenten entsprechen den kritischen Paaren dieser Arbeit. Die Kongruenz zweier solcher Agenten bedeutet, daß diese zwei Zustände (Agenten) in jedem spezielleren Kontext zusammenführbar sein müssen, spiegelt also den in Abbildung 4.2 dargestellten Übergang von allgemeiner Situation zu spezieller Situation wieder. Die Autoren erwähnen ohne Angabe von Gründen, daß die Kongruenz zweier Agenten im allgemeinen nicht entscheidbar ist. Zwei Gründe kann man nach den in dieser Arbeit vorgestellten Überlegungen angeben:

- Zusammenführbarkeit, also Kongruenz kann überhaupt nur für terminierende Programme entscheidbar sein.
- Dem cc-Formalismus im allgemeinen fehlt die „Kontextmonotonie", d.h. die Invarianz von Berechnungssequenzen gegenüber der Transformation in einen neuen, spezielleren Kontext. In [SS93] wird gezeigt, daß der **atomic tell**-Operator mit der Kontextmonotonie nicht vereinbar ist.

Kapitel 9

Schlußbemerkungen

9.1 Zusammenfassung der Arbeit

In dieser Arbeit wurde die Sprache CHR untersucht, die dem Benutzer erlaubt, Constraintlöser in einer höheren Sprache zu implementieren. Nach einer formalen Darstellung der Auswertung von CHR-Programmen wurde für die operationale Semantik aus Abschnitt 3.2 eine Korrektheits- und eine Vollständigkeitsaussage bewiesen.

Eine wichtige Eigenschaft von Constraintlösern ist die Konfluenz, die sicherstellt, daß ein CHR-Programm ein Ziel unabhängig von der Regelauswahl zu demselben Ergebnis vereinfachen kann. In dieser Arbeit wurde ein notwendiges, hinreichendes und entscheidbares syntaktisches Kriterium für die Konfluenz von terminierenden CHR-Programmen entwickelt. Dieses Kriterium ist die Zusammenführbarkeit aller kritischen Paare des Programmes. Die Konfluenz ist besonders bei nebenläufigen Anwendungen sehr wichtig, da sie sicherstellt, daß die Antwort gleich ist, unabhängig von der Reihenfolge, in der die Constraints dem Constraintlöser übergeben werden. Bei der semantischen Analyse von CHR-Programmen kann man einerseits die Vollständigkeitssätze für konfluente Programme verbessern, und andererseits stellt die Konfluenz eines bereichsbeschränkten CHR-Programms sicher, daß es auch konsistent ist.

Aufbauend auf der Konfluenz wurde dann ein Verfahren vorgestellt, das terminierende, nicht-konfluente und bereichsbeschränkte CHR-Programme zu äquivalenten terminierenden und konfluenten vervollständigen kann. Dieses Verfahren basiert auf dem Vervollständigungsverfahren von Knuth und Bendix für Termersetzungssysteme.

Oftmals hat man dabei aber das Problem, daß kritische Paare nicht zusammenführbar sind, da zum einen semantisch äquivalente, aber syntaktisch verschiedene Constraints nicht identifiziert werden können, es zum anderen aber auch nicht möglich ist, CHR-Regeln hinzuzufügen, weil das die Terminierung kosten würde. Deswegen wurde in dieser Arbeit ein Ansatz vorgestellt, der die Möglichkeit schafft, eine Äquivalenz auf Constraints zu definieren. Diese Äquivalenz drückt meistens die Permutativität gewisser Constraintsymbole aus. Dazu wurde bei der Auswertung von CHR das Matching um das Matching modulo Permutativität erweitert, und die Konfluenz und die Vervollständigung modulo dieser Äquivalenz untersucht.

9.2 Ausblick

Im folgenden wird eine Auswahl von Arbeitsgebieten angegeben, deren Bearbeitung dazu beitragen würde, dem Benutzer mehr Werkzeuge zur Verfügung zu stellen, um die in CHR geschriebenen Constraintlöser besser zu analysieren.

Äquivalenz von CHR-Programmen

In Abschnitt 6.2 haben wir einen Ansatz kennengelernt, um die Verträglichkeit von Constraints zu untersuchen. Manchmal hat man es aber auch mit äquivalenten Constraintlösern zu tun, die auf verschiedene Arten implementiert sind und man will natürlich den effizienteren auswählen. Dem Benutzer muß also ein Werkzeug zur Verfügung gestellt werden, mit dem er die operationale Äquivalenz von zwei CHR-Programmen feststellen kann.

Definition 9.1 Seien P_1, P_2 zwei CHR-Programme. P_2 *subsumiert* P_1 *operational*, falls für jeden Zustand S gilt: wenn es einen Endzustand S_1 gibt mit $S \mapsto^*_{P_1} S_1$, dann gibt es einen Endzustand S_2 mit $S \mapsto^*_{P_2} S_2$, wobei S_1, S_2 Varianten sind.

Definition 9.2 Zwei CHR-Programme P_1, P_2 sind *operational äquivalent*, falls P_1 von P_2 operational subsumiert wird und umgekehrt.

Die Idee ist, daß man sich wie beim Überprüfen der lokalen Konfluenz auf eine endliche Menge von Zuständen beschränkt, um die operationale Äquivalenz von zwei CHR-Programmen festzustellen. Diese Zustände werden aus den Regelköpfen und den Wächtern gebildet.

Definition 9.3 Sei P ein CHR-Programm. Dann ist die Menge der *Äquivalenzzustände* von P wie folgt definiert:

$$\begin{array}{rcl} M_P & := & \{\langle H, \top, G, \emptyset, \mathcal{V}\rangle \mid (R@H \;\Leftrightarrow\; G \mid B) \text{ eine Simplifikationsregel aus } P\} \\ & \cup & \{\langle H_1 \wedge H_2, \top, G, \emptyset, \mathcal{V}\rangle \mid (R@H_1 \backslash H_2 \;\Leftrightarrow\; G \mid B) \text{ eine Simpagationsregel aus } P\} \\ & \cup & \{\langle H, \top, G, \emptyset, \mathcal{V}\rangle \mid (R@H \;\Rightarrow\; G \mid B) \text{ eine Propagationsregel aus } P\} \end{array}$$

Definition 9.4 Seien P_1 und P_2 zwei CHR-Programme. Ein Zustand S ist (P_1, P_2)*-zusammenführbar*, falls es zwei variante Zustände S_1, S_2 gibt, sodaß $S \mapsto^*_{P_1} S_1$ und $S \mapsto^*_{P_2} S_2$.

Ein möglicher Satz, der noch zu beweisen ist, wäre dann:

Hypothese
Seien P_1 und P_2 CHR-Programme. P_1 und P_2 sind operational äquivalent, wenn alle Zustände aus $M_{P_1} \cup M_{P_2}$ (P_1, P_2)-zusammenführbar sind.

Wenn man sich auf terminierende und konfluente CHR-Programme einschränkt, dann ist die Überprüfung der operationalen Äquivalenz einerseits entscheidbar, andererseits einfacher, da ein Ziel eine eindeutige Normalform hat. Außerdem kann man die Effizienz der Programme vergleichen, indem man beim Zusammenführbarkeitstest die Anzahl der Berechnungsschritte verwaltet.

Man kann sich dann überlegen, unter welchen Kriterien zwei semantisch äquivalente CHR-Programme operational äquivalent sind. Ein mögliches Kriterium wäre die Untersuchung von CHR-Programmen mit einer unabhängigen Constrainttheorie. Das folgende Beispiel zeigt, daß die semantische Äquivalenz von CHR-Programmen die operationale Äquivalenz nicht impliziert.

Beispiel 9.5 Das Constraint `max` wird in zwei verschiedenen Constraintlösern definiert. In beiden Constraintlösern bedeutet `max(X,Y,Max)`, daß `Max` das Maximum von `X` und `Y` ist.

Sei P_1 das CHR-Programm, das nur über die vordefinierten Constraints $<, =$ und $\geq$ verfügt:

```
max(X,Y,Z) ⇔ X < Y | Z=Y.
max(X,Y,Z) ⇔ X ≥ Y | Z=X.
```

Und sei P_2 das CHR-Programm, das nur über die vordefinierten Constraints $>, =$ und $\leq$ verfügt:

```
max(X,Y,Z) ⇔ X ≤ Y | Z=Y.
max(X,Y,Z) ⇔ X > Y | Z=X.
```

Wie man leicht sieht, sind P_1 und P_2 semantisch äquivalent, da die deklarativen Leseweisen beider Programme äquivalent sind. Sie sind aber nicht operational äquivalent, da $\langle\top, \mathtt{max(X,Y,Z)}, \mathtt{X} \geq \mathtt{Y}, \emptyset, [\mathtt{X},\mathtt{Y},\mathtt{Z}]\rangle$ mit P_2 nicht vereinfacht werden kann und mit P_1 zu $\langle\top, \top, \mathtt{Z} = \mathtt{X}, \emptyset, [\mathtt{X},\mathtt{Y},\mathtt{Z}]\rangle$ reduziert werden kann. Im folgenden werden nur die benutzerdefinierten und vordefinierten Speicher explizit angegeben:

$$
\begin{tikzcd}
\mathtt{max(X,Y,Z), X < Y} \arrow[d, bend right, "P_1"'] \arrow[d, bend left, "P_2"] \\
\mathtt{Z = Y}
\end{tikzcd}
\quad
\begin{tikzcd}
\mathtt{max(X,Y,Z), X \geq Y} \arrow[d, "P_1"] \\
\mathtt{Z = X}
\end{tikzcd}
\quad
\begin{tikzcd}
\mathtt{max(X,Y,Z), X > Y} \arrow[d, bend right, "P_1"'] \arrow[d, bend left, "P_2"] \\
\mathtt{Z = X}
\end{tikzcd}
\quad
\begin{tikzcd}
\mathtt{max(X,Y,Z), X \leq Y} \arrow[d, "P_2"] \\
\mathtt{Z = Y}
\end{tikzcd}
$$

Terminierung von CHR-Programmen

Eine weitere wichtige Eigenschaft eines Constraintlösers, die in dieser Arbeit vorausgesetzt wurde, ist die Terminierung. In Abschnitt 5.1 wird erläutert, daß eine Übertragung entsprechender Arbeiten aus dem Bereich Termersetzungssysteme keine zufriedenstellenden Ergebnisse liefern kann. Außerdem ist es zu erwarten, daß die Wechselwirkung zwischen den Simplifikations- (bzw. Simpagations-) und Propagationsregeln bei der Terminierung ähnliche Schwierigkeiten verursachen wird wie bei der Konfluenz. Erste Ansätze zur Terminierung finden sich in [Frü97].

Vervollständigung von allgemeinen CHR-Programmen

In Abschnitt 5 wurde untersucht, inwieweit die Automatisierung des Vervollständigungsverfahren von Knuth und Bendix auch auf CHR zu übertragen ist. Es wurde ein Ansatz vorgestellt, das ein terminierendes und nicht-konfluentes CHR-Programm zu einem äquivalenten terminierenden und konfluenten Programm vervollständigen kann. Dies gilt aber nur für bereichbeschränkte CHR-Programme. Ein weiteres interessantes Thema, das an die vorliegende Arbeit anknüpft, wäre dann neue Ansätze zu untersuchen, die für allgemeine CHR-Programme gelten.

9.3 Perspektiven der Constraint-Programmierung

Seit Anfang der neunziger Jahre wird Constraint-Programmierung mit Erfolg von mehreren Firmen (Ilog mit Numerica, IlogSolver und IlogSchedule, Cosytec mit CHIP 4, Siemens Nixdorf mit IF/Prolog, PrologIA mit Prolog IV) weltweit kommerziell eingesetzt. Der gemeinsame Umsatz mit Constrainttechnologie wurde 1996 auf 100 Millionen US-Dollar geschätzt. Dieses Gebiet hat von Anfang an in Forschung und Praxis gleichermaßen für Aktivität gesorgt.

Vorteile des Einsatzes von Constraint-Programmierung sind

- die deklarative Modellierung von Problemen mit Hilfe von Constraintsystemen, die schnell zu robuster, flexibler und wartbarer Software führt,
- das Schließen mit solchen Informationen und die automatische Propagierung der Effekte, wenn neue Information (in Form von Constraints) bekannt wird, z.B. die Berechnung der Konsequenzen einer Entscheidung,
- die gute Kombinierbarkeit von Constraintlösen mit Such- und Optimierungsverfahren zur Lösung kombinatorischer Probleme

In [Wal96] wird der bisherige Erfolg von Constraints in Bezug auf kommerzielle Anwendungen untersucht, in dem die obigen drei Merkmale der Constraint-Programmierung in Betracht gezogen werden. Die dort genannten industriellen Anwendungsgebiete der Constrainttechnologie sind zum einen die Modellierung, (ausführbare) Spezifikation, Design, Synthese, Simulation, Verifikation und Fehlerdiagnose von elektronischen, elektrischen und mechanischen Komponenten und ganzen industriellen Abläufen, von Computer-Hardware und Softwarekomponenten und zum anderen Planung, Logistik und Management von Produktion, Personal, Finanzen, Verkehr, Netzwerken und Ressourcen (insbesondere in der Zeit- und Kapazitätswirtschaft) sowie Transport- und Plazierungsoptimierung, Design, Konfiguration und Layoutgenerierung.

Bei der Programmierung einer Anwendung muß man zwischen zwei Implementierungsebenen unterscheiden:

1. Eine Anwendungssprache, mit der man die Anforderungen der Anwendung implementiert.

2. Eine Constraintsprache, die die in einer Anwendung vorkommenden Constraints behandelt.

Die meisten kommerziellen Anwendungen werden mit Constraintsprachen implementiert, die als Blackbox-System betrachtet werden. Die Anwendungssprache ist meistens eine imperative Sprache (oft C). Obwohl man damit viele Probleme schnell und effizient lösen kann, können Anwendungen vorliegen, die mit einem existierenden Constraintlöser nicht direkt implementierbar sind. Außerdem ist die Anpassung einer Anwendung, die in C geschrieben ist, meistens aufwendiger als das Ändern eines Logikpogramms. Für solche Anwendungen eignet sich eine CLP-Sprache, wobei das Constraintlösen mit Hilfe einer regelbasierten Sprache wie CHR erfolgt [FA97a].

Im folgenden stellen wir einige Anwendungen von Constraints sowohl in der Praxis als auch in der Forschung genauer vor, die bestätigen, daß Constraint-Programmierung eine allgemeine Methode für elegantes, effizientes und deklaratives Problemlösen mit höheren Programmiersprachen ist. Die Beispiele zeigen, daß CLP mit regelbasierten Sprachen wie CHR ein noch größeres Potential haben als die Blackbox-Systeme, so daß sich der bisher schon beachtliche kommerzielle Erfolg der CLP noch verstärken könnte.

9.3.1 Modellierung durch Constraints

Im folgenden stellen wir eine Anwendung vor, die die Vorteile der Modellierung von Problemen durch ein Logikprogramm und der Implementierung der in dieser Anwendung vorkommenden Constraints durch eine regelbasierte Sprache bestätigt.

Der Münchener Mietspiegel Online

Mit einem Mietspiegel läßt sich eine Vergleichsmiete zu einer Mietwohnung errechnen. Diese Vergleichsmiete ist in gerichtlich zu klärenden Rechtsstreitigkeiten zu Mietfragen als Beweismittel zugelassen. Der Mietspiegelberechnungsdienst im Internet (IMS) [ABF96, FA96, FA97c] ist ein Beispiel für ein constraintbasiertes Entscheidungsunterstützungssystem. Er erlaubt es, durch eine Formularseite im World Wide Web in wenigen Sekunden die ortsübliche Vergleichsmiete einer Wohnung zu berechnen. Mittels Constraints ist der IMS sogar in der Lage, mit ungenauen und unvollständigen Angaben und der statistischen Unschärfe korrekt umzugehen. Damit läßt sich erstmals ein Mietspiegel auch dazu verwenden, bei der Wohnungsuche das Mietpreisniveau zu bestimmen, ohne sich auf eine bestimmte Wohnung festlegen zu müssen.

Die Idee war, zuerst die Tabellen, Regeln und Formeln der Papierversion des Mietspiegels unter der Annahme zu programmieren, daß alle notwendigen Angaben zu Verfügung stehen. Wegen der Deklarativität der Constraintlogikprogrammierung war die Implementierung in wenigen Wochen möglich: Tabellen lassen sich durch Fakten, Regeln durch CLP-Regeln darstellen. Dann haben wir Constraints eingeführt, um die Unschärfe (wegen des statistischen Modells) und Unvollständigkeit (wegen ungenauer oder unvollständiger Benutzerangaben) zu berücksichtigen. Dabei werden die Formeln zur Vergleichsmietenberechnung nun als Constraints betrachtet, der Rest des Programms blieb praktisch unverändert. Das Verhalten dieser Constraints wurde in einem Constraintlöser spezifiert. Es genügte, dazu den existierenden Constraintlöser über endlichen Bereichen aus der CHR-Bibliothek zu erweitern, daß er auch Intervallconstraints über nichtlinearen (Un)Gleichungen behandeln kann. Diese Gleichungen und Ungleichungen zwischen zwei Variablen oder Werten sind notwendig, um die im Mietspiegel vorhandenen Formeln zur Berechnung der Nebenkosten und Vergleichsmiete auszudrücken. Der Constraintlöser besteht aus ca. 20 Zeilen Code.

Derzeit sind für rund 150 größere Städte Deutschlands solche Mietspiegel (in Papierform) erstellt worden. Die Verwendung einer CLP-Sprache ermöglicht eine einfache Wartung und Anpassung des IMS-Programms an neue Mietspiegel. Außerdem ist Vergleichsmietenberechnung von einem Mietspiegel zu einem anderen verschieden. Hier ist eine Anpassung oder Modifizierung des Constraintlösers nötig. Dies ist sehr leicht, da der Constraintlöser mit einer regelbasierten Sprache implementiert ist.

9.3.2 Propagierung durch Constraints

Ein Constraintlöser hat die Aufgabe, Wertebereichsänderungen von einer Variablen zur nächsten zu propagieren. Damit lassen sich die Effekte der Änderung eines Parameters auf alle anderen Parameter in Verallgemeinerung einer Tabellenkalkulation (Spreadsheet) unmittelbar berechnen, sichtbar machen und zur Entscheidungsfindung studieren. Im Unterschied zu einer Tabellenkalkulation kann man mit Constraints aber in beliebige Richtungen rechnen und dies mit ungenauen und unvollständigen Angaben bzw. Daten. Die Propagierung arbeitet sehr effektiv vor allem in interaktiven Entscheidungsunterstützungssystemen. Sie schließt von vornherein diejenigen Werte aus, die in keiner Lösung vorkommen können. Jedes Constraint versucht also dynamisch, den Suchraum zu reduzieren. Ein typisches Beispiel eines Constraint-basierten Spreadsheets ist das „Short Term Production Planning“-System von Renault. Die Autofirma setzt seit 1995 dieses System ein, zur optimalen Planung der Zulieferung und Fertigung von Varianten eines Autotyps innerhalb eines Fertigungsabschnittes.

Wir werden im folgenden eine Anwendung der Propagierung durch Constraints vorstellen, die im Gebiet automatischer Deduktion Verwendung findet.

CPUHR-Tableaux

Für viele Anwendungen im Bereich der automatischen Deduktion sind Tableau-Verfahren besonders geeignet, da nicht nur die Inkonsistenz einer Menge festgestellt wird, sondern bei Konsistenz auch Modelle generiert werden. Diese Modelle können gegebenfalls dann als Erklärungen oder Gegenbeispiele dienen. Ausgehend von der „Positiv Unit Hyper Resolution“-Methode (PUHR) [BY96], die eine Formalisierung des Modellsgenerators SATCHMO [MB88, ABEG95] ist, wurde die CPUHR-Tableau-Methode [AS97b] entwickelt. Das Ziel war dabei, erstens auf die Skolemisierung der Formeln zu verzichten und zweitens Constraints zu erlauben. Dies ist sehr nützlich im Hinblick auf mögliche Anwendungen etwa im Bereich Integritätsüberprüfung in Datenbanken [BT97].

Im Gegensatz zu den PUHR-Tableaux, die nur bereichsbeschränkte Klauseln behandeln können, kann der CPUHR-Kalkül auch Theorien behanden, die Formeln folgender From haben:

$$\forall\overline{X}(b_1 \wedge \cdots \wedge b_l \rightarrow \exists\overline{Y}(h_1 \vee \cdots \vee h_m \vee c_1 \vee \cdots \vee c_n))$$

- b_i und h_j sind Atome ohne Constraintsymbole. c_k sind Constraintsatome.
- l, m, n können 0 sein.
- Jede freie Variable der Konsequenz kommt in der Bedingung der Implikation vor.

Wir werden die CPUHR-Tableau-Methode, die genauso wie die PUHR-Tableau-Methode auf Vorwärtsschließen basiert, anhand eines Beispiels erklären.

Beispiel 9.6 Gegeben seien Informationen über eine Universität und einen Studenten: Wenn die Vorlesung `X` Voraussetzung für die Vorlesung `Y` ist, dann muß ein Student, der `Y` zu einem Zeitpunkt `TY` hört, auch `X` zu einem Zeitpunkt `TX` gehört haben, wobei `TX < TY` sein soll. `john` hat nur Vorlesungen vor `1994` gehört und dann ab `1996`. Er hört die Vorlesung `constraintprogrammierung` im Jahr `1996`. `constraintprogrammierung` hat als Voraussetzung die Vorlesung `logik`.

```
1. hoert(S, Y, TY), voraussetzung(Y, X) -> exists TX: hoert(S, X, TX), TX < TY.
2. hoert(john,C,T)  -> T < 1994 ∨ T ≥ 1996.
3. hoert(john,constraintprogrammierung,1996).
4. voraussetzung(constraintprogrammierung,logik)
```

```
                         o
                         |
   hoert(john,constraintprogrammierung,1996)
                         |
 voraussetzung(constraintprogrammierung,logik)
                         |
               hoert(john,logik,TX)
                    TX < 1996
                   /         \
            TX < 1994     TX ≥ 1996
                              |
                              ⊥
```

Abbildung 9.1: Das generierte Tableau

Dabei sind `<` und $\geq$ Constraintssymbole.

Das generierte Tableau für diese Theorie ist in Abbildung 9.1 dargestellt. Zuerst werden die Fakten 3 und 4 der Theorie eingefügt. Eine Instanz der Regel 1 kann feuern und wir fügen `hoert(john,logik,T')` und `T' < 1996` ein. Dies läßt eine Instanz der Regel 2 feuern und man unterscheidet hier zwei Fälle, die den Disjunkten der rechten Seite der Implikation entsprechen. Im rechten Fall haben wir einen Widerspruch zwischen den Constraints `T' < 1996` und `T'`$\geq$`1996` und der Ast ist geschlossen. Wir markieren dies durch das Symbol „$\perp$“.

Im linken Fall ist keine Regel mehr anwendbar, da keine neue Information eingefügt werden kann. Zum Beispiel kann Regel 2 durch die Fakten 3 und 4 nicht aktiviert werden, da `T` zu `1996` instantiiert würde. Dies führt aber zu einer Tautologie, da `1996`$\geq$`1996` trivialerweise immer wahr ist. Der Constraintlöser sammelt und vereinfacht die Constraints, die in einem Ast vorkommen und man bekommt dann als Modell eine Menge von Atomen:

$$\begin{aligned} A = \{&\mathtt{hoert}(\mathtt{john}, \mathtt{constraintprogrammierung}, \mathtt{1996}),\\ &\mathtt{voraussetzung}(\mathtt{constraintprogrammierung}, \mathtt{logik}),\\ &\mathtt{hoert}(\mathtt{john}, \mathtt{logik}, \mathtt{T}')\} \end{aligned}$$

und eine Menge von Constraints

$$C = \{\mathtt{T}' < \mathtt{1994}\}.$$

Dies bedeutet dann, daß die Bedingungen des Beispiels 9.6 konsistent sind unter der Bedingung, daß `john` die Vorlesung `logik` in einem Zeitraum vor `1994` gehört hat. Jede Belegung der Variable `T'`, die C erfüllt, kann auf A angewandt werden. Dies führt (zusammen mit einer unendlichen Menge von Atomen für die Constrainttheorie) zu einem Herbrandmodell der Theorie (und der Constrainttheorie).

Hier konnte durch den Einsatz von CLP eine bekannte Methode erweitert werden. Maßgeblich dafür war die Verfügbarkeit der Regelsprache. Mit einem Blackbox-System wäre dies nicht möglich gewesen.

9.3.3 Lösen kombinatorischer Probleme durch Constraints

Das wichtigste Gebiet der kommerziellen Anwendung von Constraints ist das Lösen von kombinatorischen Problemen. Sie sind Bestandteil vieler industrieller Anwendungen, wie zum Beispiel Scheduling, Konfiguration oder Stundenplanerstellung. Aus einer großen Menge von möglichen Wertekombinationen gilt es diejenigen zu bestimmen, die gewisse Constraints erfüllen (oder besonders gut erfüllen). Klassische Ansätze zur Lösung solcher Probleme leiden oft an mangelnder Flexibilität und Erweiterbarkeit. Der Erfolg der Constraint-Programmierung bei der Lösung solcher Optimierungsprobleme beruht auf der deklarativen Modellierung, der Constraintpropagierung und der guten Kombinierbarkeit mit Such- und Optimierungsverfahren.

Der Dienstplaner für Krankenstationen

Immer noch werden an den meisten Krankenhäusern die Dienstpläne von Hand erstellt. Dabei wird für jedes Mitglied einer Station und für jeden Tag des Planungszeitraums – üblicherweise ein Monat – eine oder keine Schicht zugeteilt, wofür verschiedene „harte“ Bedingungen unbedingt erfüllt werden müssen, und „weiche“ Bedingungen nach Möglichkeit. Eine solche Planungstätigkeit ist, speziell bei größeren Krankenstationen, sehr aufwendig: typischerweise benötigt eine Person einen ganzen Tag zur Planung eines Monats.

Die automatische Generierung von Dienstplänen für Krankenstationen ist heute noch Gegenstand der Forschung. Bei der Erzeugung eines Dienstplans müssen unterschiedliche Constraints berücksichtigt werden. Man unterscheidet verschiedene Arten von Constraints:

- Gesetze regeln unter anderem die maximale Arbeitszeit einer Person pro Tag und pro Woche, Freizeitausgleich oder Mutterschaftsurlaub.
- Dienstliche Regelungen, die für ein spezielles Krankenhaus, einen Teil des Krankenhauses oder auch nur für eine Station gelten.
- Personaldaten geben den individuellen Rahmen für eine Person. Dazu zählen vor allem die vertraglich feststehende monatliche Arbeitszeit, noch ausstehender Urlaub und angefallene Überstunden.
- Wünsche sind schließlich Anforderungen an den Dienstplan, die das Personal selbst formuliert. Meist sind das Freiwünsche, z.B. für Wochenenden, Feiertage oder Urlaub.

Aufgrund des immens großen Suchraums wird ein Dienstplan üblicherweise von Hand in zwei Phasen erzeugt. Die Frei- und die Nachtschichten werden in der ersten Phase zugeordnet. In der zweiten Phase werden dann Früh- und Spätschichten auf die noch nicht zugeordneten Zellen des Dienstplans verteilt.

Das interaktive constraintbasierte System INTERDIP [AS97a] erlaubt es, Teillösungen als endgültig festzulegen, was einerseits die Komplexität einschränkt, andererseits die Akzeptanz durch die Benutzer erhöht. Für 20 Schwestern und 30 Tage kann INTERDIP einen Dienstplan teilautomatisch und interaktiv innerhalb weniger Minuten (statt von Hand einiger Stunden) erstellen. Bei der Erzeugung eines Dienstplans kann man nicht immer davon ausgehen, daß alle Constraints erfüllt werden. Deswegen unterscheidet man zwei Arten von Constraints. „Harte“ Constraints müssen immer erfüllt werden, „weiche“ Constraints können verletzt werden. Der verwendete IF/Prolog-Constraintlöser ist unanpassbar und bietet keine Unterstützung für weiche Constraints. Die Idee war dann, die weichen Constraints nur bei der Suche nach einem „guten“ Dienstplan zu verwenden. Jedem weichen Constraint werden Kosten zugeordnet und dann wird ein Dienstplan generiert, der alle harten Constraints erfüllt. Durch ein Branch-and-Bound-Verfahren wird dann dieser Plan unter Berücksichtigung der weichen Constraints optimiert.

Mit CHR können dagegen Constraintlöser geschrieben werden, die harte und weiche Constraints gemeinsam behandeln. Diese Erfahrung mit INTERDIP ist die Motivation für den Instituts-Stundenplaner.

Der Stundenplaner für das Institut für Informatik

Die Erstellung eines Stunden- und Raumplans am Institut für Informatik läuft wie folgt ab: Ausgehend vom Stundenplan des letzten Jahres wird unter Berücksichtigung von Wünschen der Dozenten ein erster Stundenplan erstellt. Dieser wird zusammen mit Angaben der benötigten Größe und technischen Ausstattung der Räume an die zentrale Verwaltung gegeben. Die zentrale Verwaltung erstellt dann einen ersten Plan, der den Veranstaltungen auch Räume zuweist. Dabei wird der Stundenplan nicht verändert, doch kann es vorkommen, daß die zugewiesenen Räume nicht den Erfordernissen entsprechen. In diesem Fall muß nach Rücksprache mit der zentralen Verwaltung und den betroffenen Dozenten der Stundenplan geändert werden.

Für die Stundenplanung ist ebenfalls eine Vielfalt von Bedingungen zu berücksichtigen:

- Im Grundstudium darf es keine Überschneidung bei den Pflichtveranstaltungen der Informatik und Mathematik geben. Auch sollen sich Wahl- und Pflichtveranstaltungen nicht überschneiden.
- Im Hauptstudium sollte es keine Überschneidungen zwischen Veranstaltungen der Typen einer Stufe der folgenen Hierarchie geben:
  1. Vorlesungen und Praktika
  2. Übungen und Hauptseminare
  3. Oberseminare
- Überschneidungen zwischen Veranstaltungen der Typen verschiedener Stufen sind zulässig.
- Veranstaltungen sollten zwischen neun und 18 Uhr stattfinden.
- Keine Veranstaltungen für Professoren am Mittwoch nachmittag.
- Die zwei Blöcke einer vierstündigen Vorlesung müssen an verschiedenen Tagen stattfinden. Raum und Beginn sollten gleich sein.
- Veranstaltungen eines Lehrstuhls sollten sich nicht überschneiden. Pro Einheit sollte möglichst nur eine Vorlesung pro Tag stattfinden.
- Wünsche der Dozenten, z.B. keine Veranstaltung am Montag.

Wie bei INTERDIP ahmt der Stundenplaner die manuelle Planung nach. Er erzeugt für das Institut für Informatik, jeweils ausgehend vom Stundenplan des vorhergehenden Semesters, einen neuen für das kommende. Mit Papier und Bleistift braucht man leicht einen Tag, um einen Stundenplan zu generieren. Der Stundenplaner kann die Berechnungszeit auf wenige Minuten reduzieren [AM97]. Man unterscheidet auch hier zwischen harten und weichen Constraints. Zum Beispiel sind die Wünsche der Dozenten oftmals ausschlaggebend: Ein Dozentenwunsch ist wichtiger als die Forderung, daß Veranstaltungen nur zwischen neun und 18 Uhr stattfinden sollten. Außerdem ist die Forderung, daß sich die Veranstaltungen einer Einheit nicht überschneiden sollten, wichtiger als ein Dozentenwunsch.

Um die weichen Constraints behandeln zu können, wurde der CHR-Constraintlöser über endlichen Bereichen so modifiziert, daß jeder Wert im Domain einer Variable mit einer Präferenz vermerkt ist. Die Constraint-Sprache CHR erwies sich zur Behandlung der weichen Constraints aus zwei Gründen als besonders geeignet: Zum einen konnte unter dem Einsatz von CHR die Behandlung der harten und weichen Constraints spezifiziert werden. Zum anderen besteht das CHR-Programm aus nur ca. 10 Zeilen Code.

Die oben vorgestellten Anwendungen zeigen, daß die Constraint-Sprache CHR sich zur einfachen Programmierung beziehungsweise „rapid prototyping“ eignet, ohne daß die Effizienz beeinträchtigt wird.

Anhang A

Grundlagen der Logik und der Logikprogrammierung

A.1 Prädikatenlogik erster Stufe

Der erste Abschnitt fixiert die Syntax der Prädikatenlogik erster Stufe, d.h. die verwendete Sprache. Die Semantik, also die Bedeutung der Sprache, wird im zweiten Abschnitt definiert. Für eine ausführliche Darstellung wird auf [HJW78] und [Men87] verwiesen. Die meisten Definitionen dieses Abschnittes sind aus [Abd96] und [FA97b] entnommen.

A.1.1 Syntax

Definition A.1 Das *Alphabet* für eine Sprache der Prädikatenlogik erster Stufe besteht aus den Mengen:

- $\mathcal{P}$ eine abzählbare Menge von *Prädikatensymbolen* mit Stelligkeit: Sie werden mit $p, q, r, \ldots$ bezeichnet. Ist ein Prädikatensymbol p von der Stelligkeit n, schreiben wir auch p/n. Nullstellige Prädikatensymbole werden auch Aussagenvariablen genannt.
- $\mathcal{F}$ eine abzählbare Menge von *Funktionssymbolen* mit Stelligkeit: Sie werden mit $f, g, h, \ldots$ bezeichnet. Ist ein Funktionssymbol f von der Stelligkeit n, schreiben wir auch f/n. Nullstellige Funktionssymbole werden auch Konstanten genannt.
- $\mathcal{V}$ eine abzählbar unendliche Menge von *Variablen*: Sie werden mit $X, Y, Z, \ldots$ bezeichnet.
- eine Menge von *logischen Junktoren*: $\top$ und $\bot$ (Nullstellige Verknüpfungen),$\neg$,$\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
- die *Quantoren* $\forall$ (Allquantor) und $\exists$ (Existenzquantor).
- Klammern und Kommata: „(“, „)“, „,“.

Die Mengen $\mathcal{P}$ und $\mathcal{F}$ bilden die *Signatur* $\Sigma = (\mathcal{P}, \mathcal{F})$.

Wir verwenden hier eine an der Logikprogrammierung orientierte Syntax: Prädikaten- und Funktionssymbole beginnen mit kleinem Anfangsbuchstaben und Variablen mit großem. Funktionssymbole werden auch Funktoren genannt. Die Abkürzung $\bar{x}$ steht für eine Sequenz von Variable

Im folgenden sei eine Signatur Σ und eine Menge von Variablen $\mathcal{V}$ gegeben. *Ausdrücke* in Sprachen erster Stufe sind Terme und Formeln.

Definition A.2 Die Menge $\mathcal{T}(\Sigma, \mathcal{V})$ der *Terme* ist die kleinste Menge mit:

- Ist $X \in \mathcal{V}$, dann ist $X \in \mathcal{T}(\Sigma, \mathcal{V})$.

- Sind $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ und f ein n-stelliges Funktionssymbol, dann ist auch $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$. Die Terme $t_1, \ldots, t_n$ heißen *Argumente* von f.

Im Fall eines nullstelligen Funktionssymbols schreibt man f statt $f()$.

Definition A.3 Die Menge $\mathcal{F}(\Sigma, \mathcal{V})$ der *Formeln* ist die kleinste Menge mit:

- $\top, \bot \in \mathcal{F}(\Sigma, \mathcal{V})$.
- Ist $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ und p ein n-stelliges Prädikatensymbol, dann ist auch $p(t_1, \ldots, t_n) \in \mathcal{F}(\Sigma, \mathcal{V})$. Die Terme $t_1, \ldots, t_n$ heißen *Argumente* von p.
- Sind $F, G \in \mathcal{F}(\Sigma, \mathcal{V})$, dann sind auch $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G) \in \mathcal{F}(\Sigma, \mathcal{V})$.
- Ist $X \in \mathcal{V}$ und $F \in \mathcal{F}(\Sigma, \mathcal{V})$, dann sind auch $(\forall X\ F)$, $(\exists X\ F) \in \mathcal{F}(\Sigma, \mathcal{V})$.

Die Formel $p(t_1, \ldots, t_n)$ nennt man *atomare Formel* (*Atom*), alle anderen *zusammengesetzte Formeln*. Im Fall eines nullstelligen Prädikatensymbols schreibt man p statt $p()$.

Wie üblich sparen wir Klammern durch Präzedenz- und Assoziativitätsregeln: Quantoren und Negation binden enger als die zweistelligen logischen Verknüpfungen. Unter diesen bindet die Konjunktion am engsten, danach folgt die Disjunktion und dann die Implikation und am Ende die Äquivalenz.

In den folgenden Definitionen dieses Kapitels werden der Einfachheit halber zusammengesetzte Formeln durch Formeln dargestellt, die lediglich aus den logischen Junktoren $\bot$, $\rightarrow$ und dem Quantor $\forall$ gebildet werden, wobei wir die übrigen Junktoren wie folgt darauf zurückführen:

- $\neg F := F \rightarrow \bot$
- $\exists X F := \neg \forall X \neg F$
- $\top := \neg \bot$
- $F \leftrightarrow G := (G \rightarrow F) \wedge (F \rightarrow G)$
- $F \vee G := \neg F \rightarrow G$
- $F \wedge G := \neg(F \rightarrow \neg G)$

Definition A.4 Wir definieren die Mengen $\text{frei}(t)$ bzw. $\text{frei}(F)$ der *freien Variablen* eines Terms t bzw. einer Formel F induktiv nach dem Aufbau von t bzw. F:

- $\text{frei}(X) := \{X\}$ für eine Variable X.
- $\text{frei}(f(t_1, \ldots, t_n)) := \text{frei}(t_1) \cup \ldots \cup \text{frei}(t_n)$ für ein n-stelliges Funktionssymbol f und Terme $t_1, \ldots, t_n$.
- $\text{frei}(p(t_1, \ldots, t_n)) := \text{frei}(t_1) \cup \ldots \cup \text{frei}(t_n)$ für ein n-stelliges Prädikatensymbol p und Terme $t_1, \ldots, t_n$.
- $\text{frei}(F \rightarrow G) := \text{frei}(F) \cup \text{frei}(G)$ für Formeln F und G.
- $\text{frei}(\forall X\ F) := \text{frei}(F) \setminus \{X\}$ für eine Formel F und eine Variable X.
- $\text{frei}(\bot) := \emptyset$.

Formeln ohne freie Variablen werden als *geschlossene Formeln* oder *Sätze* bezeichnet. Eine *Theorie* ist eine Menge von Sätzen. Terme ohne freie Variablen werden als *variablenfreie Terme* oder *Grundterme* bezeichnet.

Definition A.5 Sei G eine Formel und $\{X_1, \ldots, X_n\}$ die Menge der freien Variablen von G. Der *universelle Abschluß* $\forall\, G$ und der *existentielle Abschluß* $\exists\, G$ der Formel G ist wie folgt definiert:

$$\begin{aligned} \forall G &= \forall X_1 \ldots \forall X_n\ G \\ \exists G &= \exists X_1 \ldots \exists X_n\ G \end{aligned}$$

A.1.2 Prädikatenlogische Formel in Klauselform

Beweisverfahren in der Logik werden oft auf Formeln in einer Normalform angewendet. Der in Abschnitt A.2 vorgestellte LP-Kalkül ist nur auf Formeln einer speziellen syntaktischen Form anwendbar, der sogenannten *Klauselform*, die durch folgende Definitionen eingeführt werden soll:

Definition A.6 Ein *Literal* ist eine atomare Formel F (*positives Literal*) oder eine negierte atomare Formel $\neg F$ (*negatives Literal*).

Definition A.7 Eine Klausel ist eine Formel der Form

$$\forall X_1, \ldots \forall X_m (L_1 \vee \ldots \vee L_n),$$

in der die L_i für Literale und die $X_1, \ldots, X_m$ für all die Variablen stehen, die in $L_1, \ldots, L_n$ vorkommen..

Klauseln werden in einer speziellen Klauselnotation geschrieben:

$$\forall X_1, \ldots \forall X_m (A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_k).$$

Diese Notation wird oft durch

$$A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_k$$

wiedergegeben, wobei $A_1, \ldots, A_n, B_1, \ldots, B_k$ Atome sind und $X_1, \ldots, X_m$ all jene Variablen, die in diesen Atomen vorkommen.

A.1.3 Substitutionen

Im folgenden betrachten wir die Ersetzung von Variablen durch Terme.

Definition A.8 Eine *Substitution* ist eine Abbildung $\theta : \mathcal{V} \longrightarrow \mathcal{T}(\Sigma)$ von der Menge der Variablen in die Menge der Terme , die nur an endlich vielen Stellen von der Identität abweicht. Wir schreiben θ als eine endliche Menge von Paaren wie folgt: $\theta = \{X_1 \hookleftarrow t_1, \ldots, X_n \hookleftarrow t_n\}$, wobei $X_i \in \mathcal{V}$, $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$. Paare der Form $X \hookleftarrow X$ werden weggelassen. Die Identität (leere Substitution) wird mit ε bezeichnet.
Eine Substitution θ kann wie folgt auf kanonische Weise auf Terme und quantorenfreie Formeln fortgesetzt werden:

- $f(t_1, \ldots, t_n)\theta := f(t_1\theta, \ldots, t_n\theta)$ für $f \in \mathcal{F}$;
- $p(t_1, \ldots, t_n)\theta := p(t_1\theta, \ldots, t_n\theta)$ für $p \in \mathcal{P}$;
- $\perp\theta := \perp$;
- $(F \rightarrow G)\theta := F\theta \rightarrow G\theta$

$\theta\beta$ bezeichnet die Komposition der Substitutionen θ und β. Im Gegensatz zur üblichen Schreibweise von Abbildungen werden Substitutionen in Postfix-Notation geschrieben.

Definition A.9 Eine Formel (Ein Term) F ist eine *Instanz* einer Formel (eines Termes) G, falls es eine Substitution θ gibt mit $F = G\theta$. F ist *Variante* von G, falls F Instanz von G und G Instanz von F ist.
Eine *frische Variante* einer Formel relativ zu einer Formelmenge S ist eine Variante einer Formel, die keine gemeinsamen Variablen mit allen Formel aus S hat.

Eine besondere Rolle spielt die Unifikation. Ihre Aufgabe ist, Atome oder Terme einander anzugleichen.

Definition A.10 Eine Substitution θ heißt *Unifikator* für eine Menge $S = \{A_1, A_2, \ldots, A_n\}$, falls $A_1\theta = A_2\theta = \ldots = A_n\theta$. Gibt es für S einen solchen Unifikator, dann heißt S *unifizierbar*. Ein Unifikator θ heißt *allgemeinster Unifikator* für S, falls es für jeden Unifikator σ von S eine Substitution β gibt, so daß $\sigma = \theta\beta$.

Robinson bewies in [Rob65], daß es unter den Unifikatoren von zwei unifizierbaren Objekten immer einen allgemeinsten gibt und daß dieser bis auf die Umbenennung von Variablen (Variantenbildung) eindeutig ist. Außerdem gab er einen Algorithmus zur Berechnung eines allgemeinsten Unifikators an.

A.1.4 Semantik

Nachdem die Syntax einer Sprache der Prädikatenlogik definiert ist, muß die Semantik einer Formel geklärt werden, nämlich was eine Formel bedeutet. Dazu wird im folgenden der Begriff der Interpretation eingeführt. Eine Interpretation ordnet den nullstelligen Funktionssymbolen der Sprache Objekte zu, den mehrstelligen Abbildungen zwischen Objekten und den Prädikatensymbolen Relationen über Objekten.

Definition A.11 Eine *Interpretation* $\mathcal{I}$ ist ein Paar $(D,\mathcal{I}[.])$ mit:

- D ist eine nichtleere Menge („Universum" oder „Definitionsbereich").
- $\mathcal{I}[f] : D^n \to D$ für jedes n-stellige Funktionssymbol f.
- $\mathcal{I}[p] \subseteq D^n$, für jedes n-stellige Prädikatensymbol p.

Definition A.12 Eine *Variablenbelegung* für die Variablenmenge $\mathcal{V}$ bzgl. eines Universums D ist eine Abbildung $\alpha : \mathcal{V} \to D$.

Definition A.13 Wenn $\mathcal{I}$ eine Interpretation und α eine Variablenbelegung ist, bezeichnet $\mathcal{I}_\alpha$ folgende Interpretation von Termen und Formeln.

Die Funktion $\mathcal{I}_\alpha[t]$ zur Interpretation von Termen ist definiert durch:

- $\mathcal{I}_\alpha[X] = \alpha(X)$ für $X \in \mathcal{V}$,
- $\mathcal{I}_\alpha[f(t_1, \ldots, t_n)] = \mathcal{I}[f](\mathcal{I}_\alpha[t_1], \ldots, \mathcal{I}_\alpha[t_n])$ für ein n-stelliges Funktionssymbol f und Terme $t_1, \ldots, t_n$.

Die Relation $\mathcal{I}_\alpha \models F$ („$\mathcal{I}_\alpha$ erfüllt F") zur Interpretation von Formeln ist definiert durch:

- $\mathcal{I}_\alpha \not\models \perp$, d.h. $\mathcal{I}_\alpha \models \perp$ gilt nicht;
- $\mathcal{I}_\alpha \models p(t_1, \ldots, t_n)$ gdw. $(\mathcal{I}_\alpha[t_1], \ldots, \mathcal{I}_\alpha[t_n]) \ \in \mathcal{I}_\alpha[p]$;
- $\mathcal{I}_\alpha \models F \to G$ gdw. $\mathcal{I}_\alpha \not\models F$ oder $\mathcal{I}_\alpha \models G$;
- $\mathcal{I}_\alpha \models \forall X F$ gdw. für alle $u \in D$ gilt $\mathcal{I}_{\alpha[X \hookleftarrow u]} \models F$.

Die Abkürzung „gdw." steht für „genau dann, wenn".

Zu einer Variablenbelegung α bezeichnet $\alpha[Y \hookleftarrow a]$ die modifizierte Variablenbelegung

$$\alpha[Y \hookleftarrow a](X) = \begin{cases} a, \text{ falls } X = Y \\ \alpha(X), \text{ sonst} \end{cases}$$

Die Variablenbelegung $\alpha[Y \hookleftarrow a]$ ist also gleich der Variablenbelegung α bis auf die Belegung von Y, die durch $[Y \hookleftarrow a]$ als a definiert wird.

Ob eine Formel erfüllt wird, hängt von der Belegung ihrer freien Variablen ab. Da aber für unsere Zwecke Sätze reichen, werden im folgenden der Modellbegriff und der Folgerungsbegriff nur für Sätze definiert.

Definition A.14 $\mathcal{I}$ ist ein *Modell* des Satzes F, geschrieben $\mathcal{I} \models F$, falls $\mathcal{I}_\alpha \models F$ für alle Variablenbelegungen α gilt.
Ist M eine Menge von Sätzen, so heißt $\mathcal{I}$ *Modell* von M, falls $\mathcal{I} \models F$ für alle $F \in M$.

Die folgenden Definitionen gelten analog auch für Mengen von Sätzen.

Definition A.15 Ein Satz F ist *erfüllbar* (konsistent), falls er ein Modell hat.
Ein Satz ist *unerfüllbar*, falls er nicht erfüllbar ist.
Ein Satz F ist eine *Tautologie* (*allgemeingültig*), falls $\mathcal{I} \models F$ für alle Interpretationen $\mathcal{I}$.

Definition A.16 Aus einem Satz F *folgt (logisch)* ein Satz G, geschrieben $F \models G$, falls jedes Modell von F auch Modell von G ist.
Zwei Sätze F und G sind *logisch äquivalent*, falls $F \models G$ und $G \models F$.

A.1.5 Herbrand-Semantik

Besondere Modelle sind die *Herbrand-Modelle*, bei denen das Universum der Interpretationen die Menge alle Grundterme ist. Solche Herbrandmodelle existieren nur dann, wenn es überhaupt Grundterme gibt. Deswegen trifft man die Verabredung, daß es wenigstens eine Konstante in der Sprache gibt.

Definition A.17 Das *Herbrand-Universum* H_Σ ist die Menge aller variablenfreien Terme.
Eine *Herbrand-Interpretation* $\mathcal{H} = (H_\Sigma, \mathcal{I}[.])$ ist eine Interpretation, die jedem n-stelligen Funktionssymbol f eine Funktion $\mathcal{I}[f] : H_\Sigma^n \to H_\Sigma$ zuordnet, die die Terme $t_1, \ldots, t_n \in H_\Sigma$ auf den Term $f(t_1, \ldots, t_n) \in H_\Sigma$ abbildet. Die Interpretation der Prädikatensymbole unterliegt keiner Einschränkung.

Eine Herbrand-Interpretation ist also eine Interpretation über dem Herbrand-Universum, die jedem variablenfreien Term den Term selbst zuordnet. Herbrandmodelle sind gewissermaßen kanonisch, man kann ihre Elemente explizit konstruieren.

Das Prinzip der Herbrand-Semantik geht zurück auf ein berühmtes Theorem von Leopold Löwenheim und Thoralf Skolem. Die Idee ist, ein festes Universum zu bestimmen, so daß es ausreicht, nur Interpretationen über diesem Universum zu betrachten. Das Ziel ist dann, zu jeder Formel und jeder beliebigen Interpretation eine Herbrand-Interpretation zu geben, die der Formel dieselbe Bedeutung zuordnet. Damit würde es ausreichen, für $F \models G$ nur Herbrand-Interpretationen zu betrachten. Dies gilt nicht für beliebige Formel, aber es gilt für Formeln in Klauselform.

Jede prädikatenlogische Formel F läßt sich in eine prädikatenlogische Formel F' überführen, für die gilt:

1. F ist erfüllbar gdw. F' ist erfüllbar.
2. F' ist in Klauselform:

$$F' := \forall X_1, \ldots, \forall X_m (C_1 \wedge \ldots \wedge C_n),$$

   wobei jedes C_i eine Klausel ist.

Beim Beweis für die Unerfüllbarkeit von einer Formel F in Klauselform reicht es aus, wenn man die Unerfüllbarkeit von F in dem Herbrand-Universum feststellt.

A.1.6 Kalküle

Ein Kalkül erweitert die Sprache der Prädikatenlogik um das syntaktische Gegenstück zur logischen Folgerung, nämlich um den Ableitungsbegriff. Ein Kalkül definiert also syntaktische *Ableitungen* als Operationen auf den Formeln. Mittels eines Kalküls kann man auch die operationale

Semantik einer Programmiersprache beschreiben, d.h. formal angeben, wie Berechnungen in der Programmiersprache erfolgen.

Im allgemeinen besteht ein Kalkül aus einer Menge von logischen Axiomen und einer Menge von Schlußregeln, mit denen aus Formeln weitere Formeln abgeleitet werden können. In dieser Arbeit werden wir einen Kalkül durch ein Zustandsübergangssystem beschreiben, das die Abarbeitung eines Programms durch Zustandsübergänge beschreibt, und eine Kongruenzrelation, die triviale Gleichheiten von Zuständen beschreibt.

Definition A.18 Ein *Zustandsübergangssystem* ist ein Tripel $(\mathcal{Z}, \mathcal{T}, \mapsto)$, wobei $\mathcal{Z}$ eine Menge von Zuständen ist, $\mathcal{T} \subseteq \mathcal{Z}$ die Menge der terminalen Zustände (Anfangs- und Endzustände) und $\mapsto \subseteq \mathcal{Z} \times \mathcal{Z}$ die Übergangsrelation, für die gilt:

$$S \not\mapsto S' \text{ für alle Endzustände } S \in \mathcal{T} \text{ und alle } S' \in \mathcal{Z},$$

d.h. von Endzuständen aus ist kein Zustandsübergang möglich.

Eine *Reduktion* (ein Zustandsübergang, ein Ableitungsschritt) von einem *(Ausgangs-)Zustand* S zu einem *(Folge-)Zustand* S' kann erfolgen, wenn bestimmte *(Reduktions-)Bedingungen* erfüllt sind. Wir schreiben dies als *Reduktionsregel (Ableitungsregel, Inferenzregel, Schlußregel)* von der Form:

$$\frac{\begin{array}{c}\text{Bedingung } 1\\ \dots\\ \text{Bedingung } n\end{array}}{S \mapsto S'}$$

Definition A.19 Eine *Berechnung* ist eine Sequenz $S_0 \mapsto S_1 \mapsto S_2 \mapsto \dots$ von Reduktionen. Wir schreiben $S \mapsto^n T$, falls es $S_0, \dots, S_n$ gibt mit $S = S_0$, $T = S_n$ und $S_i \mapsto S_{i+1}$ $(i = 0, \dots, n-1)$. Wir schreiben $S \mapsto^* T$, wenn es ein $n \geq 0$ gibt mit $S \mapsto^n T$.
Eine *Ableitung* ist eine Berechnung, die entweder in einem Endzustand endet oder unendlich ist.

Definition A.20 Eine *Äquivalenzrelation* $\cong$ ist eine zweistellige Relation, die reflexiv, symmetrisch und transitiv ist:

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

Definition A.21 Eine *Kongruenz* $\cong$ *über Formeln* ist eine Äquivalenzrelation, für die gilt:
Wenn $\varphi \cong \psi$ dann auch $\neg\varphi \cong \neg\psi$ und wenn $\varphi \cong \psi$ und $\varphi' \cong \psi'$ dann auch $\varphi \rightarrow \varphi' \cong \psi \rightarrow \psi'$.

Die logische Äquivalenz nach Definition A.16 ist eine Kongruenz. 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.

In dieser Arbeit wird die in Abbildung A.1 definierte Kongruenz verwendet. Die Kongruenz $\cong$ 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 $\perp$ alle anderen Konjunkte überflüssig macht.

Definition A.22 Ein *Kalkül* ist ein Quadrupel $(\mathcal{Z}, \mathcal{T} \cong, \mapsto)$ bestehend aus einer Menge $\mathcal{Z}$ von Zuständen, einer Menge von Terminalen Zuständen $\mathcal{T}$, einer Kongruenz $\cong$ auf $\mathcal{Z}$ und einer Übergangsrelation $\mapsto$ auf $\mathcal{Z}$.

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

Abbildung A.1: Kongruenz

A.2 Logikprogrammierung

Wie wir im letzten Abschnitt gesehen haben, beschäftigt sich die Logik mit der Bedeutung vorgegebener Repräsentation. Daneben spielt jedoch auch die Frage, ob ein Kalkül effizient genutzt werden kann, um ein Fragment der Prädikatenlogik ausführen zu können, eine entscheidende Rolle. In diesem Abschnitt wird ein Kalkül vorgestellt, der auf dem Resolutionsprinzip basiert. Dieser Kalkül (LP-Kalkül) ist speziell für ein effizientes Arbeiten in der Hornklauseluntermenge der Klausellogik ausgelegt und beschreibt die Syntax und operationale Semantik der meisten LP-Sprachen.

A.2.1 LP-Kalkül

Die Signatur des LP-Kalküls enthält beliebige Funktions- und Prädikatensymbole. Die Syntax ist den folgenden Definitionen und Abb. A.2 zu entnehmen. Danach geben wir das zugehörige Zustandsübergangssystem an.

Atome:	A, B	$::=$	$p(t_1, \ldots, t_n),\ n \geq 0$
Ziele:	G, H	$::=$	$\top \mid \bot \mid A \mid G \wedge H$
LP-Klauseln:	K	$::=$	$A \leftarrow G$
LP-Programme:	P	$::=$	$\{K_1, \ldots, K_m\},\ m \geq 0$

Abbildung A.2: LP-Syntax

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.

Definition A.23 Ein *Ziel* ist entweder $\top$ oder $\bot$ oder ein Atom oder eine Konjunktion von Zielen. $\top$ wird auch *leeres Ziel* genannt.

Definition A.24 Eine *(LP-)Klausel (definite Klausel)* hat die Form $A \leftarrow G$, wobei wir das Atom A als *Kopf* und das Ziel G als *Rumpf* der Klausel bezeichnen. Klauseln von der Form $A \leftarrow \top$ werden *Fakten* genannt, alle anderen *Regeln*.
Ein *LP-Programm* ist eine endliche Menge von LP-Klauseln.

Im LP-Kalkül sind Zustände Paare aus einem Ziel und einer Substitution. Man unterscheidet zwischen erfolgreichen und erfolglosen Endzuständen, Ableitungen bzw. Zielen.

Definition A.25 Ein *Zustand* ist ein Paar $\langle G, \theta \rangle$, wobei G ein Ziel ist und θ eine Substitution.
Ein *Anfangszustand* ist ein Zustand von der Form $\langle G, \varepsilon \rangle$, wobei ε die Identitätssubstitution ist.
Ein Zustand heißt *erfolgreicher Endzustand*, falls er von der Form $\langle \top, \theta \rangle$ ist. Er heißt *erfolgloser Endzustand (gescheitert)*, falls er von der Form $\langle \bot, \varepsilon \rangle$ ist.

Intuitiv repräsentiert G die Ziele, die noch zu lösen sind, und θ die bisher berechneten Substitutionen.

Definition A.26 Eine Ableitung ist *erfolgreich*, wenn ihr Endzustand erfolgreich ist. Eine Ableitung ist *erfolglos* (*gescheitert*), wenn ihr Endzustand erfolglos (gescheitert) ist.
Ein Ziel G ist *erfolgreich*, wenn es eine erfolgreiche Ableitung beginnend mit $\langle G, \varepsilon \rangle$ hat. Ein Ziel G ist *erfolglos (endlich gescheitert)*, wenn es nur erfolglose Ableitungen beginnend mit $\langle G, \varepsilon \rangle$ hat.

Definition A.27 Eine Substitution θ wird *(berechnete) Antwort eines Zieles* G genannt, falls es eine erfolgreiche Ableitung $\langle G, \varepsilon \rangle \mapsto^* \langle \top, \theta \rangle$ gibt.

In der Reduktionsregel **Unfold** (Abb. A.3) wird unter Anwendung der Kongruenz ein beliebiges Atom A aus dem Ziel G gewählt. Aus dem Programm P wird eine (frische) Variante einer beliebigen Klausel $(B \leftarrow H)$ gewählt, so daß diese Variante keine gemeinsamen Variablen mit dem Programm P und dem aktuellen Zustand hat und so daß A unter der Substitution θ mit dem Kopf der Klausel B mit dem allgemeinsten Unifikator β unifizierbar ist. Durch die Variantenbildung (Variablenumbennung) wird vermieden, daß Variablen im Programm P gebunden werden und damit das Programm modifiziert wird und die Klausel nicht in vollem Umfang wiederverwendbar ist. Im neuen Zustand wird A durch den Rumpf der Klausel, H, ersetzt und die neue Substitution β auf die bereits vorhandene Substitution θ angewendet. Die Reduktionsregel **Clash** deckt den

Unfold

$$\frac{\begin{array}{c}(B \leftarrow H) \text{ eine frische Variante aus } P\\ \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\theta\end{array}}{\langle A \wedge G, \theta \rangle \mapsto \langle H \wedge G, \theta\beta \rangle}$$

Clash

$$\frac{\begin{array}{c}\text{Es gibt keine Klausel } (B \leftarrow H) \text{ aus } P,\\ \text{so daß ein Unifikator von } B \text{ und } A\theta \text{ existiert}\end{array}}{\langle A \wedge G, \theta \rangle \mapsto \langle \bot, \varepsilon \rangle}$$

Abbildung A.3: LP-Reduktionsregeln

Fall ab, daß für ein Atom A keine Klausel gefunden werden kann, mit der die Reduktionsregel **Unfold** möglich wäre.

In diesem einfachen LP-Kalkül ist die Auswahl der Klausel innerhalb eines Programms und die Auswahl des Atoms aus dem Ziel nicht vorbestimmt (Nichtdeterminismus). In der LP-Sprache Prolog wird immer das am weitesten links stehende Atom eines Zieles ausgewählt und ganz entfaltet. Dabei werden die Klauseln in textueller Reihenfolge ausgewählt: Scheitert die Ableitung mit einer bestimmten Klausel, versucht man, durch ein systematisches Suchverfahren, nämlich *Rücksetzen*, aus der Sackgasse zu kommen: Es wird eine neue Ableitung für das Atom mit Hilfe der nächsten Klausel versucht. Man versucht immer, das zuletzt gewählte Atom, bei dem noch Klauseln zur Auswahl stehen, erneut zu entfalten (chronologisches Rücksetzen). Es wird so versucht, nur einen möglichst kleinen Teil der Gesamtableitung neu zu berechnen. Auf diese Weise werden erfolgreiche Ableitungen gesucht. Die Auswahl der Klauseln bei der Suche durch Rücksetzen wird als „don't know"-Nichtdeterminismus bezeichnet.

A.2.2 Korrektheit und Vollständigkeit des LP-Kalküls

Eine LP-Klausel $A \leftarrow G$ kann als der universelle Abschluß der Implikation $G \rightarrow A$ gelesen werden. Die deklarative Semantik eines LP-Programms P ist dann die Konjunktion der logischen Leseweisen aller Klauseln von P (in Zeichen $P^{\rightarrow}$).

Im folgenden geben wir die Sätze an, die die Übereinstimmung zwischen operationaler und deklarativer Semantik ausdrücken. Der folgende Satz besagt, daß alles, was ableitbar ist, auch logisch aus dem Programm folgt (Korrektheit).

Satz A.28 Sei P ein LP-Programm, G ein Ziel. Wenn θ eine berechnete Antwort von G ist, dann gilt

$$\mathcal{P} \models \forall G\theta$$

Der folgende Satz besagt, daß eine Verallgemeinerung von allem, was logisch folgt, auch ableitbar ist (Vollständigkeit).

Satz A.29 Sei P ein LP-Programm, G ein Ziel und θ eine Substitution. Wenn $\mathcal{P} \models \forall G\theta$ gilt, dann gibt es eine berechnete Antwort σ von G, so daß $\theta = \sigma\beta$ ist.

Literaturverzeichnis

[AAB$^+$93] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. S. und J. Widén. SICStus Prolog User's Manual. Technical Report T93-01, Swedish Institute of Computer Science, 1993.

[Abd96] S. Abdennadher. Von der Logikprogrammierung zu Constraint Handling Rules. Arbeitspapier PMS-AP-1996-1, Institut für Informatik, Ludwig-Maximilians-Universität München, 1996.

[Abd97] S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP'97*, LNCS 1330. Springer-Verlag, November 1997.

[ABEG95] S. Abdennadher, F. Bry, N. Eisinger und T. Geisler. The Theorem Prover Satchmo: Strategies, Heuristics, and Applications (System Description). In *IVèmes Journées Francophones de Programmation en Logique*, Dijon, 1995.

[ABF96] S. Abdennadher, P. Blenninger und T. Frühwirth. Rent Estimates with Constraints over the Internet. In *Vèmes Journées Francophones de Programmation en Logique et programmation par Contraintes*, Clermont Ferrand, 1996.

[ACD$^+$94] A. Aggoun, D. Chan, P. Dufrense, E. Falvey, H. Grant, A. Herold, G. MacCartney, M. Maier, D. Miller, B. Perez, E. van Rossum, J. Schimpf, P. Tsahageas und D. de Villeneuve. *ECL^iPS^e 3.4 User Manual.* ECRC Munich, Germany, Juli 1994.

[AFM96a] S. Abdennadher, T. Frühwirth und H. Meuss. Implémentation de Constraint Solvers: Théorie et Pratique. In *Forum de la Recherche en Informatique*, Tunis, Tunisia, Juli 1996.

[AFM96b] S. Abdennadher, T. Frühwirth und H. Meuss. On Confluence of Constraint Handling Rules. In *2nd International Conference on Principles and Practice of Constraint Programming, CP'96*, LNCS 1118. Springer-Verlag, August 1996.

[AFM98] S. Abdennadher, T. Frühwirth und H. Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints Journal*, Kluwer Academic Publishers. Angenommen zur Veröffentlichung, 1998.

[AFMM95] S. Abdennadher, T. Frühwirth, M. Marte und H. Meuss. A Confluence Test for Concurrent Constraint Programs. Forschungsbericht PMS-FB-1995-4, Institut für Informatik, Ludwig–Maximilians–Universität München, Oktober 1995.

[AKP94] H. Aït-Kaci und A. Podelski. Functions as Passive Constraints in LIFE. *ACM Transactions on Programming Languages and Systems*, 16(4), Juli 1994.

[AM97] S. Abdennadher und M. Marte. University Timetabling using Constraint Handling Rules. In *VIèmes Journées Francophones de Programmation en Logique*, Nantes, 1997.

[AS97a] S. Abdennadher und H. Schlenker. INTERDIP – Ein Interaktiver Constraint-basierter Dienstplaner für Krankenstationen. In F. Bry, B. Freitag und D. Seipel, Hrsg., *12th Workshop on Logic Programming WLP'97*, September 1997.

[AS97b] S. Abdennadher und H. Schütz. Model Generation with Existentially Quantified Variables and Constraints. In *6th International Conference on Algebraic and Logic Programming*, LNCS 1298. Springer-Verlag, 1997.

[BD86] L. Bachmair und N. Dershowitz. Commutation, Transformation, and Termination. In J. H. Siekmann, Hrsg., *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, LNCS 230. Springer-Verlag, Juli 1986.

[BH92] M. P. Bonacina und J. Hsiang. On Rewrite Programs: Semantics and Relationsship with PROLOG. *Journal of Logic Programming*, 14:155–180, 1992.

[BT97] F. Bry und S. Torge. Model Generation for Applications – A Tableaux Method Complete for Finite Satisfiability. Forschungsbericht PMS-FB-1997-15, Institut für Informatik, Ludwig–Maximilians–Universität München, 1997.

[BY96] F. Bry und A. Yahya. Minimal Model Generation with Positive Unit Hyper-Resolution Tableaux. In *5th Workshop on Theorem Proving with Tableaux and Related Methods*, Springer LNAI, 1996.

[CD93] P. Codognet und D. Diaz. Boolean Constraint Solving Using clp(FD). In D. Miller, Hrsg., *Logic Programming - Proceedings of the 1993 International Symposium*, Vancouver, Canada, 1993. The MIT Press.

[CFMW97] M. Codish, M. Falaschi, K. Marriott und W. Winsborough. A Confluent Semantic Basis for the Analysis of Concurrent Constraint Logic Programs. *Journal of Logic Programming*, 30(1), 1997.

[CG81] K. L. Clark und S. Gregory. A Relational Language for Parallel Programming. In *Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, New York, 1981.

[CKvC83] A. Colmerauer, H. Kanoui und M. van Caneghem. PROLOG, Theoretical Bases And Current Developments. *TECH. AND SCI. INF. (FRANCE)*, 2(4), Juli - August 1983.

[Col90] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7), Juli 1990.

[dBGMP94] F.S. de Boer, M. Gabbrielli, E. Marchiori und C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Conference Record of the 21st ACM Symposium on Principles of Programming Languages*, 1994.

[Der82] N. Dershowitz. Ordering for term rewriting systems. *Theoretical Computer Science*, 17, 1982.

[Der93] N. Dershowitz. A Taste of Rewrite Systems. In P. E. Lauer, Hrsg., *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Springer-Verlag, 1993.

[DHS+90] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun und A. Herold. The CHIP System: Constraint Handling in Prolog. In M. Stickel, Hrsg., *Proceedings of the Tenth International Conference on Automated Deduction*, LNCS 449. Springer-Verlag, 1990.

[DJ84] N. Dershowitz und N. A. Josephson. Logic Programming by Completion. In Sten-Åke Tärnlund, Hrsg., *Proceedings of the Second International Conference on Logic Programming*, Seiten 313–320, Uppsala, 1984.

[DOS88] N. Dershowitz, N. Okada und G. Sivakumar. Confluence of Conditional Rewrite Systems. In J.-P. Jouannaud und S. Kaplan, Hrsg., *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems*, LNCS 308, 1988.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf und F. Berthier. The Constraint Logic Programming Language CHIP. Technical Report TR-LP-37, ECRC, Munich, Germany, Mai 1988.

[ENP92] N. Eisinger, A. Nonnengart und A. Präcklein. Termersetzungssysteme. In K.H. Bläsius und H.-J. Bürckert, Hrsg., *Deduktionssysteme*, Kapitel 3.4. Oldenbourg Verlag, 1992.

[FA96] T. Frühwirth und S. Abdennadher. The Munich Rent Advisor. In *Proceedings 1st Workshop on Logic Programming Tools for Internet Applications*, Bonn, 1996.

[FA97a] T. Frühwirth und S. Abdennadher. Anwendungen Constraintbasierter Programmierung. In *GI-Jahrestagung, Aachen*. Springer-Verlag, 1997.

[FA97b] T. Frühwirth und S. Abdennadher. *Constraint-Programmierung: Grundlagen und Anwendungen*. Springer-Verlag, September 1997.

[FA97c] T. Frühwirth und S. Abdennadher. Der Mietspiegel im Internet: Ein Fall für Constraint-Logikprogrammierung. *KI 1/97, Themenheft Constraints*, April 1997.

[FB95] T. Frühwirth und P. Brisset. High-Level Implementations of Constraint Handling Rules. Technical report, ECRC, 1995.

[FGMP95] M. Falaschi, M. Gabbrielli, K. Marriott und C. Palamidessi. Confluence in Concurrent Constraint Programming. In V.S. Alagar und M.Nivat, Hrsg., *Proceedings of AMAST '95*, LNCS 936. Springer-Verlag, 1995.

[FHK+92] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy und M. Wallace. Constraint Logic Programming: An Informal Introduction. In G. Comyn, N.E. Fuchs und M.J. Ratcliffe, Hrsg., *Logic Programming in Action*, LNCS 636. Springer-Verlag, 1992.

[Frü95] T. Frühwirth. Constraint Handling Rules. In A. Podelski, Hrsg., *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, März 1995.

[Frü97] T. Frühwirth. *A Declarative Language for Constraint Systems: Theory and Practice of Constraint Handling Rules*. Habilitationsschrift, in Arbeit, 1997.

[HD83] J. Hsiang und N. Dershowitz. Rewrite Methods for Clausal and Non-Clausal Theorem Proving. In *Proceedings of the Tenth International Colloquium on Automata, Languages and Programming (Barcelona, Spain)*, LNCS 154. Springer-Verlag, Juli 1983.

[HJM+87] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey und R. Yap. *The CLP(R) Programmer's Manual*. Monash University, Clayton 3168, Victoria, Australia, 1987.

[HJW78] H. D. Ebbinghaus, J. Flum und W. Thomas. *Einführung in die mathematische Logik*. Wissenschaftliche Buchgesellschaft Darmstadt, 1978.

[Hol92] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe und M. Wirsing, Hrsg., *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming (Leuven, Belgium)*, LNCS 631. Springer-Verlag, 1992.

[HS88] M. Höhfeld und G. Smolka. Definite Relations over Constraint Languages. LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, Germany, Oktober 1988.

[Hue80] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term-Rewriting Systems. *Journal of the ACM*, 27(4), Oktober 1980.

[Hui90] S. Le Huitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart und J. Małuszyński, Hrsg., *Proceedings of Programming Language Implementation and Logic Programming*, LNCS 456. Springer-Verlag, August 1990.

[JL86] J. Jaffar und J.L. Lassez. Constraint Logic Programming. Technical Report 86/74, Monash University, Victoria, Australia, Juni 1986.

[JL87] J. Jaffar und J. Lassez. Constraint logic programming. In Michael J. O'Donnell, Hrsg., *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, Januar 1987.

[JM94] J. Jaffar und M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 20, 1994.

[Kap84] S. Kaplan. Conditional Rewrite Rules. In *Theoretical Computer Science*, Band 33, 1984.

[KB70] D. E. Knuth und P. B. Bendix. *Simple Word Problems in Universal Algebra*. Pergamon Press, 1970.

[KK71] R. Kowalski und D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2, 1971.

[KM86] D. Kapur und D. R. Musser. Inductive Reasoning with Incomplete Specifications. In *Symposion on Logic in Computer Science*, Washington, D.C., USA, Juni 1986. IEEE Computer Society Press.

[Kum92] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1), 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 1987.

[Mac92] A. Mackworth. Constraint Satisfaction. In Stuart C. Shapiro, Hrsg., *Encyclopedia of Artificial Intelligence*. Wiley, 1992. Volume 1, second edition.

[Mah87] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In J.-L. Lassez, Hrsg., *Proceedings of the Fourth International Conference on Logic Programming*. The MIT Press, Mai 1987.

[Mah93] M. Maher. A Logic Programming View of CLP. In *Proceedings of the 10th International Conference on Logic Programming (ICLP)*, 1993.

[Mar96] M. Marte. Implementation eines Konfluenz-Tests für Constraint-Handling-Rules-Programme. Fortgeschrittenenpraktikum, Institut für Informatik, Ludwig-Maximilians-Universität München, 1996.

[MB88] R. Manthey und F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *9th Int. Conf. on Automated Deduction (CADE)*, LNCS 310, Argonne, IL, USA, may 1988. Springer-Verlag.

[Men87] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks, 1987.

[Meu96] H. Meuss. Konfluenz von Constraint-Handling-Rules-Programmen. Diplomarbeit, Institut für Informatik, Ludwig-Maximilians-Universität München, 1996.

[MO95] K. Marriott und M. Odersky. A Confluent Calculus for Concurrent Constraint Programming with Guarded Choice. In *1st International Conference on Principles and Practice of Constraint Programming, CP'95*. Springer-Verlag, September 1995.

[Neu90] U. Neumerkel. Extensible Unification by Metastructures. In *Proceedings of Metaprogramming in Logic (META'96)*, Leuven, Belgium, 1990.

[New42] M. H. A. Newman. On Theories With a Combinatorial Definition of Equivalence. In *Annals of Math*, Band 43, 1942.

[OB93] W. Older und F. Benhamou. Programming in CLP(BNR). In Paris Kanellakis, Jean-Louis Lassez und Vijay Saraswat, Hrsg., *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[Pla93] D. A. Plaisted. Equational Reasoning and Term Rewriting Systems. In D. Gabbay, C. Hogger, J. A. Robinson und J. Siekmann, Hrsg., *Handbook of Logic in Artificial Intelligence and Logic Programming*, Band 1, Kapitel 5. Oxford University Press, Oxford, 1993.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1), Januar 1965.

[Sar92] V.A. Saraswat. Concurrent Constraint Programming: A Brief Survey. Technical report, Xerox PARC, August 1992.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3), September 1989.

[Smo91] G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In *Digital Equipment Paris Research Laboratory Research Report*, France, Juni 1991.

[SRP91] V. A. Saraswat, M. Rinard und P. Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, Januar 1991.

[SS86] L. Sterling und E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Mass., 1986.

[SS93] G. Smolka und C. Schulte. Logische Programmierung. Skriptum zur Vorlesung, Fachbereich 14 - Informatik, Universität des Saarlandes, 1993.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1989.

[vH91] P. van Hentenryck. Constraint Logic Programming. *The Knowledge Engineering Review*, 6, 1991.

[vHSD92] P. van Hentenryck, H. Simonis und M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence*, 58(1-3), Dezember 1992.

[Wal96] M. Wallace. Practical Applications of Constraint Programming. *Constraints Journal*, 1(1,2), September 1996.

Abbildungsverzeichnis

Index

Lebenslauf

Persönliche Daten:	
Vor- und Familienname:	Slim Abdennadher
Geburtstag:	11. November 1967
Geburtsort:	Sfax, Tunesien
Familienstand:	verheiratet
Ausbildung	
Seit 1994	Promotion an der Ludwig-Maximilians-Universität in München
WS 87/88 bis SS 1993	Studium der Informatik an der Universität Kaiserslautern
Juli 1987	Feststellungsprüfung im Studienkolleg in Heidelberg
Juli 1986	Baccalauréat in Lyceé „9 Avril 1938“ in Sfax
Forschungsinteressen:	
	Constraint-Programmierung
	Deduktive Datenbaken und Logik-Programmierung
	Deduktion und Theorem-Beweiser
	Intelligente Anwendungen im Internet