INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstrasse 67 D–80538 München

Ludwig-
Maximilians-
Universität
München

Implementation eines Konfluenz-Tests für Constraint Handling Rules-Programme

Michael Marte

Fortgeschrittenenpraktikum

Beginn der Arbeit:	1. Januar 1996
Abgabe der Arbeit:	30. August 1996
Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Dipl.-Inform. Slim Abdennadher
	Dr. Thom Frühwirth

Inhaltsverzeichnis

Kapitel 1

Einleitung

Constraint Logic Programming (CLP) ist eine in den 80er Jahren entstandene Verbindung von Logikprogrammierung (LP) und Constraint Reasoning (CR).

Die zentrale Idee der LP ([Llo87]) ist die Trennung von Problembeschreibung und Problemlösung. Der Programmierer gibt nur noch die Problembeschreibung, die Problemlösung bleibt der Implementation der verwendeten Sprache überlassen. Die verbreiteste Sprache der LP ist Prolog.

CR ([FM94, Kum92]) beschäftigt sich mit der Lösung von Constraintproblemen. Ein Constraintproblem besteht aus einer Menge von Variablen und Relationen (Constraints) zwischen diesen Variablen. Die Aufgabe besteht darin, eine Belegung für die Variablen zu finden, sodaß alle Constraints erfüllt sind. Wertebereiche der Variablen und Art der Relationen bestimmen, welches Constraintsystem für die Lösung des Problems geeignet ist. Ein Modul, das effiziente Algorithmen für ein bestimmtes Problem in einem bestimmten Wertebereich implementiert, heißt Constraintsolver.

CLP fügt diesen Ideen einen weiteren Ansatz hinzu: Aus partieller Information über Relationen von Variablen können partielle Schlußfolgerungen gezogen werden. Wenn neue Information bzw. Constraints dem Speicher hinzugefügt werden, dann ist es Aufgabe des Constraintsolvers, den Speicher zu normalisieren und zu überprüfen, ob die vorhandene Information für den Schluß auf eine vorgegebene Menge von Constraints ausreichend ist ([FHK+92, vH91, JM94]).

Im Interesse rationeller Entwicklung korrekter und effizienter Software ist es wünschenswert, die das Problem beschreibenden Constraints möglichst problemnah zu formulieren. Dem steht das Black-Box-Konzept kommerziell verfügbarer Constraintsolver im Wege: Da der Anwender keinen Einblick in die Implementation des Solvers hat, sind Ergänzung und Modifikation aufwendig ([vHSD92]). Es besteht also Bedarf an benutzerdefinierten Constraintsolvern. In diese Lücke springt die Sprache Constraint Handling Rules (CHR, [Frü95, BFL+94]). CHR-Programme sind wegen des regelbasierten Formalismus von CHR leicht erweiterbar und modifizierbar.

Eine CHR-Regel beschreibt, unter welchen Bedingungen bestimmte Constraints durch andere Constraints ersetzt werden bzw. welche Constraints beim Auftreten bestimmter Constraints erzeugt werden. Im Gegensatz zu Prolog kennt CHR kein Rücksetzen, d.h. die Anwendung einer Regel kann nicht mehr rückgängig gemacht werden.

Dieser sogenannte „don't care"-Indeterminismus bringt Probleme mit sich: Für identische Eingaben kann das CHR-Programm unterschiedliche Antworten erzeugen. Das Ergebnis kann von der Reihenfolge der Regelanwendung abhängen. Das ist nicht erwünscht, denn schließlich ist es Aufgabe eines Constraintsolvers, vorgegebene Constraints in eine Normalform zu bringen. Bei einem Constraintsolver, der diese erfreuliche Eigenschaft, die Konfluenz genannt wird, nicht aufweist, liegt meistens ein Programmierfehler vor.

Eine bemerkenswerte Folge der Konfluenz zeigt [AFM96] für die Klasse von CHR-Programmen, die nur aus Simplifikations- und Simpagationsregeln bestehen (propagationsfreie Programme): Die deklarative Interpretation eines konfluenten, propagationsfreien Programms ist eine konsistente Theorie. Geht man schließlich von einem korrekten CHR-Programm aus, dann bedeutet die Konfluenz insbesondere, daß es für eine gegebene Constraintmenge keine zwei Berechnungen gibt, sodaß die eine die Inkonsistenz der Constraintmenge feststellt, während die andere die Constraintmenge zwar vereinfacht, aber dennoch mit einer Menge komplexer Constraints terminiert.

[AFM96] gibt außerdem ein entscheidbares, hinreichendes und notwendiges syntaktisches Kriterium für die Konfluenz von terminierenden propagationsfreien CHR-Programmen. Für CHR-Programme mit Propagationsregeln ist noch kein theoretisch abgesichertes Kriterium für die Konfluenz veröffentlicht.

Die Beiträge dieser Arbeit sind:

- Eine benutzerfreundliche und effiziente Implementation eines Konfluenz-Tests für propagationsfreie CHR-Programme auf der Grundlage des in [AFM96] gegebenen Kriteriums, die zur Durchführung von Experimenten aber auch Programme mit Propagationsregeln als Eingabe erlaubt. Implementationssprache ist Prolog. Der Konfluenz-Tester läuft auf ECL^{i}PS^e und dem frei erhältlichen SWI-Prolog.

- Eine Zusammenstellung der für die Implementation des Konfluenz-Tests notwendigen theoretischen Grundlagen. Für die Behandlung von nicht propagationsfreien Programmen wurde die Definition des kritischen Paares aus [AFM96] erweitert (s. 2.3.2).

- Eine Untersuchung darüber, welche Aussagen über vordefinierte und benutzerdefinierte Prolog-Prädikate in der „instanziierungsarmen" Umgebung eines Konfluenz-Tests gemacht werden können (s. Abschnitt 4). Die Ergebnisse der Untersuchung flossen in einen Prolog-Metainterpreter ein, der vom Konfluenz-Tester benutzt wird.

Ausgangspunkt für die Entwicklung des vorliegenden Konfluenz-Testers war ein Prototyp, der in [Meu96] vorgestellt wird.

Diese Arbeit ist wie folgt gegliedert: Nach einer Einführung in die theoretischen Grundlagen wird die Benutzerschnittstelle vorgestellt. Daran schließt sich die detaillierte Diskussion einer modellhaften Implementation an. Viel Raum wurde auch der Behandlung von Prolog-Prädikaten eingeräumt. Im Anhang findet sich der vollständige Quelltext des Konfluenz-Testers.

Kapitel 2

Constraint Handling Rules

2.1 Syntax von CHR

Vorausgesetzt sei eine Signatur, die die Menge der Constraintsymbole mit Stelligkeit, die Menge der Funktionssymbole mit Stelligkeit und die Menge der Variablen enthält. Die Mengen seien paarweise disjunkt. Die Menge der Constraintsymbole sei unterteilt in die disjunkten Mengen der benutzerdefinierten und der vordefinierten Constraintsymbole.

Definition 2.1.1 Ein Ausdruck der Form $p(t_1, \ldots, t_n)$ heißt *(benutzerdefiniertes/vordefiniertes) Constraint*, wenn p ein benutzerdefiniertes/vordefiniertes Constraintsymbol ist.

Definition 2.1.2 Seien $n, k, m \geq 1$. Sei $H_1, \ldots, H_n$ eine Konjunktion benutzerdefinierter Constraints und sei $G_1, \ldots, G_k$ eine Konjunktion vordefinierter Constraints, sodaß alle Variablen von $G_1, \ldots, G_k$ auch in $H_1, \ldots, H_n$ auftreten. Sei weiters $B_1, \ldots, B_m$ eine Konjunktion von benutzer- und vordefinierten Constraints. Dann heißt der Ausdruck

$$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_m$$

eine *Simplifikationsregel*. Der Ausdruck

$$H_1, \ldots, H_n \Rightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_m$$

heißt *Propagationsregel*. Für eine Simplifikationsregel der Form

$$H_1, \ldots, H_i, H_{i+1}, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid H_1, \ldots, H_i, B_1, \ldots, B_m$$

mit $i \geq 1$ und $n \geq 2$ schreibt man abkürzend eine *Simpagationsregel*

$$H_1, \ldots, H_i \backslash H_{i+1}, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_m.$$

$H_1, \ldots, H_n$ heißt *Regelkopf* oder *Kopf*, $G_1, \ldots, G_k$ heißt *Wächter* und $B_1, \ldots, B_m$ heißt *Regelrumpf* oder *Rumpf*. „|" ist der *Commit-Operator*.

Eine CHR-*Regel* (oder kurz *Regel*) ist eine Simplifikations-, Propagations- oder Simpagationsregel.

Notation 2.1.3 Wenn der Wächter einer Regel **true** ist, dann kann man ihn und den Commit-Operator auch weglassen. So schreibt man z.B. für ($H \Leftrightarrow$ **true** $\mid B$) abkürzend ($H \Leftrightarrow B$).

Definition 2.1.4 Alle Variablen, die im Kopf einer CHR-Regel R vorkommen, heißen *global* bezüglich R, alle anderen heißen *lokal* bezüglich R.

Definition 2.1.5 Ein CHR-*Programm* ist eine Menge von CHR-Regeln. Ein CHR-Programm ohne Propagationsregeln heißt *propagationsfrei*.

Bemerkung 2.1.6 Simpagationsregeln werden in dieser Arbeit nicht weiter behandelt, da eine Simpagationsregel durch eine Simplifikationsregel ausgedrückt werden kann.

2.2 Operationale Semantik von CHR

2.2.1 Vorbemerkungen

Für CHR sind drei sehr ähnliche operationale Semantiken in der Literatur dokumentiert: [Frü95] gibt eine informale operationale Semantik. [AFM96] enthält eine Formalisierung dieser Semantik. [Meu96] verfeinert diese Formalisierung. Zur vom CHR-Interpreter implementierten Semantik gibt es keine Veröffentlichung. Die hier gegebene operationale Semantik ist eine Zusammenstellung der für diese Arbeit relevanten Begriffe der operationalen Semantiken aus [AFM96] und [Meu96].

2.2.2 Constrainttheorien

Eine Constrainttheorie beschreibt die semantischen Eigenschaften der vordefinierten Constraints. Bis auf wenige Einschränkungen kann eine Constrainttheorie völlig frei sein, d.h. der beschriebene Constraintbereich kann beliebige Eigenschaften haben.

Definition 2.2.1 Eine Theorie T heißt *Constrainttheorie*, wenn sie die folgenden Eigenschaften hat:

- Es muß möglich sein eine update-Funktion v zu definieren (s. Definition 2.2.8). Das heißt, daß T eine Normalform für Konjunktionen von Constraints erlauben muß.
- T muß die Eigenschaften der Gleichheit $\doteq$ als Äquivalenzrelation beschreiben. Es genügt, wenn die Clark'sche Gleichheitstheorie (CET) in T enthalten ist.
- T muß die Eigenschaften der Constraints **true** (Wahrheitswert „wahr“) und **false** (Wahrheitswert „falsch“) beschreiben.

Über diese Forderungen hinaus ist die Wahl der in T beschriebenen Constraints und der semantischen Eigenschaften dieser Constraints frei.

Notation 2.2.2 **CT** bezeichnet eine beliebige Constrainttheorie.

2.2.3 Zustände

Notation 2.2.3 $\square$ bezeichnet die leere Konjunktion.

Definition 2.2.4 Ein *Zielatom* ist ein Constraint. Ein *Ziel* ist eine Konjunktion von Zielatomen.

Definition 2.2.5 Ein *Zustand* ist ein 4-Tupel

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

wobei G ein Ziel, C_U eine Konjunktion von benutzerdefinierten Constraints, C_B eine Konjunktion von vordefinierten Constraints und $\mathcal{V}$ eine geordnete Liste von Variablen ist. G heißt *Zielspeicher*, C_U heißt *Speicher für benutzerdefinierte Constraints* und C_B heißt *Speicher für vordefinierte Constraints*.

Definition 2.2.6 Für einen Zustand $\langle G, C_U, C_B, \mathcal{V} \rangle$ heißt eine Variable *global*, wenn sie in $\mathcal{V}$ vorkommt, andernfalls heißt sie *lokal*.

Notation 2.2.7 $\mathcal{S}$ bezeichnet die Menge aller Zustände.

2.2.4 Update-Funktionen

Die grundlegenden Operationen eines eingebauten Constraintsolvers sind die Vereinfachung und die Normalisierung des Speichers für vordefinierte Constraints. Diese Operation kann durch eine update-Funktion beschrieben werden.

Definition 2.2.8 Eine Funktion $v : \mathcal{S} \mapsto \mathcal{S}$ heißt *update-Funktion*, wenn sie die folgenden Bedingungen erfüllt:

- v ist korrekt, d.h. wenn $v(\langle Gs, C_U, C_B, \mathcal{V} \rangle) = \langle Gs', C'_U, C'_B, \mathcal{V} \rangle$, dann gilt $\mathbf{CT} \models C_B \leftrightarrow C'_B$. ($C_B$ und C'_B sollen also logisch äquivalent sein.)
- Für einen Zustand $\langle Gs, C_U, C_B, \mathcal{V} \rangle$ mit einem Gleichheitsconstraint $X \doteq t$ in C_B ersetzt v alle Vorkommen von X in Gs, C_U und C_B (außer in $X \doteq t$ selbst) durch t.
- Für einen Zustand $\langle Gs, C_U, C_B, \mathcal{V} \rangle$ entfernt v alle Gleichheitsconstraints der Form $X \doteq t$ und $Y \doteq X$, wenn X lokal ist.
- v ist idempotent, d.h. für jeden Zustand S gilt $v(v(S)) = v(S)$.

Für das Folgende sei vorausgesetzt, daß mit v bezeichnete Funktionen update-Funktionen sind.

2.2.5 Entailment

Definition 2.2.9 *Entailment* ($\rightarrow_o$) ist wie folgt definiert:

$$\langle Gs_1, C_{U1}, C_{B1}, \mathcal{V} \rangle \rightarrow_o \langle Gs_2, C_{U2}, C_{B2}, \mathcal{V} \rangle$$

genau dann, wenn

$$\langle Gs'_1, C'_{U1}, C'_{B1}, \mathcal{V} \rangle = v(\langle Gs'_2, C'_{U2}, C'_{B1} \wedge C'_{B2}, \mathcal{V} \rangle),$$

wobei

$$\langle Gs'_1, C'_{U1}, C'_{B1}, \mathcal{V} \rangle = v(\langle Gs_1, C_{U1}, C_{B1}, \mathcal{V} \rangle)$$

und

$$\langle Gs'_2, C'_{U2}, C'_{B2}, \mathcal{V} \rangle = v(\langle Gs_2, C_{U2}, C_{B2}, \mathcal{V} \rangle).$$

Satz 2.2.10 $\langle Gs, C_U, C_B, \mathcal{V} \rangle \rightarrow_o \langle Gs, C_U, C'_B, \mathcal{V} \rangle$ gilt genau dann, wenn $\mathbf{CT} \models C_B \rightarrow C'_B$.

2.2.6 Berechnungsschritte

Notation 2.2.11 Wenn c ein benutzerdefiniertes Constraint ist, dann ist

$$c(t_1, \ldots, t_n) \doteq c(s_1, \ldots, s_n)$$

eine Abkürzung für

$$t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n.$$

Definition 2.2.12 Die Zustandsübergangsrelation $\rightarrow_P$ ist für ein Programm P durch die Berechnungsschritte **Introduce**, **Solve**, **Simplify**, **Propagate** und **Idempotence** definiert. Alle Berechnungsschritte dürfen nur auf Zustände angewendet werden, deren Speicher für vordefinierte Constraints ungleich **false** ist.

- **Introduce** transportiert ein benutzerdefiniertes Constraint aus dem Zielspeicher in den Speicher für benutzerdefinierte Constraints:

  $$\langle C \wedge Gs, C_U, C_B, \mathcal{V}\rangle \rightarrow_P \langle Gs, C \wedge C_U, C_B, \mathcal{V}\rangle$$

- **Solve** transportiert ein vordefiniertes Constraint aus dem Zielspeicher in den Speicher für vordefinierte Constraints und normalisiert den resultierenden Zustand:

  $$\langle C \wedge Gs, C_U, C_B, \mathcal{V}\rangle \rightarrow_P v(\langle Gs, C_U, C \wedge C_B, \mathcal{V}\rangle)$$

- **Simplify** wendet eine Simplifikationsregel an:

  $$\langle Gs, C_U, C_B, \mathcal{V}\rangle \rightarrow_P v(\langle Gs \wedge B, C'_U, H \doteq H' \wedge C_B, \mathcal{V}\rangle),$$

  wenn $H' \wedge C'_U$ eine Umordnung von C_U ist, $(H \Leftrightarrow G \mid B)$ eine Variante mit frischen Variablen einer Regel aus P ist und

  $$\langle H', \Box, C_B, []\rangle \rightarrow_o \langle H', \Box, H' \doteq H \wedge G, []\rangle$$

  gilt.

Die folgenden Schritte sind nur dann Teil der Definition von $\rightarrow_P$, wenn P nicht propagationsfrei ist.

- **Propagate** wendet eine Propagationsregel an:

  $$\langle Gs, C_U, C_B, \mathcal{V}\rangle \rightarrow_P v(\langle Gs \wedge B, C_U, H \doteq H' \wedge C_B, \mathcal{V}\rangle),$$

  wenn $H' \wedge C'_U$ eine Umordnung von C_U ist und $(H \Rightarrow G \mid B)$ eine Variante mit frischen Variablen einer Regel aus P ist,

  $$\langle H', \Box, C_B, []\rangle \rightarrow_o \langle H', \Box, H' \doteq H \wedge G, []\rangle$$

  gilt und eine Anwendungsbedingung (s. unten) erfüllt ist.

- **Idempotence** entfernt redundante benutzerdefinierte Constraints:

  $$\langle Gs, C_U, C_B, \mathcal{V}\rangle \rightarrow_P \langle Gs, C_1 \wedge C'_U, C_B, \mathcal{V}\rangle,$$

  wenn C_U eine Umordnung von $C_1 \wedge C_2 \wedge C'_U$ ist und

  $$\langle \Box, C_1 \wedge C_2, C_B, []\rangle \rightarrow_o \langle \Box, C_1 \wedge C_2, C_1 \doteq C_2, []\rangle$$

  gilt.

Simplify und **Propagate** sind die Berechnungsschritte, die den Speicher für benutzerdefinierte Constraints gemäß P vereinfachen. Wenn der Kopf H einer Regel an benutzerdefinierte Constraints H' angeglichen[1] werden kann und der Wächter G in der resultierenden Umgebung erfüllt ist, dann feuert die Regel: Der Rumpf B wird dem Zielspeicher hinzugefügt und **Simplify** entfernt außerdem H' aus dem Speicher für benutzerdefinierte Constraints.

Angleichen von H an H' und Test des Wächters G bringen die Definitionen von **Simplify** und **Propagate** durch Anwendung von $\rightarrow_o$ zum Ausdruck:

$$\langle H', \Box, C_B, []\rangle \rightarrow_o \langle H', \Box, H' \doteq H \wedge G, []\rangle$$

ist genau dann wahr, wenn $H' \doteq H$ und G keine Gleichheitsconstraints für die Variablen von H' enthalten und G keine sonstigen Constraints enthält, die Information tragen, die noch nicht in C_B enthalten ist.

[AFM96] definiert den Zustand als Tripel. Zwischen Zielspeicher und Speicher für benutzerdefinierte Constraints wird nicht unterschieden. Für diese Arbeit wurde die in [Meu96] definierte Zustandsrepräsentation gewählt. Die 4-Tupel-Darstellung macht den **Introduce**-Schritt notwendig. Dadurch ergeben sich jedoch keine Unterschiede zur in [AFM96] definierten operationalen Semantik:

- **Simplify**, **Propagate** und **Idempotence** können erst dann auf ein neues Constraint angewendet werden, wenn es von **Introduce** vom Zielspeicher in den Speicher für benutzerdefinierte Constraints transportiert wurde.

- Nach einem **Solve**-Schritt ist jeder **Introduce**-Schritt möglich und umgekehrt.

Somit kann auf die Ergebnisse aus [AFM96] Bezug genommen werden.

Propagate und **Idempotence** stammen aus [Meu96]. Sie werden dort zur Behandlung von nicht propagationsfreien CHR-Programmen eingeführt und zusammen mit mehreren Anwendungsbedingungen, die triviale Nicht-Terminierung verhindern sollen, diskutiert. Allerdings gibt [Meu96] kein Kriterium für die Konfluenz von nicht propagationsfreien CHR-Programmen. **Propagate** und **Idempotence** wurden in diese Arbeit übernommen, um einen Rahmen für das Experimentieren mit solchen Programmen zu schaffen.

Eine Anwendungsbedingung, die [Meu96] als theoretisch gut handhabbar einstuft, die Terminierung aber nicht in allen Fällen sicherstellt, ist die *Redundanzbedingung*. Diese fordert, daß die Instanz einer Propagationsregel nur dann angewendet werden darf, wenn sie „neue“ Information erzeugt.

[1] engl. *matching*

Der ECL^{i}PS^e-CHR-Interpreter implementiert eine andere Anwendungsbedingung, die *Instanzbedingung mit Indizierung*. Diese fordert, daß die Instanz einer Propagationsregel nur dann angewendet werden darf, wenn sie noch nie angewendet worden ist ([Frü95], 3.4), 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 mit Indizierung sichert die Terminierung in mehr Fällen als die Redundanzbedingung, wird in [Meu96] aber als theoretisch schlecht handhabbar eingestuft.

Eine Variante der Instanzbedingung mit Indizierung ist die *Instanzbedingung ohne Indizierung*. Hier wird zwischen syntaktischen gleichen Constraints nicht unterschieden.

2.2.7 Äquivalenz von Zuständen

Definition 2.2.13 Zustände $S = \langle Gs, C_U, C_B, \mathcal{V}\rangle$ und $S' = \langle Gs', C'_U, C'_B, \mathcal{V}'\rangle$ heißen *äquivalent* ($S \cong S'$), wenn eine der folgenden Bedingungen erfüllt ist:

- Wenn $C_B = C'_B = \mathbf{false}$.
- Wenn es Umordnungen $\dot{Gs}$, $\dot{C_U}$ und $\dot{C_B}$, von Gs, C_U und C_B gibt, sodaß $\langle \dot{Gs}, \dot{C_U}, \dot{C_B}, \mathcal{V}\rangle = \langle Gs', C'_U, C'_B, \mathcal{V}'\rangle$.

2.2.8 Ableitungen und Berechnungen

Definition 2.2.14 Für Zustände S und T gilt $S \to_P^* T$ genau dann, wenn

- $S = T$ oder
- $T = v(S)$ oder
- $S \to_P S_1 \to_P \ldots \to_P S_n \to_P T$ $(n \geq 0)$.

Der Ausdruck $S \to_P^* T$ heißt *endliche Ableitung* für S. T heißt das *Ergebnis* von $S \to_P^* T$.

Wenn die Zwischenzustände $S_1, \ldots, S_n$ einer endlichen Ableitung $S \to_P S_1 \to_P \ldots \to_P S_n \to_P T$ nicht von Bedeutung sind, dann schreibt man abkürzend $S \to_P^* T$.

Definition 2.2.15 Ein Zustand der Form $\langle G, \square, \mathbf{true}, \mathcal{V}\rangle$ heißt *Anfangszustand*, wenn $\mathcal{V}$ die geordnete Menge aller Variablen von G ist.

Ein Zustand $\langle G, C_U, C_B, \mathcal{V}\rangle$ heißt *Endzustand*, wenn $C_B = \mathbf{false}$ oder kein Zustandsübergang mehr möglich ist und $C_B \neq \mathbf{false}$.

Definition 2.2.16 Eine *endliche Berechnung* für einen Zustand S ist eine endliche Ableitung $S \to_P^* T$ mit Endzustand T.

Eine *endliche Berechnung* für ein Ziel G ist eine endliche Ableitung $S \to_P^* T$ mit Anfangszustand $S = \langle G, \square, \mathbf{true}, \mathcal{V}\rangle$ und Endzustand T.

Eine endliche Berechnung $S \to_P^* \langle G, C_U, C_B, \mathcal{V}\rangle$ heißt *gescheitert*, wenn $C_B = \mathbf{false}$. Andernfalls heißt sie *erfolgreich*.

Ein Zustand T heißt *Endzustand für einen Zustand* S, wenn T ein Endzustand ist und es eine endliche Berechnung $S \to_P^* T$ gibt.

Definition 2.2.17 Zustände S und T heißen *zusammenführbar* [2] ($S \downarrow T$), wenn es Zustände S' und T' mit $S' \cong T'$ gibt, sodaß $S \rightarrow_P^* S'$ und $T \rightarrow_P^* T'$.

Bemerkung 2.2.18 Sei P ein propagationsfreies Programm und seien S und T Zustände. Wenn es endliche Ableitungen $S \rightarrow_P^* S'$ und $T \rightarrow_P^* T'$ mit $S' \cong T'$ gibt, dann gibt es endliche Berechnungen $S \rightarrow_P^* S' \rightarrow_P^* S''$ und $T \rightarrow_P^* T' \rightarrow_P^* T''$ mit $S'' \cong T''$. Das heißt, daß Zustände, die durch endliche Ableitungen zusammenführbar sind, auch durch endliche Berechnungen, die die Ableitungen fortsetzen, zusammenführbar sind.

Wenn P nicht propagationsfrei ist, dann hängt die Gültigkeit dieser Aussage von der Wahl der Anwendungsbedingung ab. Mit der Redundanzbedingung gilt diese Aussage, mit einer Instanzbedingung gilt sie nicht.

2.3 Konfluenz von CHR-Programmen

Von zentraler Bedeutung ist der Begriff des kritischen Paares, welcher der Theorie der Conditional Term Rewriting Systems (CTRS, [DOS88]) entliehen ist. [AFM96] adaptiert diesen Begriff für propagationsfreie CHR-Programme. Da der Konfluenz-Test auch CHR-Programme mit Propagationsregeln behandeln soll, muß die Definition aus [AFM96] hier erweitert werden.

Definition 2.3.1 Ist R eine CHR-Regel und ist H der Kopf von R, dann berechnet

$$\text{survive}(R) := \begin{cases} \text{true} & \text{, falls } R \text{ eine Simplifikationsregel ist} \\ H_1 \wedge \ldots \wedge H_n, & \text{falls } R \text{ eine Propagationsregel ist} \\ & \text{und } H_1, \ldots, H_n \text{ der Kopf von } R \text{ ist} \end{cases}$$

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

Definition 2.3.2 (Kritisches Paar) Sei R eine CHR-Regel mit Wächter G, Rumpf B und Kopf H und sei R' eine CHR-Regel mit Wächter G', Rumpf B' und Kopf H', sodaß R und R' keine Variablen gemeinsam haben. [3] R und R' können bis auf Variablenumbenennung gleich sein. Seien weiters $\{H_i \,|\, 1 \leq i \leq n\}$ und $\{H'_i \,|\, 1 \leq i \leq m\}$ die Mengen der Atome in H beziehungsweise H'.

Wenn ein oder mehrere Atome $H_{i_1}, \ldots, H_{i_k}$ des Kopfs von R mit einem oder mehreren Atomen $H'_{j_1}, \ldots, H'_{j_k}$ des Kopfs von R' unifizieren, dann heißt das Tripel

$$\left(\begin{array}{c} G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k} \mid \\ B \wedge \text{survive}(R) \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m} \downarrow B' \wedge \text{survive}(R') \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n} \mid \\ \mathcal{V} \end{array} \right)$$

ein kritisches Paar von R und R'. $\{i_1, \ldots, i_n\}$ und $\{j_1, \ldots, j_m\}$ sind dabei Permutationen von $\{1, \ldots, n\}$ beziehungsweise $\{1, \ldots, m\}$. $\mathcal{V}$ ist die Liste der Variablen in $H_1, \ldots, H_n, H'_1, \ldots, H'_m$.

[2] engl. *joinable*

[3] Das bedeutet keine Einschränkung der Allgemeinheit, denn jedes CHR-Programm kann durch Variablenumbenennung in ein Programm transformiert werden, sodaß keine Regel Variablen mit einer anderen Regel gemeinsam hat. ([AFM96], 3.4)

Das „Paar“, auf das der Begriff zurückgeht, besteht aus den zwei Termen links und rechts vom $\downarrow$. Der Term links vom $\downarrow$ heißt *linke Seite des kritischen Paares*, der Term rechts davon heißt *rechte Seite des kritischen Paares*. Das erste Element des Tripels heißt *Wächter des kritischen Paares*.

Ein kritisches Paar modelliert die Situation, daß zwei Regeln bzw. zwei Varianten einer Regel auf denselben Constraints feuern können. Das Feuern auf denselben Constraints kommt in der Modellierung durch die Überlappung von Kopfatomen 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.

Seien die Bezeichnungen aus der Definition vorausgesetzt. Dann stellen die Seiten eines kritischen Paares die Constraints von Zuständen T und T' gegenüber, die man durch Anwendung von R bzw. R' auf einen Zustand

$$S = \langle \Box, H'_{j_1} \wedge \ldots \wedge H'_{j_m} \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n}, H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}, \mathcal{V} \rangle$$

erhält. S repräsentiert die Zustände, auf die R und R' angewendet werden können, wenn die $H_{i_1}, \ldots, H_{i_k}$ mit den $H'_{j_1}, \ldots, H'_{j_k}$ unifizieren.

Die durch ein kritisches Paar beschriebene Situation ist theoretisch und praktisch von Bedeutung, wie aus der folgenden Aufstellung von Definitionen und Sätzen aus [AFM96] hervorgeht, die direkt zu einer Vorgehensweise für einen Konfluenz-Test führen.

Definition 2.3.3 Ein kritisches Paar $(G \mid B_1 \downarrow B_2 \mid \mathcal{V})$ heißt *zusammenführbar*, wenn die Zustände $\langle B_1, \Box, G, \mathcal{V} \rangle$ und $\langle B_2, \Box, G, \mathcal{V} \rangle$ zusammenführbar sind.

Definition 2.3.4 Ein CHR-Programm heißt *konfluent*, wenn für alle Zustände S, S_1, S_2 gilt:

Wenn $S \rightarrow_P^* S_1$ und $S \rightarrow_P^* S_2$, dann sind S_1 und S_2 zusammenführbar.

Definition 2.3.5 Ein CHR-Programm heißt *lokal konfluent*, wenn für alle Zustände S, S_1, S_2 gilt:

Wenn $S \rightarrow_P S_1$ und $S \rightarrow_P S_2$, dann sind S_1 und S_2 zusammenführbar.

Definition 2.3.6 Ein CHR-Programm heißt *terminierend*, wenn jede mögliche Berechnung aller möglichen Ziele endlich ist.

Satz 2.3.7 (Konfluenz von propagationsfreien CHR-Programmen) Ein terminierendes und lokal konfluentes propagationsfreies CHR-Programm ist konfluent.

Satz 2.3.8 (Lokale Konfluenz von propagationsfreien CHR-Programmen) Ein propagationsfreies CHR-Programm ist genau dann lokal konfluent, wenn all seine kritischen Paare zusammenführbar sind.

Diese beiden Sätze liefern die Vorgehensweise für einen Konfluenz-Test: Um schließen zu können, daß ein terminierendes propagationsfreies CHR-Programm konfluent ist, muß man zeigen, daß alle kritischen Paare des Programms zusammenführbar sind.

2.4 Konfluenz von Constraintsymbolen

Bei der Programmanalyse ist eine Möglichkeit zur Untersuchung ausgewählter Constraintsymbole hilfreich. So kann man Schwachpunkte am untersuchten Programm genau lokalisieren und verbessern. Bei Constraintsymbolen in einem terminierendem Programm unterscheidet man daher zwei Arten von Konfluenz:

Definition 2.4.1 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 und alle Constraintsymbole, auf die sich p indirekt abstützt, partiell konfluent sind.

Eine Menge C von Constraintsymbolen heißt *total (partiell) konfluent*, wenn alle Constraintsymbole in C total (partiell) konfluent sind.

Es fehlt noch die folgende rekursive Definition:

Definition 2.4.2 Ein Constraintsymbol *p stützt sich auf ein 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 ein Constraint q indirekt 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 indirekt auf q abstützt.

Der folgende, leicht einzusehende Satz zeigt Zusammenhänge zwischen den bisher definierten Begriffen auf und rechtfertigt damit die Definitionen 2.4.1 und 2.4.2.

Satz 2.4.3 Für jedes beliebige CHR-Programm P und die Menge C der Constraintsymbole, die in P definiert werden (also in mindestens einem Regelkopf vorkommen), gilt:

1. Alle Constraintsymbole in C sind total konfluent, wenn alle Constraintsymbole in C partiell konfluent ist.

2. P ist konfluent, wenn alle Constraintsymbole in C total konfluent sind.

Kapitel 3

Der Konfluenz-Tester

3.1 Die Anforderungen

Der Katalog der Anforderungen an den Konfluenz-Tester wurde im Verlauf des Praktikums öfters nachgebessert, um neue praktische Erfahrungen und theoretische Erkenntnisse zu berücksichtigen, und umfaßte schließlich die folgenden Punkte:

- Für ein benutzerdefiniertes Constraint sollen alle kritischen Paare ausgegeben werden können.
- Ein benutzerdefiniertes Constraint soll auf partielle und totale Konfluenz getestet werden können. Für die Suche nach äquivalenten Endzuständen soll ein vom CHR-Interpreter unabhängiger Expansionsalgorithmus eingesetzt werden. Der Algorithmus soll terminieren, wenn
  - der Speicher für eingebaute Constraints inkonsistent ist ([Frü95], 3.3) oder
  - keine Regelinstanz mehr anwendbar ist. Die Anwendbarkeit der Instanz einer Propagationsregel soll mit der Instanzbedingung *ohne* Indizierung entschieden werden.
- Nach der Bildung eines kritischen Paares soll der Benutzer Gelegenheit erhalten den Wächter des kritischen Paares im Dialog zu vereinfachen.
- Das Programm soll in Prolog geschrieben werden und unter ECL^{i}PS^e laufen.

Weiters erschienen wünschenswert

- die Kompatibilität zu dem frei verfügbaren SWI-Prolog-System, um die Arbeit am Programm auch zu Unzeiten auf dem Heimrechner fortsetzen zu können, für dessen Betriebssystem ECL^{i}PS^e nicht verfügbar ist, und
- das selektive Mitschreiben von Informationen über den Programmablauf zur Erleichterung der Fehlersuche.

Die zwei letzten Punkte waren jedoch nicht gefordert und wurden auf eigene Initiative hin implementiert.

3.2 Aufbau und Ablauf einer Sitzung

Aufbau und Ablauf einer Sitzung mit dem Konfluenz-Tester gestalten sich wie folgt:

1. Wechseln in das Verzeichnis, im dem sich die Dateien des Testers befinden.
2. Laden der Prolog-Umgebung.
3. Laden des Konfluenz-Testers. Unter ECL^{i}PS^e gibt man dazu das Kommando '`[eclt].`', unter SWI-Prolog '`[swit].`'
4. Laden eines Constraintsolvers.
5. Arbeiten mit dem Konfluenz-Tester.
6. Verlassen der Prolog-Umgebung.

3.3 Die Benutzerschnittstelle

Die grundlegenden Prädikate der Benutzerschnittstelle sind:

- `load(+Dateiname)` lädt einen Constraintsolver und schreibt eine Liste der geladenen Constraints. Der Dateiname darf die Dateikennung `.chr` nicht enthalten, da diese automatisch ergänzt wird.
- `lacp` [1] schreibt eine Liste der kritischen Paare für den geladenen Constraintsolver.
- `lcp(+Constraintspezifikation)` [2] schreibt eine Liste der kritischen Paare für das spezifizierte Constraint. [3]
- `partial(+Constraintspezifikation)` überprüft das spezifizierte Constraint auf partielle Konfluenz und schreibt alle nicht zusammenführbaren kritischen Paare.
- `total(+Constraintspezifikation)` überprüft das spezifizierte Constraint auf totale Konfluenz und schreibt alle nicht zusammenführbaren kritischen Paare.
- `confluent` überprüft den geladenen Constraintsolver auf Konfluenz und schreibt alle nicht zusammenführbaren kritischen Paare.
- `based_upon(+Constraintspezifikation, -Liste)` unifiziert *Liste* mit einer Liste der Constraints, auf die sich die Definition des spezifizierten Constraints stützt.

Ein Constraint wird eindeutig spezifiziert durch Angabe des Namens und der Arität in der Form *Name/Arity*. Eine Spezifikation ohne Arität wird automatisch um die Arität ergänzt, wenn eine eindeutige Zuordnung möglich ist. Ist die Spezifikation mehrdeutig, dann wird eine Liste der in Frage kommenden Constraints ausgegeben.

[1] `lacp` steht für *list all critical pairs.*

[2] `lcp` steht für *list critical pairs.*

[3] Ein kritisches Paar ist genau dann ein kritisches Paar für ein Constraint, wenn das Constraint in der Liste der gemeinsamen Kopfatome vorkommt/ im Wächter aufscheint.

Zum Mitschreiben der Ausgaben in eine Datei gibt es die Prädikate `lacp/1`, `lcp/2`, `partial/2`, `total/2` und `confluent/1`, die als zusätzliches Argument einen Dateinamen erwarten. Existiert die angegebene Datei bereits, dann wird sie überschrieben.

Zur Erleichterung der Fehlersuche generiert der Konfluenz-Tester nach jedem wichtigen Schritt eine Nachricht. Auch Ergebnisse werden in Form von Nachrichten gefaßt. Da jede Nachricht einer Nachrichtenklasse zugewiesen ist, ist das Auswählen von Nachrichten aus dem Nachrichtenstrom möglich.

Es gibt zwei Schnittstellen zur Nachrichtenauswahl. Die eine besteht aus zwei Schaltern, die mit den folgenden Prädikaten bedient werden:

- `joinable_on` schaltet das Mitschreiben zusammenführbarer kritischer Paare ein. Diese Einstellung hat den Vorteil, daß kritische Paare schon vor dem Versuch des Zusammenführens geschrieben werden, sodaß klar wird, welche kritischen Paare schwierig zu behandeln sind. In der Grundeinstellung werden nur nicht zusammenführbare kritische Paare geschrieben.

- `joinable_off` schaltet in die Grundeinstellung zurück.

- `trace_on` schaltet das Mitschreiben aller Nachrichten für `partial/*`, `total/*` und `confluent/*` ein. Diese Einstellung eignet sich zur Fehlersuche und zum Nachvollzug der Schritte, mit denen die Zusammenführbarkeit eines kritischen Paare gezeigt oder widerlegt wurde.

- `trace_off` schaltet in die Grundeinstellung zurück.

Die andere Schnittstelle zur Nachrichtenauswahl ist implizit durch das Prädikat definiert, auf dem alle Variationen des Konfluenz-Tests aufsetzen:

- `partial(+Constraintspezifikation, loginfo(+Nachrichtenklassen, +Dateiname, +Strom))` überprüft das spezifizierte Constraint auf partielle Konfluenz und schreibt eine Nachricht dann in den Strom *Strom*, wenn ihre Klasse in der Liste `Nachrichtenklassen` vorkommt. Es werden Nachrichten der Klassen `user_info`, `feasible`, `expand`, `joinable`, `solve`, `idempotence`, `unfold`, `noisy_unfold`, `entails`, `update`, `joinable_critical_pairs` und `unjoinable_critical_pairs` erzeugt. Ihre Bedeutung wird in der Besprechung der Details der Implementation deutlich.

In der Grundeinstellung erfolgt keine Benutzeranfrage zur Vereinfachung der Wächter von kritischen Paaren. Geändert werden kann das mit einem Schalter, der mit den folgenden Prädikaten bedient wird:

- `consult_on` schaltet die interaktive Wächter-Vereinfachung ein.

- `consult_off` schaltet in die Grundeinstellung zurück.

Ebenfalls wird in der Grundeinstellung die Idempotenz nicht angewendet. Auch hier gibt es einen Schalter, der mit den folgenden Prädikaten bedient wird:

- `idempotence_on` schaltet die Anwendung der Idempotenz ein.

- `idempotence_off` schaltet in die Grundeinstellung zurück.

3.4 Vorbemerkungen zur Diskussion der Implementation

Die im folgenden gegebenen Programme entsprechen zuweilen nicht der tatsächlichen Implementation; zugunsten von Kürze und Klarheit beschränkt sich die Wiedergabe auf das Wesentliche. Insbesondere fehlen diejenigen Parameter und Prozeduraufrufe, die das Mitschreiben implementieren. Die vollständigen Programme können im Anhang nachgelesen werden.

Einige Bemerkungen zur Notation: Im folgenden steht ein im laufenden Text kursiv gesetzter Großbuchstabe für einen beliebigen Prolog-Term. Ein im laufenden Text in Schreibmaschinenschrift gesetzter Großbuchstabe steht für eine Prolog-Variable. Ein T steht also für eine Variable, z.B. `X`, oder eine atomare Formel, z.B. `q(a, X)`. `X` steht jedoch nur für sich selbst. Die Einführung einer Metasprache ist hier notwendig, um über Variablen der Objektsprache reden zu können. Definitionen von Prädikaten sind aber wie üblich in Schreibmaschinenschrift gesetzt, darin vorkommende Prolog-Variablen stehen für beliebige Terme. Anfragen an das Prolog-System sind ebenfalls in Schreibmaschinenschrift gesetzt. Ein kursiv gesetzter Großbuchstabe steht hier wieder für einen beliebigen Prolog-Term, ein in Schreibmaschinenschrift gesetzter Großbuchstabe steht für eine Prolog-Variable. Um den Lesefluß nicht zu stören, fehlt bei im laufenden Text vorkommenden Anfragen an das Prolog-System der terminierende Punkt.

3.5 Die Regelrepräsentation

Die Form, in der Regeln eines CHR-Programms vorliegen, ist für die Verarbeitung durch ein Prolog-Programm ungünstig. Die CHR-Syntax erlaubt es, die gleiche Regel auf verschiedene Weisen zu notieren, denn Regelname und Wächter sind optional. Kopf, Wächter und Rumpf von Regeln liegen als Konjunktionen vor, Prolog unterstützt Listen aber besser als Konjunktionen. Regelnamen müssen nicht eindeutig sein. Für eine Implementation der Instanzbedingung sind eindeutige Regelnamen günstig. Es empfiehlt sich daher die Einführung einer Regelnormalform, die genau auf die Bedürfnisse der Implementation abgestimmt ist. Da ein schneller Regelzugriff von entscheidender Bedeutung für die Effizienz einer Implementation ist, empfiehlt es sich weiters, die Regeln schon beim Laden des CHR-Programms in Normalform zu bringen. In der vorliegenden Implementation wird eine Regel als Fakt `rule/7` repräsentiert. Die Argumente der Reihe nach:

1. Die Regel, wie sie im Programm steht.
2. Die Regel in Normalform mit eindeutigem, eventuell automatisch generiertem Namen und Wächter. Wenn die Originalregel keinen Wächter hat, dann ist der Wächter der Normalform gleich **true**.
3. Der eindeutige Regelname.
4. Der Kopf als Liste.
5. Der Wächter als Liste. Die leere Liste steht für **true**.
6. Der Rumpf als Liste. Die leere Liste steht für **true**.
7. Die Liste der Atome des Kopfes, die bei Regelanwendung im Speicher bleiben. Für Simplifikationsregeln ist das die leere Liste, für Propagationsregeln ist das der ganze Kopf.

3.6 Die Repräsentation kritischer Paare

Das Bilden kritischer Paare an sich ist nicht teuer, doch da der dazu verwendete Algorithmus doppelte kritische Paare erzeugt und das einen Konfluenz-Test verlangsamt, ist es ratsam, eine Datenbank kritischer Paare anzulegen. Damit können redundante kritische Paare erkannt und vermieden und damit die Effizienz erhöht werden. In der vorliegenden Implementation wird ein kritisches Paar als Fakt `critical_pair/7` repräsentiert. Die Argumente sind der Reihe nach:

1. Der Name der ersten Regel.
2. Der Name der zweiten Regel.
3. Die Liste der unifizierten Kopfatome.
4. Der Wächter als Liste. Die leere Liste steht für **true**.
5. Die linke Seite als Liste. Die leere Liste steht für **true**.
6. Die rechte Seite als Liste. Die leere Liste steht für **true**.
7. Die Liste der globalen Variablen.

3.7 Die Bildung kritischer Paare

`build_critical_pair(`*Name*`/`*Arity*`,` *CP*`)` ist erfolgreich, wenn es ein kritisches Paar für das Constraintsymbol mit Namen *Name* und Stelligkeit *Arity* gibt. Im Erfolgsfall wird *CP* mit einem Fakt `critical_pair/7` unifiziert, der das kritische Paar repräsentiert. `build_critical_pair` ist wiedererfüllbar, sodaß durch eine failure driven loop alle kritischen Paare für ein Constraintsymbol gewonnen werden können. Bleibt das Constraintsymbol unspezifiziert, dann können durch eine failure driven loop alle kritischen Paare für das geladene CHR-Programm gewonnen werden.

`build_critical_pair` holt zunächst zwei Regeln, in deren Köpfen das spezifizierte Constraintsymbol vorkommt. Nur wenn in beiden Köpfen das spezifizierte Constraintsymbol vorkommt, ist es möglich, aus den Regeln ein kritisches Paar für das spezifizierte Constraintsymbol zu bilden. Da zwei Regeln R_0 und R_1 in zwei Reihenfolgen, nämlich (R_0, R_1) und (R_1, R_0), gewonnen werden können, ist es zur Vermeidung der Bildung doppelter kritischer Paare notwendig, nur eine Reihenfolge zu betrachten. Das wird durch Anwendung von `rule_geq`, das eine Ordnung auf Regeln implementiert, erreicht.

Die Unifikation von Kopfatomen wird von `unify_heads` erledigt. Um sicherzustellen, daß das kritische Paar ein kritisches Paar für das spezifizierte Constraintsymbol ist, muß anschließend überprüft werden, ob das spezifizierte Constraintsymbol in der Liste der unifizierten Atome vorkommt. Die weitere Bildung der Komponenten des kritischen Paares folgt der Definition.

Eine Möglichkeit zur Verbesserung der Effizienz soll noch erläutert werden: Um die Wahrscheinlichkeit der Erkennung von doppelten kritischen Paaren beim Erstellen der Datenbank kritischer Paare und von trivialerweise zusammenführbaren kritischen Paaren bei der Entscheidung der Zusammenführbarkeit zu erhöhen, werden die linke und rechte Seite und die Liste der unifizierten Kopfatome sortiert.

```
build_critical_pair(Name/Arity,
                    critical_pair(Name1, Name2,
                                  Shared_atoms,
                                  Guard, Left, Right, Global_vars)) :-

    % select rules from the CHR program
    rule(Rule1, _, Name1, Head1, Guard1, Body1, Surviving_atoms1),
    for_one(name_and_arity(Name/Arity), Head1),
    rule(Rule2, _, Name2, Head2, Guard2, Body2, Surviving_atoms2),
    for_one(name_and_arity(Name/Arity), Head2),

    % avoid some duplicate critical pairs
    rule_geq(Rule1, Rule2),

    % make conjunction of guards
    append(Guard1, Guard2, Guard),

    % unify head atoms
    unify_heads(Head1, Head2, Shared_atoms0,
                Not_shared_atoms1, Not_shared_atoms2),
    for_one(name_and_arity(Name/Arity), Shared_atoms0),
    msort(Shared_atoms0, Shared_atoms),

    % make left side
    flatten([Body1, Not_shared_atoms2, Surviving_atoms1], Left0),
    msort(Left0, Left),

    % make right side
    flatten([Body2, Not_shared_atoms1, Surviving_atoms2], Right0),
    msort(Right0, Right),

    % make list of global variables
    term_variables(Head1-Head2, Global_vars).


rule_geq(Rule1, Rule2) :-
    copy_term(Rule1-Rule2, Rule1_copy-Rule2_copy),
    term_variables(Rule1_copy-Rule2_copy, Vars),
    checklist(setvar(dummy), Vars),
    Rule1_copy @>= Rule2_copy.

% little helper
setvar(Y, Y).
```

`unify_heads(`H_1`, `H_2`, `S`, `R_1`, `R_2`)` ist erfolgreich, wenn es eine natürliche Zahl n gibt, sodaß es möglich ist, n Atome aus der Liste H_1 mit n Atomen aus der Liste H_2 zu unifizieren. In diesem Fall wird S mit der Liste der unifizierten Atome unifiziert. R_1 bzw. R_2 werden mit den Listen der Atome aus H_1 bzw. H_2 unifiziert, die nicht mit Atomen des anderen Kopfs unifiziert worden sind. `unify_heads` ist wiedererfüllbar.

```
unify_heads(H1, H2, S, R1, R2) :-
    length(H1, K),
    length(H2, L),
    min(K, L, M),
    enumeration(1, M, E),
    member(N, E),
    unify_heads(N, H1, H2, S, R1, R2).

unify_heads(0, H1, H2, [], H1, H2) :- !.

unify_heads(M, H1, H2, [A | S], R1, R2) :-
    delete(A, H1, H3),
    delete(A, H2, H4),
    N is M - 1,
    unify_heads(N, H3, H4, S, R1, R2).
```

3.8 Die Zustandsrepräsentation

Die Zustandsrepräsentation der Implementation orientiert sich an derjenigen der operationalen Semantik. Ein Zustand wird mit einem Fakt `state/4` 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.

3.9 Die Darstellung von Gleichheitsconstraints

Der vorliegende Konfluenz-Tester stützt sich aus historischen Gründen auf die Unifikation von Prolog. Deswegen werden Gleichheitsconstraints nie explizit, sondern verschwinden als Variablenbindungen im Prolog-System und werden automatisch propagiert. Das hat folgende Auswirkungen:

- Eine Implementation von `update` kann sich das Propagieren von Gleichheiten und das Entfernen von Gleichheitsconstraints für lokale Variablen sparen. Es kann nicht verhindert werden, daß neue Bindungen in die Liste der globalen Variablen propagiert werden. Das ist sogar notwendig, denn Ergebnisse für globale Variablen, die in keinem Constraint mehr vorkommen, können nur in der Liste der globalen Variablen explizit werden.

- Eine Implementation von $\rightarrow_o$ muß berücksichtigen, daß `update` neue Bindungen erzeugen kann, die automatisch propagiert werden. Das ist ein Problem, wie im folgenden Beispiel deutlich wird:

  Sei $S = \langle [], p(X), [\mathbf{true}], [X] \rangle$ und $T = \langle [], p(X), [X \doteq 5], [X] \rangle$. $\doteq$ meint hier die in Prolog eingebaute Unifikation. $S \not\rightarrow_o T$, denn T enthält mit $X = 5$ mehr Information als S. Eine naive Implementation von $\rightarrow_o$ würde entscheiden, daß $S \rightarrow_o T$, denn durch die Anwendung von `update` auf T entstünde aus S einer neuer Zustand $S' = \langle [], [p(5)], [\mathbf{true}], [5] \rangle$ und $S' = \texttt{update}(T) = \langle [], [p(5)], [\mathbf{true}], [5] \rangle$.

Beachtet werden muß außerdem, daß neue Bindungen auch auf Variablen übertragen werden, die mit den Variablen der Zustände in Verbindung stehen. Das ist ein gefährlicher Seiteneffekt.

Für eine korrekte Implementation von $\rightarrow_o$ ohne Seiteneffekte ist es daher notwendig, von beiden Zuständen zunächst Varianten mit frischen Variablen zu bilden. Der abschließenden Vergleich muß durch einen Äquivalenztest ersetzt werden.

- Eine Implementation von $\downarrow$ 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 Implementation von $\downarrow$ ohne die oben erwähnten Seiteneffekte ist es daher notwendig, von beiden Zuständen zunächst Varianten mit frischen Variablen zu bilden.

- Für eine Implementation von $\cong$ ergeben sich keine Probleme: Ergebnisse für globale Variablen werden nicht über Gleichheitsconstraints sondern über die Liste der globalen Variablen verglichen. Allerdings muß der abschließende Test auf Gleichheit durch einen Varianztest ersetzt werden, da eine Implementation von $\downarrow$ mit Varianten der Ausgangszustände arbeiten muß.

- In **Simplify** und **Propagate** gibt es Anwendungen von $\rightarrow_o$, die das Vorhandensein von Gleichheitsconstraints voraussetzen. Dort heißt es, daß eine Regel R mit Kopf H, Wächter G und Rumpf B nur dann auf einen Zustand $S = \langle Gs, C_U, C_B, \mathcal{V} \rangle$ angewendet werden kann, wenn es eine Umordnung $H' \wedge C'_U$ von C_U gibt, sodaß

$$\langle H', \Box, C_B, [] \rangle \rightarrow_o \langle H', \Box, H' \doteq H \wedge G, [] \rangle$$

gilt. An dieser Stelle ist es hilfreich, sich daran zu erinnern, was diese Bedingung meint: Die Instanz einer Regel kann nur dann auf einem Zustand feuern, wenn die Instanz durch Angleichen des Kopfes der Regel an benutzerdefinierte Constraints des Zustands gewonnen wurde, und der Wächter der Regelinstanz aus dem Zustand folgt, der Wächter also im betrachteten Zustand erfüllt ist und keine zusätzliche Information trägt. Die Entscheidung der Bedingung kann also in folgende Operationen zerlegt werden:

1. Finden einer Umordnung $H' \wedge C'_U$ von C_U, sodaß H an H' angeglichen werden kann. Durch das Angleichen entsteht eine Regelinstanz $\tilde{R}$ mit Kopf $\tilde{H} = H'$, Wächter $\tilde{G}$ und Rumpf $\tilde{B}$, da Prolog neue Variablenbindungen automatisch propagiert.
2. Testen, ob $\tilde{G}$ aus C_B folgt. Hier kann $\rightarrow_o$ bemüht werden; $\tilde{G}$ folgt aus C_B wenn

$$\langle H', \Box, C_B, \mathcal{V} \rangle \rightarrow_o \langle H', \Box, \tilde{G}, \mathcal{V} \rangle.$$

3.10 Die Entscheidung partieller Konfluenz

`partial` entscheidet die partielle Konfluenz für ein 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 `introduce`, `solve`, `idempotence` und `unfold`, die die in der operationalen Semantik definierten Zustandsübergänge 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.

Unter Benutzung von `feasible` wird festgestellt, ob das kritische Paar in der Realität auftreten kann. Vordefinierte Constraints, die der dazu verwendete Prolog-Metainterpreter verzögert, werden dem Anwender zur Vereinfachung vorgelegt. Tritt das kritische Paar in der Realität nicht auf, dann erfolgt Rücksetzen und das nächste kritische Paar wird untersucht.

Dann wird festgestellt, ob das kritische Paar zusammenführbar ist. Dazu wird das kritische Paar in zwei Zustände übersetzt und `joinable` aufgerufen. Ist das kritische Paar zusammenführbar, dann erfolgt Rücksetzen und das nächste kritische Paar wird untersucht. Ein nicht zusammenführbares kritisches Paar wird mit `display_critical_pair` angezeigt.

Die Anwendung von `fail` implementiert eine failure driven loop, sodaß alle kritischen Paare untersucht werden.

Anzumerken bleibt, daß `partial` auch ohne nähere Spezifikation eines Constraints benutzt werden kann, sodaß alle kritischen Paare untersucht werden. Dies macht sich `confluent` zunutze.

```
partial(Name/Arity) :-

    % get a critical pair for the specified constraint
    critical_pair(Name1, Name2, Shared_atoms,
                  Guard, Left, Right, Global_vars),
    for_one(name_and_arity(Name/Arity), Shared_atoms),

    % check whether the critical pair is feasible
    feasible(Guard, Delayed),

    % proove that the critical pair is not joinable
    not joinable(state(Left,  [], Delayed, Global_vars),
                 state(Right, [], Delayed, Global_vars)),

    % display the critical pair
    display_critical_pair(Name1, Name2,
                          Delayed, Left, Right, Global_vars),

    % examine next critical pair
    fail.
```

```
partial(_).


feasible(Guard, User_delayed) :-
    update(state([], [], Guard, []), state([], [], Delayed, [])),
    (   consult_user
    ->  consult_user(Delayed, User_delayed)
    ;   User_delayed = Delayed
    ),
    User_delayed \== [false].
```

3.11 Die Entscheidung der Zusammenführbarkeit

`joinable(`S`, `T`)` ist genau dann erfolgreich, wenn $S \downarrow T$. S und T sind trivialerweise zusammenführbar, wenn sie syntaktisch gleich sind. Diesen Fall erledigt die erste Klausel. [4] Im anderen Fall ist es nach Bemerkung 2.2.18 ausreichend, endliche Berechnungen $S \rightarrow_P^* S'$ und $T \rightarrow_P^* T'$ mit $S' \cong T'$ zu finden. Wie in 3.9 ausgeführt wurde, ist es für eine korrekte Implementation von $\downarrow$ ohne Seiteneffekte 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 `equivalent_states` verglichen. Sind die Ergebnisse nicht äquivalent, dann erfolgt Rücksetzen nach `expand`, sodaß 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`.

Zu beachten ist, daß `joinable` nur für propagationsfreie Programme zuverlässig funktioniert. Wie schon in 2.2.18 angemerkt wurde, arbeitet das gewählte Verfahren zusammen mit einer Instanzbedingung nicht zuverlässig. Zusammenführbare kritische Paare werden eventuell nicht als zusammenführbar erkannt.

```
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),
    equivalent_states(W, X),

    !.
```

[4] Die erste Klausel ist nicht notwendig, erhöht aber die Effizienz. Um die Häufigkeit dieses Falls zu erhöhen, werden bei der Bildung kritischer Paare (3.7) die linke und die rechte Seite sortiert.

3.12 Die Erzeugung von Berechnungen

`expand(`S`, `T`)` erzeugt für den Zustand S eine endliche Berechnung $S \rightarrow_P^* T$. Die erste Klausel unternimmt die folgenden Schritte:

1. Mit `introduce` werden *alle* möglichen **Introduce**-Schritte durchgeführt.
2. Mit `idempotence` werden, wenn vom Anwender gewünscht [5], *alle* möglichen **Idempotence**-Schritte durchgeführt.
3. Mit `solve` werden *alle* möglichen **Solve**-Schritte durchgeführt.
4. Durch Aufruf der zweiten Klausel wird die Regelanwendung angestoßen.

Zur Implementierung der Instanzbedingung ohne Indizierung führt `expand` eine Liste *History* ein, in der die schon angewendeten Instanzen von Propagationsregeln vermerkt werden. Diese Liste wird von `unfold` vor Regelanwendung konsultiert und nach Anwendung einer Propagationsregel erweitert. Die Liste wird durch Rücksetzen abgebaut. Nach Regelanwendung durch `unfold` wendet die zweite Klausel wiederum `introduce`, `idempotence` und `solve` an ruft sich dann rekursiv auf. `expand` ist wiedererfüllbar, da `unfold` wiedererfüllbar ist.

`introduce`, `idempotence` und `solve` scheitern niemals. `unfold` scheitert, wenn keine Regel mehr angewendet werden kann. Die Implementation der erfolgreichen Terminierung von `expand` für diesen Fall ist etwas knifflig. Eine Möglichkeit wäre, eine dritte Klausel

```
expand(S, S, _).
```

anzufügen. Diese Lösung hat den großen Nachteil, daß es für jeden Aufruf von `expand` einen zweiten Auswahlpunkt gibt, daß also mit Rücksetzen nicht nur Endzustände, sondern auch alle Zwischenzustände als Ergebnis ermittelt werden.

Eine zweite Möglichkeit wäre, eine dritte Klausel

```
expand(S, S, History) :- not unfold(S, _, History, _).
```

anzufügen, die nur für Zustände, auf die keine Regel des Programms mehr angewendet werden kann, erfolgreich ist. Diese Lösung ist teuer, da sie für einen Endzustand alle Regeln erfolglos probiert und für andere Zustände einige Regeln erfolglos probiert und dann eine Regel anwendet.

Ein eleganter Ausweg ist die Anwendung des *Soft Cut-Operators* `(*->)/2`. Der Ausdruck A `*->` B`;` C bedeutet: Wenn A bewiesen werden kann, dann beweise B, behalte aber alle Auswahlpunkte für A, sodaß durch Rücksetzen andere Beweise für A gefunden werden können. Wenn A scheitert, dann beweise C ([ECR95] , Abschnitt 5.10.1, S. 32). Rücksetzen in A erfolgt dann, wenn B scheitert oder wenn durch eine failure driven loop, wie sie `joinable` implementiert, weitere Beweise angefordert werden. `(*->)/2` verhält sich ähnlich wie `(->)/2`, mit dem Unterschied, daß `(->)/2` alle Auswahlpunkte für A entfernt.

Zur Verbesserung der Effizienz verkleinert die vorliegende Implementation den Suchraum, indem vor Anwendung von **Simplify** oder **Propagate** zuerst *alle* möglichen **Introduce**-Schritte und dann *alle* möglichen **Solve**-Schritte durchgeführt werden. Dieses Vorgehen ist unproblematisch:

[5] Siehe `idempotence_on` und `idempotence_off` in Abschnitt 3.3.

- **Simplify**, **Propagate** und **Idempotence** können erst dann auf ein neues Constraint angewendet werden, wenn es von **Introduce** vom Zielspeicher in den Speicher für benutzerdefinierte Constraints transportiert wurde.
- Nach einem **Solve**-Schritt ist jeder **Introduce**-Schritt möglich und umgekehrt.
- Nach Anwendung von **Solve** ist jede Anwendung von **Simplify**, **Propagate** oder **Idempotence** möglich, die auch schon vor Anwendung von **Solve** möglich war.

```
expand(S, T) :-
    introduce(S, A),
    idempotence(A, B),
    solve(B, C),
    expand(C, T, []).

expand(S, T, History) :-
    unfold(S, A, History, New_history),
    introduce(A, B),
    idempotence(B, C),
    solve(C, D),
    expand(D, T, New_history)
    *-> true
    ;   T = S.
```

3.13 Die Zustandsübergänge

`introduce(`S`, `T`)` führt für den Zustand S alle möglichen **Introduce**-Schritte durch und unifiziert T mit dem Ergebnis der Berechnung. Wenn **Introduce** auf S nicht anwendbar ist, dann gilt $T = S$.

```
introduce(S, T) :-
    S = state(Goals0, Userdef0, Builtin, Global_vars),
    filter(is_userdef_constraint, Goals0, Userdef1, Goals1),
    append(Userdef0, Userdef1, Userdef2),
    T = state(Goals1, Userdef2, Builtin, Global_vars).
```

`solve(`S`, `T`)` führt für den Zustand S alle möglichen **Solve**-Schritte durch und unifiziert T mit dem Ergebnis der Berechnung. Wenn **Solve** auf S nicht anwendbar ist, dann gilt $T = S$.

`solve` kann davon ausgehen, daß sich im Zielspeicher von S keine benutzerdefinierten Constraints mehr befinden, da in diesem Kontext S das Ergebnis von `introduce` ist.

```
solve(S, T) :-
    S = state(Goals0, Userdef, Builtin0, Global_vars),
    append(Goals0, Builtin0, Builtin1),
    update(state([], Userdef, Builtin1, Global_vars), T).
```

`idempotence(`S`, `T`)` führt, wenn ein Fakt `use_idempotence` vorhanden ist, für den Zustand S alle möglichen **Idempotence**-Schritte durch und unifiziert T mit dem Ergebnis der Berechnung. Wenn kein Fakt `use_idempotence` vorhanden ist oder **Idempotence** auf S nicht anwendbar ist, dann gilt $T = S$.

```
idempotence(S, T) :-
    use_idempotence,
    !,
    S = state(Goals, Userdef0, Builtin, Global_vars),
    sort(Userdef0, Userdef1),
    T = state(Goals, Userdef1, Builtin, Global_vars).

idempotence(S, S).
```

`unfold` implementiert **Simplify** und **Propagate**. `unfold(`S`, `T`, `H`, `H'`)` wendet die Instanz einer Simplifikations- oder Propagationsregel auf den Zustand S an und unifiziert T mit dem resultierenden Zustand. `unfold` scheitert, wenn keine Regel auf S angewendet werden kann. Die Instanz einer Propagationsregel kann nur dann angewendet werden, wenn sie noch nicht in H verzeichnet ist. H' wird mit der neuen Liste der angewendeten Instanzen von Propagationsregeln unifiziert. Wurde eine Simplifikationsregel angewendet, dann gilt $H' = H$.

Die erste Klausel von `unfold` scheitert, wenn der Speicher für vordefinierte Constraints von S gleich **false** ist. Die zweite Klausel holt eine Variante einer Regel mit frischen Variablen und versucht, die Kopfatome mit benutzerdefinierten Constraints aus S mit `matches` abzugleichen. Wenn das gelingt, wird für die Instanz einer Propagationsregel zur Verhinderung trivialer Nicht-Terminierung sichergestellt, daß sie noch nie gefeuert hat. Zur Implementierung der Instanzbedingung ohne Indizierung wird ein Liste verwendet, in der die Instanzen vermerkt werden, die schon zur Anwendung gekommen sind. Zur Motivation des verwendeten Formats für Listeneinträge sollen naheliegende Formate diskutiert werden.

Jeder Eintrag muß einen eindeutigen Schlüssel enthalten, der die Regel identifiziert, auf die sich der Eintrag bezieht. Da die Syntax von CHR für Regeln keine eindeutigen Namen verlangt, müssen die Regeln selbst als Schlüssel verwendet werden. Da Prolog als Ergebnisse von Datenbankanfragen immer Varianten mit frischen Variablen liefert, läuft es darauf hinaus, daß Varianten der Regeln als Schlüssel verwendet müssen. Es ergibt sich das Format (R, H'), wobei R die Variante einer Regel und H' der Kopf der Instanz von R ist, die gefeuert hat. Denkbar ist auch das Format (R, L), wobei L eine Liste der Köpfe der Instanzen von R ist, die gefeuert haben. Hier ist die Implementation aber deutlich aufwendiger. Beide Formate haben den Nachteil, daß der Schlüsselvergleich aufwendig ist.

Setzt man eindeutige Regelnamen voraus, dann ist es möglich, die Anwendung einer Propagationsregel durch einen Eintrag der Form (N, H) zu vermerken, wobei N der Name und H der Kopf der Regelinstanz ist, die gefeuert hat. Dieses Format hat den Vorteil, daß für den Schlüsselvergleich `==` verwendet werden kann, wie es auch für den Vergleich von Köpfen von Regelinstanzen verwendet werden muß.

Für die Implementation wurde die letztere Darstellung gewählt, da der syntaktische Vergleich von Atomen billiger ist als die Anwendung von `variant` auf Regelvarianten. Als weiterer Vorteil ergibt sich eine übersichtliche Darstellung beim Mitschreiben. Die Vergabe eindeutiger Regelnamen wurde dem CHR-Programm-Lader überantwortet.

Ist die Anwendbarkeit einer Regelinstanz nicht eingeschränkt, und ist der Wächter der Regelinstanz im durch S gegebenen Kontext erfüllt, was mit `entails` getestet wird, dann wird die Regelinstanz auf S angewendet.

```
unfold(state(_, _, [false], _), _, _, _) :- !, fail.
```

```
unfold(S, T, History, New_history) :-

    % split initial state
    S = state([], Userdef, Builtin, Global_vars),

    % rule selection and matching
    rule(_, Normalized_rule, Name, Head, Guard, Body, Surviving_atoms),
    matches(Head, Userdef, Unmatched_atoms),

    % avoid trivial non-termination for propagation rules
    (   Normalized_rule = (_ @ _ ==> _)
    ->  not member(==, (Name, Head), History)
    ;   true
    ),

    % guard check
    entails(state(Head, [], Builtin, Global_vars),
            state(Head, [], Guard,   Global_vars)),

    % fire
    append(Unmatched_atoms, Surviving_atoms, Userdef1),
    T = state(Body, Userdef1, Builtin, Global_vars),

    % record application of instance of propagation rule
    (Normalized_rule = (_ @ _ ==> _)
    ->  New_history = [(Name, Head) | History]
    ;   New_history = History
    ).
```

`matches(`K`, `L`, `R`)` gleicht die Atome einer Liste K nichtdeterministisch an Atome einer Liste L an und unifiziert R mit der Liste der Atome aus L, an die kein Atom aus K angeglichen worden ist. `matches` scheitert, wenn K mehr Atome als L enthält oder es keine weitere Möglichkeit gibt, Atome aus K an Atome aus L anzugleichen.

`matches` benutzt `select` zur nichtdeterministischen Auswahl von Atomen aus L. Da Angleichen einer einseitigen Unifikation entspricht, kann die Unifikation zum Angleichen benutzt werden, wenn nur zuvor sichergestellt wird, daß die Unifikation die Atome aus L unverändert belassen wird. Für diesen Test wird `instance` benutzt.

Anzumerken bleibt, daß das Angleichen *simultan* erfolgen muß. Werden die Atome einzeln angeglichen, kann es zu Bindungen zwischen den Variablen in L kommen, was nicht erlaubt ist.

```
matches(K, L, R) :-
    length(K, M),
    length(L, N),
    M =< N,
    select(L, M, S, R),
    instance(S, K),
    K = S.
```

3.14 Die Entscheidung des Entailments

`entails` implementiert $\rightarrow_o$. `entails(`S`, `T`)` ist genau dann erfolgreich, wenn $S \rightarrow_o T$.

Wie in 3.9 ausgeführt wurde, ist es für eine korrekte Implementation von $\rightarrow_o$ ohne Seiteneffekte notwendig, von beiden Zuständen zunächst Varianten mit frischen Variablen zu bilden. Das weitere Vorgehen folgt der Definition.

```
entails(S, T) :-

    % avoid side effects
    copy_term(S, W),
    copy_term(S-T, U-V),

    % decide entailment
    update(W, X),
    U = state(_, _, U_builtin, _),
    V = state(V_goals, V_userdef, V_builtin, V_global_vars),
    append(U_builtin, V_builtin, Builtin),
    update(state(V_goals, V_userdef, Builtin, V_global_vars), Y, Loginfo),
    equivalent_states(X, Y).
```

3.15 Die Update-Funktion

`update(`S`, `T`)` vereinfacht und normalisiert den Speicher für vordefinierte Constraints des Zustands S und unifiziert T mit dem resultierenden Zustand.

Der Speicher für vordefinierte Constraints von S wird mit `metacall` normalisiert. `metacall` implementiert einen Prolog-Metainterpreter (siehe Abschnitt 4).

```
update(S, T) :-
    S = state(Goals, Userdef, Builtin0, Global_vars),
    (metacall(Builtin1, Delayed) -> Builtin1 = Delayed; Builtin1 = [false]),
    T = state(Goals, Userdef, Builtin1, Global_vars).
```

3.16 Die Entscheidung der Äquivalenz

Die Definition von $\cong$ verlangt das Bilden von Umordnungen für die Speicher von einem der betrachteten Zustände. Für eine Implementation bedeutet das ein Effizienzproblem.

Eine Möglichkeit der Optimierung ist das Sortieren der Constraintspeicher vor dem Varianztest. Beim Sortieren ist jedoch zu beachten, daß die Variablennamen für die Sortierordnung relevant sind. Wenn die zu vergleichenden Zustände alle Variablen teilen, dann ist ein Sortieren der Constraintspeicher vor dem Varianztest tatsächlich ausreichend, das Bilden von Umordnungen kann entfallen. Wenn die zu vergleichenden Zustände nicht alle Variablen teilen, dann ist das Sortieren der Constraintsspeicher wegen der Abhängigkeit der Sortierordnung von den Variablennamen kein zuverlässiger Ersatz für das Bilden von Umordnungen; äquivalente Zustände würden nicht immer als äquivalent erkannt werden.

Die Entscheidung der Äquivalenz von Zuständen wird in `joinable` und `entails` benötigt. Dort haben die zu vergleichenden Zustände keine Variablen gemeinsam. Zwar teilen die

Zustände, für die `joinable` die Zusammenführbarkeit entscheiden soll, die globalen Variablen, doch `joinable` muß von den Zuständen Varianten mit frischen Variablen bilden. Auch `entails` muß von den Argumenten Varianten mit frischen Variablen bilden. Da man keinen Einfluß darauf hat, welche Namen Prolog für frische Variablen beim Kopieren von Termen mit `copy_term` vergibt, ist das Ergebnis einer Sortierung nicht vorhersehbar. Es gibt aber einen Ausweg: Stellt man zunächst fest, daß die Listen der globalen Variablen Varianten sind, dann können die globalen Variablen durch Unifikation identifiziert werden, sodaß die zu vergleichenden Zustände die globalen Variablen teilen und somit das Sortieren der Constraintspeicher ein zuverlässiger Ersatz für das Bilden von Umordnungen ist.

Als letztes Problem bleiben die lokalen Variablen. Diese können schon aus den Regeln, aus denen das kritische Paar hervorgegangen ist, stammen, oder durch Regelanwendung eingeführt worden sein. Da man keinen Einfluß darauf hat, welche Namen Prolog für frische Variablen beim Holen einer Regel aus der Regel-Datenbank vergibt, ist das Ergebnis einer Sortierung nicht vorhersehbar. Hier bleibt nur noch das Bilden der Umordnungen.

Da es wichtig ist, das Bilden von Umordnungen auf ein Minimum zu reduzieren, ist es naheliegend, für jeden Constraintsspeicher eine Aufteilung in zwei Speicher vorzunehmen: einen für die Constraints, in denen alle Variablen global sind, und einen zweiten Speicher für die Constraints, in denen mindestens eine lokale Variable vorkommt. Für die Speicher, in denen dann nur globale Variablen vorkommen, ist, wie gezeigt wurde, das Bilden von Umordnungen nicht notwendig, die Sortierung ist ausreichend. Für die Speicher, in denen dann auch lokale Variablen vorkommen, müssen Umordnungen gebildet werden.

`equivalent_states(`S`, `T`)` ist genau dann erfolgreich, wenn $S \cong T$.

Die erste Klausel von `equivalent_states` behandelt den Fall, daß die Speicher für vordefinierte Constraints beider Zustände gleich **false** sind. Die zweite Klausel überprüft zunächst, ob die Ergebnisse für die globalen Variablen von S und T Varianten sind. In diesem Fall werden die globalen Variablen von S mit denen von T durch Unifikation identifiziert. Dann werden die Constraintspeicher mit `sort_state` sortiert und, wie oben beschrieben, mit `filter` zerlegt und die Teilspeicher, die nur globale Variablen enthalten, auf syntaktische Gleichheit überprüft. Von den Teilspeichern von T, die auch lokale Variablen enthalten, werden dann mit `select` nichtdeterministisch Umordnungen gebildet, die mit den entsprechenden Teilspeichern von S durch `variant` verglichen werden. Zu beachten ist, daß die Argumente für `variant` keine Variablen gemeinsam haben dürfen ([ECR94], S. 595). Deshalb muß vor Anwendung von `variant` von einem Argument mit `copy_term` eine Variante mit frischen Variablen erzeugt werden.

Es ist sinnvoll, das Sortieren vor dem Zerlegen zu erledigen. Da `filter` die Sortierordnung erhält, überträgt sich die Ordnung auch auf die Teilspeicher mit lokalen Variablen. Unter günstigen Umständen entfällt dadurch die Suche nach Umordnungen.

```
equivalent_states(state(_, _, [false], _), state(_, _, [false], _)) :-
    !.

equivalent_states(S, T) :-

    % avoid side effects
    copy_term(S, U),
    copy_term(T, V),
```

```
% compare results for global variables
U = state(_, _, _, U_global_vars),
V = state(_, _, _, V_global_vars),
variant(U_global_vars, V_global_vars),

% identify global variables of U and V
U_global_vars = V_global_vars,
Global_vars = U_global_vars,
    % variable lists are the same now, so no need for two names!

% sort constraint stores
sort_state(U, W),
sort_state(V, X),

% split up constraint stores
W = state(U_goals, U_userdef, U_builtin, _),
filter(has_local_variables(Global_vars),
       U_goals, U_goals_local, U_goals_global),
filter(has_local_variables(Global_vars),
       U_userdef, U_userdef_local, U_userdef_global),
filter(has_local_variables(Global_vars),
       U_builtin, U_builtin_local, U_builtin_global),
X = state(V_goals, V_userdef, V_builtin, _),
filter(has_local_variables(Global_vars),
       V_goals, V_goals_local, V_goals_global),
filter(has_local_variables(Global_vars),
       V_userdef, V_userdef_local, V_userdef_global),
filter(has_local_variables(Global_vars),
       V_builtin, V_builtin_local, V_builtin_global),

% compare constraints without local variables
state(U_goals_global, U_userdef_global,
      U_builtin_global, Global_vars) ==
state(V_goals_global, V_userdef_global,
      V_builtin_global, Global_vars),

% compare permutations of lists of constraints with local variables
copy_term((U_goals_local, U_userdef_local,
           U_builtin_local, Global_vars),
          (U_goals_local_copy, U_userdef_local_copy,
           U_builtin_local_copy, Global_vars_copy)),
    % arguments to variant/2 must not share variables!
length(U_goals_local,   M),
length(V_goals_local,   M),
select(V_goals_local,   M, V_goals_local_permutation, _),
variant((V_goals_local_permutation, Global_vars),
        (U_goals_local_copy, Global_vars_copy)),
```

```
    length(U_userdef_local, N),
    length(V_userdef_local, N),
    select(V_userdef_local, N, V_userdef_local_permutation, _),
    variant((V_userdef_local_permutation, Global_vars),
            (U_userdef_local_copy, Global_vars_copy)),
    length(U_builtin_local, O),
    length(V_builtin_local, O),
    select(V_builtin_local, O, V_builtin_local_permutation, _),
    variant((V_builtin_local_permutation, Global_vars),
            (U_builtin_local_copy, Global_vars_copy)),
    !.
```

3.17 Hilfsprädikate

`sort_state` unifiziert T mit einem Zustand, der aus dem Zustand S durch Sortieren der Constraintspeicher hervorgegangen ist. Zum Sortieren kann `sort` nicht benutzt werden, da `sort` auch Duplikate entfernt ([ECR94], S. 537). `msort` entfernt Duplikate nicht ([ECR94], S. 379).

```
sort_state(S, T) :-
    S = state(Goals0, Userdef0, Builtin0, Global_vars),
    msort(Goals0, Goals1),
    msort(Userdef0, Userdef1),
    msort(Builtin0, Builtin1),
    T = state(Goals1, Userdef1, Builtin1, Global_vars).
```

`?- select(`L`, `N`, `S`, `R`)` wählt nichtdeterministisch N Atome aus der Liste L aus und unifiziert S mit der Liste der ausgewählten Atome und R mit der Liste der nicht ausgewählten Atome. `select` ist wiedererfüllbar. Durch Rücksetzen können alle Permutationen für alle Auswahlen von N Elementen aus L erzeugt werden.

```
select([], N, [], []) :- !, N = 0.
select(L, 0, [], L) :- !.
select(L, N, [X | T], R) :-
    delete(X, L, L0),
    M is N - 1,
    select(L0, M, T, R).
```

`?- enumeration(`F`, `T`, `L`)` unifiziert L mit der Liste der ganzen Zahlen des Intervalls $[F, T]$.

```
enumeration(F, F, [F]) :- !.
enumeration(F, T, [F | Tail]) :-
    Z is F + 1, enumeration(Z, T, Tail).
```

`?- filter(`P`, `L`, `F`, `G`)` unifiziert F mit der Liste der Elemente der Liste L, für die P erfolgreich ist, und G mit der Liste der Elemente von L, für die P scheitert.

```
filter(P, L, F) :- filter(P, L, F, _).
filter(_, [], [], []) :- !.
filter(P, [E | T0], [E | T1], L) :-
    apply(P, [E]), !, filter(P, T0, T1, L).
filter(P, [E | T0], L, [E | T1]) :-
    filter(P, T0, L, T1).
```

`?- member(`P`, `X`, `L`)` ist erfolgreich, wenn es ein Y in der Liste L gibt, sodaß $P(X,Y)$ gilt. `member/3` ist wiedererfüllbar wie `member/2`.

```
member(P, X, [Y | _]) :- apply(P, [X, Y]).
member(P, X, [_ | T]) :- member(P, X, T).
```

`?- for_one(`P`, `L`)` ist genau dann erfolgreich, wenn es ein Element in der Liste L gibt, auf das P erfolgreich angewendet werden kann. `for_one` ist nicht wiedererfüllbar.

```
for_one(P, [H | _]) :- apply(P, [H]), !.
for_one(P, [_ | T]) :- for_one(P, T).
```

3.18 Plattform-Unabhängigkeit

Möglich ist die SWI-Prolog-Version, weil der Tester den CHR-Interpreter, welcher nur auf ECL^{i}PS^e läuft, nicht benötigt.

Da ECL^{i}PS^e zusammen mit dem CHR-Interpreter bezüglich des Funktionsumfangs eine Obermenge von SWI-Prolog ist, bedeutete die Anforderung der Plattform-Unabhängigkeit die Deklaration einiger Operatoren und Prädikate, die in SWI-Prolog nicht verfügbar sind: `(<=>)/2`, `(==>)/2`, `(@)/2`, `(\)/2`, `(constraints)/1`, `(handler)/1`, `(*->)/2`, `delete/3`, `flush/1`, `term_variables/2`, `variant/2`, und `instance/2`. Diese sind in der Datei `swisp.cpp` implementiert, um den Programmtext des eigentlichen Testers nicht durch Lösungen für plattform-spezifische Probleme unübersichtlich zu machen.

An wenigen Stellen war es notwendig, die Laufumgebung ermitteln zu können. Deswegen definieren `eclsp.cpp` und `swisp.cpp` je einen Fakt `platform/1`.

Generell wurden im Konfluenz-Tester nur solche Prädikate benutzt, die in ECL^{i}PS^e verfügbar sind.

3.19 Der Prototyp

Die Merkmale des Prototypen aus [Meu96], der dem hier dokumentierten Programm zugrundeliegt, sind:

- Der Prototyp verarbeitet propagationsfreie CHR-Programme, deren Regeln nur ein Atom im Kopf haben und sich nicht auf vordefinierte Prolog-Prädikate stützen.

- Ein benutzerdefiniertes Constraint kann auf partielle und totale Konfluenz getestet werden. Für die Suche nach äquivalenten Endzuständen wird ein vom CHR-Interpreter unabhängiger Expansionsalgorithmus eingesetzt. Der Algorithmus terminiert, wenn

  - der Speicher für eingebaute Constraints inkonsistent ist oder
  - keine Regel mehr anwendbar ist. [6]

- Nach der Bildung eines kritischen Paares erhält der Benutzer Gelegenheit den Wächter des kritischen Paares im Dialog zu vereinfachen.
- Der Prototyp ist in Prolog geschrieben und läuft unter ECL^{i}PS^e.

[6] Eine Anwendungsbedingung kann entfallen, da nur Simplifikationsregeln zulässig sind.

Kapitel 4

Die Behandlung von Prolog-Prädikaten

4.1 Einleitung

Vorliegende Constraintsolver (z.B. Eclipse-CHR-Library) machen regen Gebrauch von vordefinierten und benutzerdefinierten Prolog-Prädikaten. Das erspart die Implementation der benötigten Funktionalität in CHR. Um auch diese Solver einem Konfluenz-Test zugänglich zu machen, müssen Prolog-Prädikate behandelt werden. Wie die unten stehenden Beispiele und die darauf folgende Diskussion zeigen, ist es jedoch nicht möglich, Prolog-Prädikate direkt von Prolog auswerten zu lassen; zur Sicherstellung von Korrektheit und Terminierung ist es notwendig, eine Metainstanz einzuschalten. Dieser Abschnitt stellt einen Metainterpreter vor, der die folgenden Operatoren und Prädikate behandelt:

- Negation der Unifizierbarkeit (`\=`)
- Syntaktische Vergleiche (`==`, `\==`, `@>`, `@>=`, `@<`, `@=<`) und deren Negation
- Arithmetische Operatoren (`is`, `=:=`, `=\=`, `>`, `>=`, `<`, `=<`) und deren Negation
- Den Cut (`!`)
- If-Then und If-Then-Else (`->`)
- Die Disjunktion (`;`)
- Typtest-Prädikate (`var`, `ground`, `integer`, `number`, usw.) und deren Negation
- Prädikate zur Listenverarbeitung (`append`, `delete`, `member`, usw.) und deren Negation
- Benutzerdefinierte Prädikate und deren Negation

Zur Notation in diesem Abschnitt sei auf Abschnitt 3.4 verwiesen.

4.2 Motivation

Vordefinierte und benutzerdefinierte Prolog-Prädikate werden in dieser Arbeit als vordefinierte Constraintsymbole betrachtet, sodaß die Auswertung und Vereinfachung von Prolog-Zielen in den Aufgabenbereich der Implementation einer update-Funktion fallen. Für eine Implementation ist die Verwendung des Prolog-Prädikats `call`, das ein Prolog-Ziel auswertet, naheliegend:

```
update(S, T) :-
    S = state(Goals, Userdef, Builtin0, Global_vars),
    (call(Builtin0) -> Builtin1 = []; Builtin1 = [false]),
    T = state(Goals, Userdef, Builtin1, Global_vars).
```

`call` ist wiedererfüllbar, wenn das Prolog-Ziel wiedererfüllbar ist. `update` darf aber nicht wiedererfüllbar sein, da CHR Rücksetzen nicht kennt. Es muß das erste von Prolog ermittelte Ergebnis verwendet werden. Die Verwendung von `->` stellt das schon sicher. `update` implementiert jedoch keine update-Funktion, da es nicht total und nicht korrekt ist, wie die folgenden Beispiele zeigen:

Beispiel 4.2.1 Die Anfrage

```
    ?- update(state([], [], [1 < Z], [Z]), _).
```

führt zu einer Fehlermeldung des Prolog-Systems, da `1 < Z` nur dann von Prolog ausgewertet werden, wenn beide Argumente für `<` Zahlen sind. Ähnliche Beispiele können auch für `>`, `>=`, `=<`, `is`, usw. konstruiert werden.

Beispiel 4.2.2 Vorausgesetzt sei die folgende Implementation des Prädikats `checklist`:

```
checklist(P, [A|B]) :- checklist(B), apply(P, [A]).
checklist(P, []).
```

Dann terminiert Prolog für die Anfrage

```
    ?- update(state([], [], [checklist(integer, X)], [X]), _).
```

nicht.

Beispiel 4.2.3 Vorausgesetzt sei die übliche Implementation des Prädikats `member`:

```
member(A, [A|B]).
member(A, [B|C]) :- member(A, C).
```

Dann ergibt die Anfrage

```
    ?- update(state([], [], [member(a, L)], [L]), R).
```

für `R` den Zustand `state([], [], [], [[a | L2]])` mit einer neuen, lokalen Variablen `L2` durch Anwendung der ersten Regel von `member`. Das ist nicht korrekt, denn die Substitution $\{$`L/[a | L2]`$\}$ ist nur ein Modell von unendlich vielen Modellen für `member(a, L)`.

Beispiel 4.2.4 Vorausgesetzt sei wieder die übliche Implementation des Prädikats `member`. Dann ergibt die Anfrage

```
?- update(state([], [], [member(X, [a, b, c])], [X]), R).
```

für `R` durch Anwendung der ersten Regel von `member` den Zustand `state([], [], [], [a])`. Das ist nicht korrekt, denn die Substitution {`X/a`} ist nur ein Modell von drei Modellen für `member(X, [a, b, c])`.

Die in den Beispielen aufgezeigten Probleme sind praktisch relevant, denn Prolog-Ziele werden im Rahmen eines Konfluenztests in einer anderen Umgebung ausgewertet als in realen Anwendungen des betrachteten Constraintsolvers: In den vom Konfluenztest durch Bildung kritischer Paare erzeugten Umgebungen sind meistens deutlich weniger Variablen instantiiert als in den Umgebungen realer Anwendungen, die durch die Benutzeranfragen bestimmt werden. Im Rahmen eines Konfluenztests müssen Prolog-Prädikate also in einer „instanziierungsarmen" Umgebung ausgewertet werden, wobei davon ausgegangen werden muß, daß eine Variable für einen beliebigen Term steht.

Die Beispiele und die vorangegangene Erläuterung machen deutlich, daß die Auswertung von Prolog-Prädikaten Prolog allein nicht überlassen werden kann. Vielmehr wird eine Metainstanz benötigt, die Prädikate nur dann an Prolog übergibt, wenn sichergestellt ist, daß die Auswertung mit einem sinnvollen Ergebnis terminieren wird. In anderen Fällen muß die Metainstanz versuchen, das Prädikat selbst auszuwerten. Ist eine Auswertung nicht sinnvoll oder nicht möglich, dann muß die Auswertung des Prädikats verzögert werden.

4.3 Problemanalyse und Lösungsansatz

Bei der Auswertung eines Prädikats P durch Prolog sind die folgenden Probleme zu berücksichtigen:

1. Wenn P ein vordefiniertes arithmetisches Prädikat ist (z.B. `is`, `=:=`, `<`, `=<`), dann bricht Prolog mit einem Fehler ab, wenn in den Argumenten des Prädikats Variablen vorkommen.

2. Wenn P ein vordefiniertes Prädikat ist (z.B. `ground`, `var`, `integer`, `\=`, `==`, `\==`, `not`, `->`, `@<`, `@=<`, `append`, `delete`, ...) ist, dann liefert die Auswertung durch Prolog in vielen Fällen ein falsches Ergebnis, da Prolog nicht beachtet, daß eine Variable für einen beliebigen Term steht.

3. Wenn es mehrere Modelle für ein benutzerdefiniertes Prädikat P gibt, dann wählt das obige `update` das erste aus. Weitere Modelle können nicht berücksichtigt werden, weil CHR Rücksetzen nicht erlaubt. Die Auswahl des ersten Modells ist dann die unzulässige Annahme, daß P nur durch das erste Modell erfüllt werden kann. Determinismus und Korrektheit hängen also eng zusammen.

4. Wenn P ein rekursiv definiertes Prädikat ist, dann terminiert die Auswertung von P unter Umständen nicht.

Das obige `update` ist nicht total, weil es die Punkte eins und vier nicht berücksichtigt. Es ist nicht korrekt, weil es die Punkte zwei und drei nicht berücksichtigt.

Da die Eigenschaften vordefinierter Prädikate bekannt sind, ist es nicht schwierig, die Bedingungen zu ermitteln, unter denen ein vordefiniertes Prädikat an Prolog übergeben werden darf bzw. verzögert werden muß. Diese Bedingungen sehen für jedes vordefinierte Prädikat anders aus und können fest in ein `update` eingebaut werden. Die Eigenschaften benutzerdefinierter Prädikate hingegen sind nicht bekannt. Korrektheit und Terminierung müssen hier durch Anwendung geeignet zu wählender Heuristiken gewährleistet werden.

Ein vordefiniertes arithmetisches Prädikat kann nur dann an Prolog übergeben werden, wenn in den Argumenten keine Variablen vorkommen. Unter Punkt zwei aufgeführte Prädikate können nicht von Prolog ausgewertet werden. Für die Details sei auf die Bemerkungen zur Implementation verwiesen. Für die Auswertung eines benutzerdefinierten Prädikats P sind die folgenden Fälle zu unterscheiden:

1. Wenn es kein Modell für P gibt, dann kann P durch **false** ersetzt werden.
2. Wenn es genau ein Modell für P gibt, dann kann P durch eine Konjunktion von Gleichheitsconstraints für die Variablen in P ersetzt werden. Praktisch ist das zwar nicht möglich, da Gleichheitsconstraints nicht explizit werden, aber auch nicht notwendig, da Prolog die Bindungen automatisch propagiert.
3. Wenn es endlich viele Modelle für P gibt, dann kann P durch eine Disjunktion von Konjunktionen von Gleichheitsconstraints für die Variablen in P ersetzt werden. Praktisch ist das aber nicht möglich, da Gleichheitsconstraints nicht explizit werden.
4. Wenn es unendliche viele Modelle für P gibt, dann kann P nicht ausgewertet werden, da Constraintspeicher endlich sind.

Das Grundproblem besteht also darin, zu entscheiden, ob es für P kein, ein oder mehr als ein Modell gibt. Wenn es mehr als ein Modell gibt, dann kann P nicht ausgewertet werden. Ein naheliegender Ansatz besteht darin, Modelle zu berechnen und zu zählen. Für die vorliegende Arbeit wurde jedoch ein anderer Weg gewählt. Verwendung findet die

Heuristik 4.3.1 Wenn die Anzahl der auf ein benutzerdefiniertes Prädikat P anwendbaren Regeln größer als eins ist, dann gibt es mehr als ein Modell für P.

Diese Heuristik ist offensichtlich zu stark, da sie nicht berücksichtigt, daß Regeln scheitern können. Praktisch hat sie sich aber bewährt. Zu beachten ist weiters, daß die Heuristik keine Aussage über die Anzahl der Modelle für den Fall macht, daß genau eine Regel auf P anwendbar ist. In diesem Fall sind mehrere Modelle möglich, wenn für ein Subziel der Auswertung mehrere Modelle möglich sind. Die Heuristik muß also auch auf jedes benutzerdefinierte Prädikat, das als Subziel bewiesen werden muß, angewendet werden.

Interessanterweise sichert die Anwendung der Heuristik in vielen Fällen auch die Terminierung. Wie das funktioniert, soll anhand eines Beispiels gezeigt werden. Vorausgesetzt sei ein für vordefinierte Prädikate korrektes `update`, das die Heuristik implementiert.

Beispiel 4.3.2 Sei `checklist` wie in Beispiel 4.2.2 definiert. Dann gilt:

- Die Anfrage

```
?- update(state([], [], [checklist(integer, [])], []), _).
```

ist erfolgreich mit der zweiten Regel.

- Die Anfrage

  ```
  ?- update(state([], [], [checklist(integer, X)], [X]), R).
  ```

  ergibt für `R` den Zustand `state([], [], [checklist(integer, X)], [X])`, da `checklist(integer, X)` mit den Köpfen beider Regeln unifiziert und deswegen nach der Heuristik verzögert wird.

- Die Anfrage

  ```
  ?- update(state([], [], [checklist(integer, [X | Y])], [X, Y]), R).
  ```

  ergibt für `R` den Zustand

  ```
  state([], [], [checklist(integer, Y), integer(X)], [X, Y]).
  ```

  Die Subziele `checklist(Y)` und `integer(X)` stammen aus dem Rumpf der ersten Regel. `checklist(Y)` wird nach der Heuristik verzögert, `integer(X)` kann wegen Informationsmangels nicht ausgewertet werden.

Wie das folgende Beispiel zeigt, ist die Heuristik in manchen Fällen aber auch zu schwach, um die Terminierung sicherzustellen.

Beispiel 4.3.3 Vorausgesetzt sei die übliche Implementation von `append`.

```
append([], X, X).
append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).
```

Angenommen, `append` ist benutzerdefiniert. Dann terminiert Prolog für die Anfrage

```
?- update(state([], [], [append([A | L], K, L)], [A, K, L]), _).
```

nicht, denn die Anwendung der zweiten Regel erzeugt das Subziel `append([A | L2], K, L2)` mit einer neuen, lokalen Variablen `L2`, das bis auf Variablenumbenennung mit dem ursprünglichen Ziel identisch ist. Die Heuristik greift hier nicht, weil immer nur die zweite Regel anwendbar ist.

Dieses Beispiel scheint unsinnig zu sein, doch es ist nicht auszuschließen, daß bei der Bildung eines kritischen Paares Variablen derart gebunden werden, daß Prädikate entstehen, deren Auswertung nicht terminiert. Ähnliche Beispiele können auch für `maplist` und `delete` konstruiert werden.

4.4 Die Behandlung des Cut

Ein korrektes `update` kann Vorkommen des Cut in Rümpfen von Prolog-Regeln ignorieren, denn wenn es mehrere Modelle für ein benutzerdefiniertes Prädikat gibt, dann wird es nicht ausgewertet. Wenn es nur ein Modell gibt, dann sind Vorkommen des Cut ohne Bedeutung.

4.5 Probleme einer festen Auswertungsreihenfolge

Eine Metainstanz, die wie Prolog ein Ziel von links nach rechts auswertet, ohne die Teilziele in Beziehung zu setzen, hat Probleme beim Erkennen von Widersprüchen, wie die folgenden Beispiele illustrieren. Vorausgesetzt sei ein korrektes `update`, das Teilziele nicht in Beziehung setzt.

Beispiel 4.5.1 Die Anfrage

```
?- update(state([], [], [var(X), nonvar(X)], [X]), R).
```

ergibt für `R` den Zustand `state([], [], [var(X), nonvar(X)], [X])`, da die Teilziele verzögert werden.

Beispiel 4.5.2 Die Anfrage

```
?- update(state([], [], [X < 1, X = 2], [X]), R).
```

ergibt für `R` den Zustand `state([], [], [2 < 1], [2])`, da `X < 1` verzögert wird. [1]

Problem 4.5.2 könnte durch eine Iteration gelöst werden: Die Metainstanz wird solange angewandt, bis sich das Ergebnis nicht mehr verändert. Das Problem 4.5.1 könnte durch gezieltes Vorausschauen gelöst werden: Bei der Verarbeitung von $\texttt{var}(X)$ wird in den verbleibenden Teilzielen nach $\texttt{nonvar}(X)$ gesucht. Diese „feste Verdrahtung" von Wissen über vordefinierte Constraints würde aber zu einem unüberschaubaren und schlecht wartbaren Programm führen. Solches Wissen könnte viel eleganter in Form eines Constraintsolvers für vordefinierte Constraints aufgeschrieben werden.

4.6 Probleme mit der Äquivalenz

$\cong$ ist für die praktische Anwendung zu schwach, denn es identifiziert semantisch identische, syntaktisch aber unterschiedliche Objekte nicht: $A \texttt{ < } B \ncong B \texttt{ > } A$, $\texttt{\textbackslash+ var}(X) \ncong \texttt{nonvar}(X)$, $\texttt{call}(G) \ncong G$, $A \texttt{ == } B \ncong B \texttt{ == } A$, $A \texttt{ == } B \wedge A \texttt{ == } B \ncong A \texttt{ == } B$. Insbesondere kennt $\cong$ keine Symmetrie-Eigenschaften und hat die Idempotenz der Konjunktion nicht eingebaut.

Zwei Lösungsmöglichkeiten sind naheliegend:

- Erweitere $\cong$ um die fehlenden Eigenschaften.
- Gestalte die update-Funktion so, daß beim Vergleich normalisierter Zustände die genannten Fälle nicht mehr auftreten:
  - Führe semantisch identische Constraintsymbole auf ein ausgezeichnetes Constraintsymbol zurück. Vertausche, wenn notwendig, die Argumente des Constraints: Schreibe $B \texttt{ > } A$ in $A \texttt{ < } B$ um, $\texttt{call}(G)$ in G, usw.

[1] Die Verzögerungsmechanismen von ECL^{i}PS^e kommen hier nicht ins Spiel, da `X < 1` gar nicht an Prolog übergeben wird.

- Sei S die Menge der vordefinierten symmetrischen Constraintsymbole. Dann ersetze jedes Constraint $p(X, Y)$ mit $p \in S$ durch $p(Y, X)$, wenn Y in der Termordnung kleiner als X ist: Schreibe `C == A` in `A == C` um, `C - 2 =:= A + 5` in `A + 5 =:= C - 2`, usw.
- Wenn ein Constraint mehrmals vorkommt, dann lösche alle Vorkommen bis auf eins.

Für den vorliegenden Konfluenz-Tester ist die vorgeschlagene Behandlung von symmetrischen Constraints jedoch nicht ausreichend: $\cong$ wird in `joinable` und `entails` benötigt. Dort haben die zu vergleichenden Zustände keine Variablen gemeinsam; es wird mit Kopien dieser Zustände gearbeitet (s. 3.9). Da man keinen Einfluß darauf hat, welche Namen Prolog für frische Variablen beim Kopieren von Termen mit `copy_term` vergibt, ist das Anordnen der Argumente nach der lexikographischen Ordnung keine zuverlässige Methode zur Normalisierung.

Eine andere Möglichkeit ist die, für jedes Constraint $p(X, Y)$ mit $p \in S$ ein Constraint $p(Y, X)$ *hinzuzufügen*. Diese Propagation ist aber nur zusammen mit der Idempotenz der Konjunktion mit der Idempotenz der update-Funktion verträglich. Die Idempotenz der Konjunktion verhindert nämlich das Aufblasen des Constraintspeichers bei sukzessiver Anwendung einer update-Funktion, die diese Propagation durchführt.

4.7 Der Metainterpreter

`metacall(`G`, `D`)` wertet die Liste G von Teilzielen aus und unifiziert D mit der sortierten Liste der normalisierten Teilziele von G, deren Berechnung nicht möglich oder nicht sinnvoll ist. `metacall` scheitert, wenn ein Teilziel nicht erfüllbar ist.

Aus historischen Gründen arbeitet `metacall` wie Prolog: Teilziele werden von links nach rechts ausgewertet und nicht in Beziehung gesetzt. Der Metainterpreter weist somit die in 4.5 diskutierte Schwäche beim Erkennen von Widersprüchen auf.

`metacall` implementiert das in 4.6 vorgeschlagene Verfahren zur Ausgleichung der Schwächen von $\cong$. Um den Blick auf das Wesentliche nicht zu verstellen, fehlen im folgenden die Regeln, die einfaches Umschreiben von z.B. A `>` B in B `<` A erledigen. Sie können im Anhang nachgelesen werden.

Die erste Regel behandelt den Fall, daß G eine Variable ist.

```
metacall(G, [G]) :- var(G), !.
```

Die folgenden Regeln behandeln den Fall, daß G eine Liste von Teilzielen ist. Die Anwendung von `sort` implementiert die Idempotenz der Konjunktion.

```
metacall([], D) :- !, D = [].
metacall([H | T], D) :- !,
    metacall(H, D0),
    metacall(T, D1),
    append(D0, D1, D2),
    sort(D2, D).
```

Die folgende Regel behandelt den Fall, daß G eine Konjunktion von Teilzielen ist. Die Anwendung von `sort` implementiert die Idempotenz der Konjunktion.

```
metacall((H, T), D) :- !,
    metacall(H, D0),
    metacall(T, D1),
    append(D0, D1, D2),
    sort(D2, D).
```

Ab hier kann davon ausgegangen werden, daß G ein Atom ist.

Zur Erhöhung der Effizienz erledigen die folgenden Regeln häufig vorkommende Fälle.

```
metacall(A = B, []) :- !, A = B.
metacall(fail, _) :- !, fail.
metacall(false, _) :- !, fail.
metacall(true, []) :- !.
```

Abschnitt 4.4 folgend werden Cuts ignoriert.

```
metacall(!, []) :- !.
```

Ein benutzerdefiniertes Constraint im Rumpf einer Prolog-Klausel macht keinen Sinn, da die operationale Semantik keine Regel definiert, die ein benutzerdefiniertes Constraint aus dem Speicher für vordefinierte Constraints in den Speicher für benutzerdefinierte Constraints transportiert. In Anlehnung an die Syntaxdefinition in [Frü95] wird ein benutzerdefiniertes Constraint als Syntaxfehler interpretiert und die Auswertung mit einer Fehlermeldung abgebrochen. Beispiele für benutzerdefinierte Constraints in Rümpfen von Prolog-Klauseln finden sich in der ECL^{i}PS^e-CHR-Bibliothek.

```
metacall(G, _) :-
    is_userdef_constraint(G), !,
    write("User-defined constraint met while proving Prolog goal!"), nl,
    write("Syntax error!"), nl,
    abort.
```

`var`(T) kann nicht ausgewertet werden, wenn T eine Variable ist, da T dann für einen beliebigen Term steht. Wenn T atomar ist, dann ist `var`(T) nicht erfüllt.

```
metacall(var(T), _) :- nonvar(T), !, fail.
metacall(var(T), [var(T)]) :- !.
```

`nonvar`(T) kann nicht ausgewertet werden, wenn T eine Variable ist, da T dann für einen beliebigen Term steht. Wenn T atomar ist, dann ist `nonvar`(T) erfüllt.

```
metacall(nonvar(T), []) :- nonvar(T), !.
metacall(nonvar(T), [nonvar(T)]) :- !.
```

`ground`(T) kann nicht ausgewertet werden, wenn T nicht variablenfrei ist, da Variablen in T für beliebige Terme stehen. Wenn T variablenfrei ist, dann ist `ground`(T) erfüllt.

```
metacall(ground(T), []) :- ground(T), !.
metacall(ground(T), [ground(T)]) :- !.
```

`nonground`(T) kann nicht ausgewertet werden, wenn T nicht variablenfrei ist, da Variablen in T für beliebige Terme stehen. Wenn T variablenfrei ist, dann ist `nonground`(T) nicht erfüllt.

```
metacall(nonground(T), _) :- ground(T), !, fail.
metacall(nonground(T), [nonground(T)]) :- !.
```

Ein vordefiniertes Typtest-Prädikat, mit Ausnahme von `var`, `nonvar`, `ground` und `nonground`, kann nicht ausgewertet werden, wenn das Argument eine Variable ist, da eine Variable für einen beliebigen Term steht. Wenn das Argument keine Variable ist, dann ist es schon eine Zahl, eine Liste oder ein anderes Atom. In diesem Fall kann das Prädikat an Prolog übergeben werden. Analog kann die Behandlung der Negation eines vordefinierten Typtest-Prädikats begründet werden.

```
metacall(not G, D) :-
    functor(G, Name, Arity),
    is_typetest_built_in_symbol(Name/Arity), !,
    arg(1, G, A),
    (var(A) -> D = [not G]; (not G, D = [])).

metacall(G, D) :-
    functor(G, Name, Arity),
    is_typetest_built_in_symbol(Name/Arity), !,
    arg(1, G, A),
    (var(A) -> D = [G]; (G, D = [])).
```

A `is` B kann nur dann an Prolog übergeben werden, wenn B variablenfrei ist, da Variablen in B zu einem Prolog-Fehler führen würden. In allen anderen Fällen kann A `is` B nicht ausgewertet werden.

```
metacall(A is B, []) :- ground(B), !, A is B.
metacall(A is B, [A is B]) :- !.
```

A `=:=` B kann nur dann an Prolog übergeben werden, wenn A und B variablenfrei sind, da Variablen in A oder B zu einem Prolog-Fehler führen würden. Wenn A und B syntaktisch gleich sind, dann ist A `=:=` B allgemeingültig. In allen anderen Fällen kann A `=:=` B nicht ausgewertet werden. Analog kann die Implementation für A `=\=` B begründet werden.

```
metacall(A =:= B, []) :- A == B, !.
metacall(A =:= B, []) :- ground(A), ground(B), !, A =:= B.
metacall(A =:= B, [A =:= B, B =:= A]) :- !.

metacall(A =\= B, []) :- A == B, !, fail.
metacall(A =\= B, []) :- ground(A), ground(B), !, A =\= B.
metacall(A =\= B, [A =\= B, B =\= A]) :- !.
```

A `<` B kann nur dann an Prolog übergeben werden, wenn A und B Zahlen sind, da andere Terme für A oder B zu einem Prolog-Fehler führen würden. Wenn A und B syntaktisch gleich sind, dann ist A `<` B unerfüllbar. In allen anderen Fällen kann A `<` B nicht ausgewertet werden. Analog kann die Implementation für A `=<` B begründet werden, mit der Ausnahme, daß A `=<` B allgemeingültig ist, wenn A und B syntaktisch gleich sind.

```
metacall(A < B, []) :- number(A), number(B), !, A < B.
metacall(A < B, _) :- A == B, !, fail.
metacall(A < B, [A < B]) :- !.

metacall(A =< B, []) :- number(A), number(B), !, A =< B.
metacall(A =< B, []) :- A == B, !.
metacall(A =< B, [A =< B]) :- !.
```

Für A `\=` B können drei Fälle unterschieden werden:

- Wenn A und B nicht unifizierbar sind, dann ist A `\=` B allgemeingültig.
- Wenn A und B syntaktisch gleich (und somit unifizierbar) sind, dann gilt für alle Substitutionen σ für Variablen in A, daß $A\sigma = B\sigma$. Also ist A `\=` B nicht erfüllbar.
- Wenn A und B unifizierbar und syntaktisch ungleich sind, dann unifiziert mindestens ein Subterm von A oder B mit einer Variablen von B bzw. A. Über A `\=` B kann in diesem Fall keine Aussage gemacht werden, da eine Variable für einen beliebigen Term steht.

Im Verzögerungsfall wird der Diskussion in 4.6 folgend ein Constraint B `\=` A erzeugt.

```
metacall(A \= B, []) :- A \= B, !.
metacall(A \= B, _) :- A == B, !, fail.
metacall(A \= B, [A \= B, B \= A]) :- !.
```

Für A `==` B können drei Fälle unterschieden werden:

- Wenn A und B syntaktisch gleich sind, dann gibt es keine Substitution σ für die Variablen in A, sodaß $A\sigma \neq B\sigma$. In diesem Fall ist A `==` B also allgemeingültig.
- Wenn A und B nicht unifizierbar (und somit syntaktisch ungleich) sind, dann ist A `==` B unerfüllbar.
- Wenn A und B unifizierbar und syntaktisch ungleich sind, dann unifiziert mindestens ein Subterm von A oder B mit einer Variablen von B bzw. A. Über A `==` B kann in diesem Fall keine Aussage gemacht werden, da eine Variable für einen beliebigen Term steht.

Analog kann die Implementation für A `\==` B begründet werden.

```
metacall(A == B, []) :- A == B, !.
metacall(A == B, []) :- A \= B, !, fail.
metacall(A == B, [A == B, B == A]) :- !.

metacall(A \== B, _) :- A == B, !, fail.
metacall(A \== B, []) :- ground(A), ground(B), !, A \== B.
metacall(A \== B, [A \== B, B \== A]) :- !.
```

$A; B$ kann durch $\mathtt{p}(X_1, \ldots, X_n)$ ersetzt werden, wenn dem betrachteten Constraintsolver o.B.d.A. die Prolog-Klauseln $\mathtt{p}(X_1, \ldots, X_n)$ `:-` A. und $\mathtt{p}(X_1, \ldots, X_n)$ `:-` B. hinzufügt werden, wobei $X_1, \ldots, X_n$ die Variablen in $A; B$ sind. Beide Regeln sind auf $A; B$ anwendbar und somit darf $A; B$, wenn man der Heuristik folgt, nicht ausgewertet werden.

```
metacall((A; B), [A; B]) :- !.
```

Alternativ ist es hier aber auch möglich, die Anzahl der Modelle festzustellen.

```
metacall((A; B), [A; B]) :-
    (var(A) -> true; A \= (_ -> _)), !,
    (   copy_term(A, A_copy), metacall(A_copy, _),
        copy_term(B, B_copy), metacall(B_copy, _)
    ->  D = [A; B]
    ;   (   metacall(A, D)
        ->  true
        ;   metacall(B, D)
        )
    ).
```

Für C `->` $T; E$ [2] können vier Fälle unterschieden werden:

- Wenn T und E syntaktisch gleich sind, dann muß C nicht betrachtet werden und C `->` $T; E$ kann in T umgeschrieben werden.

- Wenn T und E syntaktisch ungleich sind, dann können drei Fälle unterschieden werden:

  - Wenn C unerfüllbar ist, dann ist C `->` $T; E$ genau dann erfüllbar, wenn E erfüllbar ist.

  - Wenn die Auswertung von C erfolgreich ist und Ziele D verzögert, dann ist D eine bedingte Antwort für C: Wenn D erfüllbar ist, dann ist auch C erfüllbar. In diesem Fall ist C `->` $T; E$ genau dann erfüllbar, wenn D `->` $T\sigma; E\sigma$ erfüllbar ist, wobei σ die Substitution für die Variablen in C ist, die die Auswertung von C erzeugt hat.

  - Wenn die Auswertung von C erfolgreich ist und keine Ziele verzögert, dann ist die von der Auswertung erzeugte Substitution σ für die Variablen in C das einzige Modell für C. In diesem Fall ist C `->` $T; E$ genau dann erfüllbar, wenn $T\sigma$ erfüllbar ist.

[2] `->` findet in den Solvern Domain, List, Minmax und Time der ECL^{i}PS^e-CHR-Bibliothek Verwendung.

```
metacall(_ -> T; E, D) :- T == E, !, metacall(T, D).
metacall(C -> T; E, D) :- !,
    (   metacall(C, D_for_C)
    ->  (   D_for_C == []
        ->  metacall(T, D)
        ;   D = [D_for_C -> T; E]
        )
    ;   metacall(E, D)
    ).
```

Für `not` G können drei Fälle unterschieden werden:

- Wenn G erfüllbar und nicht variablenfrei ist, dann kann `not` G nicht ausgewertet werden: Sei $Z = G_1 \wedge \ldots \wedge G_k \wedge \texttt{not}\ G \wedge G_{k+1} \wedge \ldots \wedge G_n$ ein Ziel, sodaß σ ein Modell für $G_1 \wedge \ldots \wedge G_k$ ist. Wenn $G\sigma$ erfüllbar ist, dann bedeutet der Schluß auf die Unerfüllbarkeit von `not` G, daß Z unerfüllbar ist. Dieser Schluß ist nicht korrekt, da die Möglichkeit besteht, daß es ein Modell τ für $G_{k+1}\sigma \wedge \ldots \wedge G_n\sigma$ gibt, sodaß $G\sigma\tau$ unerfüllbar ist. In diesem Fall wäre $\sigma\tau$ ein Modell für Z. Diese Problematik ist aus der Prolog-Programmierung bekannt.
- Wenn G erfüllbar und variablenfrei ist, dann ist `not` G unerfüllbar.
- Wenn G unerfüllbar ist, dann ist `not` G allgemeingültig.

```
metacall(not G, []) :- not metacall(G, _), !.
metacall(not G, []) :- ground(G), !, fail.
metacall(not G, [not G]) :- !.
```

Auch für andere Prädikate, z.B. `append` und `delete`, könnten Spezialfälle behandelt werden. Dies geschah im Rahmen dieser Arbeit nicht mehr, der Metainterpreter kann aber sehr einfach um wenige Regeln zur Behandlung von Spezialfällen erweitert werden, wenn dies notwendig sein sollte.

Für ein benutzerdefiniertes Prädikat P können fünf Fälle unterschieden werden:

- Wenn P variablenfrei ist, dann ist P genau dann allgemeingültig, wenn es einen Widerlegungsbeweis für P gibt.
- P ist nicht variablenfrei:
  - Wenn keine Regel auf P anwendbar ist, dann ist P unerfüllbar.
  - Genau eine Regel ist auf P anwendbar:
    * Wenn die Auswertung von P kein Subziel verzögert, dann ist P allgemeingültig.
    * Wenn die Auswertung von P Subziele D verzögert, dann ist D eine bedingte Antwort für P. In diesem Fall ist P genau dann erfüllbar, wenn D erfüllbar ist. Deswegen wird P durch D ersetzt. Um ein eventuelles Aufblasen des Constraintspeichers zu vermeiden, könnte anstelle von D auch $P\sigma$ verzögert werden, wobei σ die Substitution für die Variablen in P ist, die die Auswertung von P erzeugt hat.

- Wenn mehrere Regeln auf P anwendbar sind, dann besteht die Möglichkeit, daß mehrere Modelle für P existieren. Der Heuristik 4.3.1 folgend wird P in diesem Fall verzögert.

```
metacall(P, D) :-
    is_userdef_predicate(P), !,
    (   ground(P)
    ->  (P, D = [])
    ;   (   deterministic(P)
        ->  (clause(P, B), metacall(B, D))
        ;   D = [P]
        )
    ).

deterministic(P) :-
    findall(_, clause(P, _), S),
    (S == []; S = [_]).
```

Ein vordefiniertes Prädikat P wird nur dann an Prolog übergeben, wenn `delays`(P) scheitert.

```
metacall(P, D) :- !,
    (   delays(P)
    ->  D = [P]
    ;   (P, D = [])
    ).
```

`delays`(A) ist genau dann erfolgreich, wenn das Atom A nicht an Prolog übergeben werden darf, da die Auswertung nicht terminieren, mit Fehler abbrechen oder ein falsches Ergebnis liefern würde. Diese Datenbank kritischer Fälle ist nicht vollständig und für manche Prädikate auch zu streng.

```
delays(T =.. L) :- var(T), var(L).
delays(append(A, B, C)) :-
    ((var(A), var(B)); (var(A), var(C))).
delays(delete(A, B, C)) :-
    ((var(A), var(B)); (var(A), var(C)); (var(B), var(C))).
delays(checklist(P, L)) :- (var(P); var(L)).
delays(flatten(N, _)) :- var(N).
delays(intersection(_, _, _)).
delays(length(L, N)) :- var(L), var(N).
delays(maplist(P, O, N)) :- (var(P); (var(O), var(N))).
delays(member(_, _)).
delays(subset(S, L)) :- (var(S); var(L)).
delays(subtract(K, L, _)) :- (var(K); var(L)).
delays(union(K, L, _)) :- (var(K); var(L)).
delays(bagof(_, _, _)).
delays(coverof(_, _, _)).
delays(findall(_, _, _)).
```

```
delays(setof(_, _, _)).
delays(sort(L, _)) :- var(L).
delays(sort(K, O, R, _)) :- (var(K); var(O); var(R)).
delays(instance(A, B)) :- not (ground(A), ground(B)).
delays(A @=< B) :- A \== B, not (ground(A), ground(B)).
delays(A @< B) :- A \== B, not (ground(A), ground(B)).
```

Kapitel 5

Zusammenfassung und Ausblick

Es wurde ein Konfluenz-Tester für CHR-Programme vorgestellt. Die Implementation beruht auf der in [AFM96] gegebenen entscheidbaren, syntaktischen Charakterisierung von konfluenten CHR-Programmen durch die Zusammenführbarkeit kritischer Paare, die allerdings nur für propagationsfreie Programme funktioniert. Zum Experimentieren erlaubt der Konfluenz-Tester aber auch nicht propagationsfreie Programme als Eingabe. Dazu mußte die Definition des kritischen Paares aus [AFM96] erweitert werden. Allerdings sind die Ergebnisse für nicht propagationsfreie Programme mit großer Vorsicht zu genießen. Es ist eine Aufgabe künftiger Arbeit, eine für die Praxis ausreichend leistungsfähige Anwendungsbedingung anzugeben, mit der die Charakterisierung konfluenter CHR-Programme auch auf nicht propagationsfreie Programme ausgedehnt werden kann.

Zur Auswertung von Prolog-Prädikaten verwendet der Konfluenz-Tester einen Metainterpreter, dessen Entwicklung ein Schwerpunkt dieser Arbeit war. Es wurde ausführlich erläutert, warum Prolog-Prädikate nicht Prolog allein überlassen werden können und welche Aussagen über vordefinierte und benutzerdefinierte Prolog-Prädikate in der Umgebung eines Konfluenz-Tests gemacht werden können. Der Metainterpreter kann sehr einfach erweitert werden.

Bei der Implementation wurde auf Benutzerfreundlichkeit, Effizienz und Plattformunabhängigkeit geachtet. Die Überlegungen und Designentscheidungen zur Erreichung von Effizienz wurden ausführlich in dieser Arbeit diskutiert.

Es wurde erläutert, daß der Metainterpreter Constraints nicht in Beziehung setzen kann, woraus u.a. eine deutliche Schwäche bei der Erkennung inkonsistenter Constraintspeicher resultiert. Eine Aufgabe für eine künftige Arbeit könnte darin bestehen, das für den Metainterpreter entwickelte Know-How in der Definition einer update-Funktion und schließlich in der Implementation eines Constraintsolvers für vordefinierte Prolog-Prädikate aufgehen zu lassen. Der zur Ausführung dieses Solvers notwendige CHR-Interpreter kann leicht aus `expand` durch Einfügen eines Cuts nach dem Aufruf von `unfold` gewonnen werden. Diesen CHR-Interpreter könnte man dann auch zur Entwicklung von Constraintsolvern auf anderen Plattformen als ECL^{i}PS^e einsetzen.

Da sich der Konfluenz-Tester bei der Bildung kritischer Paare und beim Angleichen in **Simplify** und **Propagate** auf die Unifikation von Prolog stützt, werden Gleichheitsconstraints nie explizit, sondern werden von Prolog in Form von Substitutionen repräsentiert (s. 3.9). Die automatische Propagation neuer Bindungen bereitete viele Probleme: Die Notwendigkeit des Arbeitens mit Kopien der Ausgangszustände in den Implementationen von $\downarrow$ und $\rightarrow_o$ erschwerte insbesondere die Implementation von $\cong$. Die Implementation der update-

Funktion muß deswegen eine Propagation für symmetrische Constraints durchführen (s. 4.6). Es wäre sicherlich einfacher gewesen, eine update-Funktion mit eingebauter Unifikation zu implementieren. Das könnte Aufgabe einer künftigen Arbeit sein.

Einer weiteren Untersuchung bedürfen auch die Komplexitätsprobleme, die beim Testen von Constraints, deren Definition viele Regeln umfaßt, auftreten können: Bei der Entscheidung der Zusammenführbarkeit für Zustände S und T werden Berechnungen $S \rightarrow_P^* S'$ und $T \rightarrow_P^* T'$ erzeugt. Sind S' und T' nicht äquivalent, dann wird $T \rightarrow_P^* T'$ verworfen und, wenn möglich, eine zweite Berechnung $T \rightarrow_P^* T''$ für T erzeugt und S' mit T'' verglichen, usw. Gibt es keinen Endzustand für T, der zu S' äquivalent ist, dann wird, wenn möglich, eine zweite Berechnung $S \rightarrow_P^* S''$ für S erzeugt und ein Endzustand für T gesucht, der zu S'' äquivalent ist, wobei Endzustände für T erneut erzeugt werden, usw. Im schlimmsten Fall, wenn S und T nicht zusammenführbar sind und es n Berechnungen für S und m Berechnungen für T gibt, werden $(m+1)n$ Berechnungen erzeugt und mn Zustandsvergleiche durchgeführt.

Naheliegend ist es, zuerst alle Endzustände für S und T zu erzeugen und in einer Datenbank abzulegen und dann nach äquivalenten Endzuständen zu suchen. Eine Implementation dieses Prinzips brachte jedoch keine Vorteile: Erwartungsgemäß verkürzten sich die Laufzeiten für nicht zusammenführbare Zustände. Für zusammenführbare Zustände verlängerten sich die Laufzeiten in vielen Fällen jedoch drastisch, insbesondere dann, wenn schon die Ergebnisse der ersten Berechnungen für die Zustände äquivalent gewesen wären.

Erfolgsversprechender erscheint es, während der Suche nach einem zu S' äquivalenten Endzustand für T eine Datenbank der Endzustände für T anzulegen. Wenn es keinen Endzustand für T gibt, der zu S' äquivalent ist, dann kann für weitere Vergleiche auf diese Datenbank zurückgegriffen werden.

Anzustreben ist auch eine Verkleinerung des Suchraums. Ein Anfang wurde schon gemacht: `expand` führt zuerst alle möglichen **Introduce**- und **Solve**-Schritte durch und verkleinert damit die Anzahl der Berechnungen für einen Zustand wesentlich.

Kapitel 6

Persönliche Erfahrungen

Schon vor Beginn meiner Arbeit am Konfluenz-Tester befaßte sich Holger Meuss im Rahmen seiner Diplomarbeit mit den theoretischen Grundlagen eines Konfluenz-Tests für CHR-Programme und entwickelte dabei den Prototyp eines Konfluenz-Testers, von dem ausgehend ich den vorliegenden Konfluenz-Tester erarbeitete. Von daher gab es zeitweise regen Austausch zwischen Holger Meuss und mir. Auf gelegentlichen Treffen der Konfluenz-Gruppe (Slim Abdennadher, Thom Frühwirth, Holger Meuss und ich) wurden Aufgaben verteilt und die Arbeit an den einzelnen Projekten koordiniert. Der Austausch kam mitunter etwas zum Erliegen, da die einzelnen Mitglieder der Konfluenz-Gruppe durch andere Aufgaben zeitweise gebunden waren; ich selbst konnte während des Wintersemesters wegen des parallelen Besuchs von Vorlesungen nicht soviel Zeit erübrigen, wie es für ein zügiges Fortschreiten der Arbeiten notwendig gewesen wäre.

Viele Diskussionen zwischen den Mitgliedern der Konfluenz-Gruppe fanden per Email statt. Eine Diskussion per Email hat für mich u.a. den Vorteil, das sie selbst schon Protokoll ist, sodaß es sehr gut möglich ist, Gründe für Designentscheidungen, etc. im nachhinein im Detail nachzuvollziehen.

Mit den Prolog zugrundeliegenden Prinzipien (Resolution, Unifikation, Rücksetzen) war ich auch schon vor Beginn des Praktikums gut vertraut. Die Arbeit am Konfluenz-Tester war eine gute Gelegenheit, auch den Umgang mit dem Debugger und die vordefinierten Prolog-Prädikate zu erlernen. Es wurde mir klar, daß es wie für den C-Programmierer auch für den Prolog-Programmierer wichtig ist, den Umfang der Bibliotheken und die Eigenschaften der einzelnen Funktionen bzw. Prädikate zu kennen, insbesondere dann, wenn das Programm auf verschiedenen Plattformen, in diesem Fall ECL^{i}PS^e und SWI-Prolog, laufen soll.

Im Rahmen der für dieses Projekt benötigten Funktionalität waren die Unterschiede zwischen den beiden Prolog-Systemen nicht groß. Die Bibliotheken des SWI-Prolog-Systems sind etwas weniger umfangreich als die von ECL^{i}PS^e. Es war jedoch kein Problem, die fehlenden Prädikate zu implementieren. Kleine Unterschiede der Schnittstellen der Eingabe/Ausgabe-Prädikate waren leicht auszugleichen. Die Benutzerschnittstelle von SWI-Prolog ist durch Verwendung der GNU-Readline-Bibliothek deutlich besser als die von ECL^{i}PS^e. Die Systeme sind in etwa gleich schnell und laufen stabil (ECL^{i}PS^{e}Version 3.5.1, SWI-Prolog Version 2.7.4[1]).

Erstmals lernte ich den theoretischen Hintergrund einer Programmiersprache kennen. Begriffe wie *operationale Semantik*, *deklarative Semantik*, *Terminierung* und *Korrektheit* rückten

[1] `ftp://swi.psy.uva.nl/pub/SWI-Prolog/`

mir deutlich ins Bewußtsein. Auch konnte ich live am Rechner erfahren, was es bedeutet, ein „Komplexitätsproblem“ zu haben.

Das Schreiben dieses Berichts war nicht nur eine gute Übung im Schreiben deutscher Texte, sondern forderte auch eine sehr eingehende Auseinandersetzung mit den theoretischen Grundlagen des Konfluenz-Testers. Einige Fehler im Programm entdeckte ich erst beim Versuch der Dokumentation. Auch wurde erst beim Schreiben deutlich, daß es für die Verständlichkeit des Programms und den Dokumentationsprozess vorteilhaft ist, die Struktur des Begriffsapparates der Theorie auf das Programm zu übertragen. Bei Testläufen entdeckte Programmfehler wiederum vertieften das Verständnis der Theorie. Schreiben von Bericht und Programm waren also verzahnte, inkrementelle Prozesse. Eine frühere, intensive Beschäftigung mit der Theorie hätte u.U. geholfen, Programmfehler, deren Lokalisierung oftmals ein Prozess war, der viel Zeit und Nerven kostete, zu vermeiden.

Die Literatur-Recherche war nicht aufwendig: Die Literaturdatenbank *Collection of Computer Science Bibliographies*[2] lieferte nach Angabe von Stichwörtern schon nach wenigen Sekunden eine Bibliographie, die direkt als Eingabe für BibTeX verwendet werden konnte.

[2] `http://liinwww.ira.uka.de/bibliography/`

Literaturverzeichnis

[AFM96] 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. Springer, August 1996.

[BFL+94] P. Brisset, T. Frühwirth, P. Lim, M. Meier, T. Le Provost, J. Schimpf und M. Wallace. *ECL^iPS^e 3.4 Extensions User Manual.* ECRC Munich Germany, Juli 1994.

[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, Seiten 31–44, 1988.

[ECR94] ECRC Munich Germany. *ECL^iPS^e 3.5 BIP Book*, Dezember 1994.

[ECR95] ECRC Munich Germany. *ECL^iPS^e 3.5 User Manual*, Februar 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, Seiten 3–35. Springer, 1992.

[FM94] E. C. Freuder und A. K. Mackworth, Hrsg. *Constraint-Based Reasoning.* MIT Press, Cambridge, Massachusetts, 1994.

[Frü95] T. Frühwirth. Constraint Handling Rules. In A. Podelski, Hrsg., *Constraint Programming: Basics and Trends*, LNCS 910. Springer, März 1995.

[JM94] J. Jaffar und M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 20:503–581, 1994.

[Kum92] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 1987.

[Meu96] Holger Meuss. Konfluenz von Constraint-Handling-Rules-Programmen. Diplomarbeit, Institut für Informatik, Ludwig–Maximilians–Universität München, 1996.

[vH91] P. van Hentenryck. Constraint Logic Programming. *The Knowledge Engineering Review*, 6:151–194, 1991.

[vHSD92] P. van Hentenryck, H. Simonis und M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence*, 58(1-3):113–159, Dezember 1992.

Anhang A

Quelltexte

```
%------------------------------------------------------------------------------
% ConTest
% A Test for Confluence of Constraint Solvers
%------------------------------------------------------------------------------


% Preprocessing

/*
The GNU C preprocessor is used for macro expansion.
There must not be any ' or " characters in Prolog comments.
Don't use them or better use the C/ C++ way to write comments.
*/
#ifdef DEBUG_ON
#define DEBUG(Goal) Goal
#define DEBUGV(Variable) Variable
#define NODEBUG(Goal) true
#else
#define DEBUG(Goal) true
#define DEBUGV(Variable) _
#define NODEBUG(Goal) Goal
#endif


% Initializations

:- (dynamic (handler)/1, (constraints)/1, (==>)/2, (<=>)/2, (@)/2).

#ifdef ECLIPSE
:-
        % load libraries
        lib(scattered), lib(apply),

        % load the CHR interpreter
        lib(chr),

        /*
        specify the format used to print terms in the debugger
        and in answer bindings in the top-level loop:
        + use portray/1,2 if defined (default)
        + print metaterm attributes using user-defined handlers (default)
```

```
        + print variables as unique numbers (don't confuse the user!)
        */
        set_flag(output_mode, "Pmv"),

        % switch occur check on
        set_flag(occur_check, on),

        % generate optimised code when not debugging
        NODEBUG(set_flag(debug_compile, off)),
        DEBUG(set_flag(debug_compile, on)),

        % make soft cut available
        op(1050, xfx, *->),
        (import (*->)/2 from sepia_kernel).
#endif

#ifdef SWI
:-
        % make SWI-Prolog recognize Prolog string objects
        style_check(+string).
#endif


% Inclusion of Modules

#ifdef ECLIPSE
#include "eclsp.cpp"
#endif
#ifdef SWI
#include "swisp.cpp"
#endif
#include "chr_loader.cpp"
#include "control.cpp"
#include "lcp.cpp"
#include "total.cpp"
#include "partial.cpp"
#include "confluent.cpp"
#include "critical_pair.cpp"
#include "output.cpp"
#include "misc.cpp"
#include "list.cpp"
#include "metacall.cpp"


% Welcome User

version(__DATE__).
/*      Defining version/1 before style_check(+string) will return a
        list for Version for the goal version(Version). */

:-      write("Welcome to ConTest version "),
        version(Version), write(Version), nl,
        consult_off,
        trace_off,
        idempotence_off,
        joinable_off.
```

```
%------------------------------------------------------------------------------
% Control of System Parameters


:- dynamic consult_user/0.

consult_on :-
        not consult_user,
        assert(consult_user),
        fail.
consult_on :-
        write("You will be consulted in feasibilty check."), nl.


consult_off :-
        retract(consult_user),
        fail.
consult_off :-
        write("You will not be consulted in feasibilty check."), nl.


:- dynamic trace_me/0.

#ifdef DEBUG_ON

trace_on :-
        not trace_me,
        assert(trace_me),
        fail.
trace_on :-
        write("Tracing is on."), nl,
        joinable_on.

trace_off :-
        retract(trace_me),
        fail.
trace_off :-
        write("No tracing."), nl,
        joinable_off.

#else

trace_on :- no_debugging_support_compiled_in.

trace_off :- no_debugging_support_compiled_in.

no_debugging_support_compiled_in :-
        write("This tester has no debugging support compiled in."), nl.

#endif


:- dynamic use_idempotence/0.

idempotence_on :-
        not use_idempotence,
        assert(use_idempotence),
```

```
        fail.
idempotence_on :-
        write("Idempotence will be applied."), nl.


idempotence_off :-
        retract(use_idempotence),
        fail.
idempotence_off :-
        write("Idempotence will not be applied."), nl.


:- dynamic show_joinable_critical_pairs/0.

joinable_on :-
        not show_joinable_critical_pairs,
        assert(show_joinable_critical_pairs),
        fail.
joinable_on :-
        write("Confluence test will display all critical pairs."), nl.


joinable_off :-
        retract(show_joinable_critical_pairs),
        fail.
joinable_off :-
        write("Confluence test will not display joinable critical pairs."), nl.


build_message_list(Messages1) :-
        Messages0 = [user_info, critical_pairs],
        (       trace_me
        ->      Messages1 = [%feasible, introduce, idempotence, %noisy_unfold,
                        entails, update,
                        joinable, expand, solve, unfold| Messages0]
        ;       Messages1 = Messages0
        ).
```

```
%------------------------------------------------------------------------------
% CHR Program Loader


:- dynamic rule/7, critical_pair/7.


load(Basename) :-
        nonvar(Basename),
        concat_atoms(Basename, '.chr', Program_filename),
        retract_chr_program,
        Log_filename = '/dev/tty',
        open(Program_filename, read, Program_stream),
        open(Log_filename, write, Log_stream),
        read_chr_program(Program_stream, 0,
                loginfo([load], Log_filename, Log_stream)),
        close(Log_stream),
        close(Program_stream),
        write("I loaded definitions for:"), nl,
        list_constraint_symbols(_/_),
        write("Building critical pairs ..."), flush,
        build_critical_pairs,
        write(" done"), nl.


retract_chr_program :-
        retract_all(handler _),
        retract_all(constraints _),
        retract_all(rule(_, _, _, _, _, _, _)),
        retract_all(critical_pair(_, _, _, _, _, _, _)).


read_chr_program(Stream, I0, Loginfo) :-
        read(Stream, Rule),
        (Rule = end_of_file
        ->      true
        ;       (process_rule(Rule, I0, I1, Loginfo), read_chr_program(Stream, I1, Loginfo))
        ).


build_critical_pairs :-
        build_critical_pair(_/_, Critical_pair),
        % does assert copy the term?
        copy_term(Critical_pair, Critical_pair_copy),
        not critical_pair_is_recorded(Critical_pair_copy),
        assert(Critical_pair_copy),
        fail.
build_critical_pairs.

critical_pair_is_recorded(Critical_pair) :-
        Critical_pair = critical_pair(N1, N2, _, _, L0, R0, V0),
        critical_pair(N1, N2, _, _, L1, R1, V1),
        variant(V0-L0-R0, V1-L1-R1).


process_rule(:- Gs, I, I, Loginfo) :- !,
        log(Loginfo, load, ["Executing ", t, nl, flush], [Gs]),
```

```
        Gs, !.

process_rule(operator(Precedence, Type, Name), I, I, _) :- !,
        op(Precedence, Type, Name).

process_rule(label_with A if B, I, I, Loginfo) :- !,
        skipping(label_with A if B, Loginfo).

process_rule(option(A, B), I, I, Loginfo) :- !,
        skipping(option(A, B), Loginfo).

process_rule(constraints C, I, I, _) :- !,
        conj2list(C, L),
        assert(constraints L).

process_rule(H :- B, I, I, _) :- !,
        functor(H, Name, Arity),
#ifdef ECLIPSE
        (not is_predicate(Name/Arity) -> dynamic(Name/Arity); true),
#endif
#ifdef SWI
        dynamic(Name/Arity),
#endif
        assert(H :- B).

process_rule(H ==> B, I0, I1, Loginfo) :- !,
        give_a_unique_name(I0, I1, Name),
        process_rule(Name @ H ==> B, _, _, Loginfo).

process_rule(H <=> B, I0, I1, Loginfo) :- !,
        give_a_unique_name(I0, I1, Name),
        process_rule(Name @ H <=> B, _, _, Loginfo).

process_rule(Name0 @ Rule, I0, I1, _) :- !,
        (rule(_, _, Name0, _, _, _, _)
        ->      give_a_unique_name(I0, I1, Name)
        ;       (Name = Name0, I1 = I0)
        ),
        normalize_rule(Name @ Rule, Normalized_rule),
        split_rule(Normalized_rule, Name, Head, Guard, Body, Remaining_atoms),
        assert(rule(Name @ Rule, Normalized_rule, Name, Head, Guard, Body, Remaining_atoms)).

process_rule(Rule, I, I, _) :- assert(Rule).


give_a_unique_name(I0, I1, Name) :-
        (handler Handler),
        concat_atoms(Handler, '_rule_', Name0),
        integer_atom(I0, I0_atom),
        concat_atoms(Name0, I0_atom, Name),
        I1 is I0 + 1.


skipping(T, Loginfo) :-
        log(Loginfo, load, ["Skipping ", t, nl, flush], [T]).
```

```
split_rule(     (Name @ Head ==> Guard | Body),
                Name, Head_list, Guard_list, Body_list, Head_list) :-
        conj2list(Head, Head_list),
        conj2list(Body, Body_list),
        conj2list(Guard, Guard_list).

split_rule(     (Name @ Head <=> Guard | Body),
                Name, Head_list, Guard_list, Body_list, []) :-
        conj2list(Head, Head_list),
        conj2list(Body, Body_list),
        conj2list(Guard, Guard_list).


% Rules in this section define a normal form for constraint handling rules.

normalize_rule( (Name @ Head ==> Guard | Body),
                (Name @ Head ==> Guard | Body)) :- !.

normalize_rule( (Name @ Head ==> Body),
                (Name @ Head ==> true | Body)) :- !.

normalize_rule( (Name @ Head1 \ Head2 <=> Guard | Body),
                (Name @ New_head <=> Guard | New_body)) :- !,
        cappend(Head1, Head2, New_head),
        cappend(Head1, Body, New_body).

normalize_rule( (Name @ Head1 \ Head2 <=> Body),
                (Name @ New_head <=> true | New_body)) :- !,
        cappend(Head1, Head2, New_head),
        cappend(Head1, Body, New_body).

normalize_rule( (Name @ Head <=> Guard | Body),
                (Name @ Head <=> Guard | Body)) :- !.

normalize_rule( (Name @ Head <=> Body),
                (Name @ Head <=> true | Body)) :- !.
```

```
%-------------------------------------------------------------------------------
% Build and Display Critical Pairs


build_critical_pair(    Name/Arity,
                        critical_pair(Name1, Name2, Shared_atoms,
                                Guard, Left, Right, Global_vars)) :-

        % select rules from the CHR program
        rule(Rule1, _, Name1, Head1, Guard1, Body1, Remaining_atoms1),
        for_one(name_and_arity(Name/Arity), Head1),
        rule(Rule2, _, Name2, Head2, Guard2, Body2, Remaining_atoms2),
        for_one(name_and_arity(Name/Arity), Head2),

        % avoid duplicate critical pairs
        rule_geq(Rule1, Rule2),

        % make conjunction of guards
        append(Guard1, Guard2, Guard),

        % unify head atoms
        unify_heads(Head1, Head2, Shared_atoms0, Not_shared_atoms1, Not_shared_atoms2),
        for_one(name_and_arity(Name/Arity), Shared_atoms0),
        msort(Shared_atoms0, Shared_atoms),

        % make left side
        flatten([Body1, Not_shared_atoms2, Remaining_atoms1], Left0),
        msort(Left0, Left),

        % make right side
        flatten([Body2, Not_shared_atoms1, Remaining_atoms2], Right0),
        msort(Right0, Right),

        % make list of global variables
        term_variables(Head1-Head2, Global_vars).


unify_heads(H1, H2, S, R1, R2) :-
        length(H1, K),
        length(H2, L),
        min(K, L, M),
        enumeration(1, M, E),
        member(N, E),
        unify_heads(N, H1, H2, S, R1, R2).

unify_heads(0, H1, H2, [], H1, H2) :- !.

unify_heads(M, H1, H2, [A | S], R1, R2) :-
        delete(A, H1, H3),
        delete(A, H2, H4),
        N is M - 1,
        unify_heads(N, H3, H4, S, R1, R2).


display_critical_pair(Name1, Name2,
                Guard, Left, Right, Global_vars, Loginfo) :-
```

```
% get rules by index name
rule(Rule1, _, Name1, _, _, _, _),
rule(Rule2, _, Name2, _, _, _, _),

% copy components for naming
copy_term(Global_vars-
        Guard-Left-Right,
        Global_vars_copy-
        Guard_copy-Left_copy-Right_copy),

% build conjunctions for display
list2conj(Global_vars_copy, Global_vars_copy_conj),
list2conj(Guard_copy, Guard_copy_conj),
list2conj(Left_copy, Left_copy_conj),
list2conj(Right_copy, Right_copy_conj),

% name the variables
name_variables(Global_vars_copy-        % give order for naming
        Guard_copy-Left_copy-Right_copy-
        Rule1-Rule2),

% display the critical pair
log(Loginfo, critical_pairs,
        ["The CRITICAL PAIR", nl, nl, t, " |", nl, t,
        nl, "=?=", nl, t],
        [Guard_copy_conj,
        Left_copy_conj, Right_copy_conj]),
(Global_vars == []
->      log(Loginfo, critical_pairs, [nl], [])
;       log(Loginfo, critical_pairs,
                [" |", nl, t, nl],
                [Global_vars_copy_conj])
),
log(Loginfo, critical_pairs,
        [nl, "has been derived from the rules", nl, nl, t, nl, t,
        nl, flush],
        [Rule1, Rule2]).
```

```
%-----------------------------------------------------------------------------
% List Critical Pairs


lacp :- !, lacp('/dev/tty').
lacp(Filename) :-
        open(Filename, write, Stream),
        ilcp(_, loginfo([user_info, critical_pairs], Filename, Stream)),
        close(Stream).

lcp(Name/Arity) :- !, lcp(Name/Arity, '/dev/tty').
lcp(Name) :- lcp(Name/_).
lcp(Name/Arity, Filename) :- !,
        complete_constraint_specification(Name/Arity),
        open(Filename, write, Stream),
        ilcp(Name/Arity, loginfo([user_info, critical_pairs], Filename, Stream)),
        close(Stream).
lcp(Name, Filename) :- lcp(Name/_, Filename).


ilcp(Name/Arity, Loginfo) :-

        % say what we are doing
        platform(Platform),
        log(Loginfo, user_info,
                ["Listing critical pairs for ", t, " on platform ", t, nl, nl, flush],
                [Name/Arity, Platform]),

        % get a critical pair for the specified constraint
        critical_pair(Name1, Name2, Shared_atoms, Guard, Left, Right, Global_vars),
        for_one(name_and_arity(Name/Arity), Shared_atoms),

        % display the critical pair
        display_critical_pair(Name1, Name2,
                Guard, Left, Right, Global_vars, Loginfo),
        log(Loginfo, critical_pairs, [nl, nl, nl, nl], []),

        % find next critical pair
        fail.

ilcp(Name/Arity, Loginfo) :-
        log(Loginfo, user_info,
                ["No (more) critical pairs for ", t, ".", nl],
                [Name/Arity]).
```

```
%------------------------------------------------------------------------------
% Partial Confluence Test


partial(Name/Arity) :- !, partial(Name/Arity, '/dev/tty').
partial(Name) :- partial(Name/_).
partial(Name/Arity, Filename) :- !,
        complete_constraint_specification(Name/Arity),
        open(Filename, write, Stream),
        build_message_list(Messages),
        ipartial(Name/Arity, loginfo(Messages, Filename, Stream)),
        close(Stream).
partial(Name, Filename) :- partial(Name/_, Filename).


ipartial(Name/Arity, Loginfo) :-

        % say what we are doing
        platform(Platform),
        (var(Name), var(Arity)
        ->      (
                (handler Handler),
                log(Loginfo, user_info,
                        ["Performing confluence test for handler ", t,
                        " on platform ", t, nl, nl, flush],
                        [Handler, Platform])
                )
        ;       log(Loginfo, user_info,
                        ["Performing partial confluence test for ", t,
                        " on platform ", t, nl, nl, flush],
                        [Name/Arity, Platform])
        ),

        % get a critical pair for the specified constraint
        critical_pair(Name1, Name2, Shared_atoms,
                Guard, Left, Right, Global_vars),
        for_one(name_and_arity(Name/Arity), Shared_atoms),

        % check whether the critical pair is feasible
        feasible(Guard, Delayed, Loginfo),

        % display the critical pair before join for debugging purposes
        (       show_joinable_critical_pairs
        ->      display_critical_pair(Name1, Name2, Delayed, Left, Right, Global_vars, Loginfo)
        ;       true
        ),

        % proove that the critical pair is not joinable
        (not joinable(state(Left,  [], Delayed, Global_vars),
                state(Right, [], Delayed, Global_vars), Loginfo)
        ->      Result = "UNJOINABLE"
        ;       Result = "JOINABLE"
        ),

        % display the critical pair after join for user convenience
        (       (not show_joinable_critical_pairs, Result == "UNJOINABLE")
        ->      display_critical_pair(Name1, Name2,
```

```
                    Guard, Left, Right, Global_vars, Loginfo)
        ;       true
        ),

        % display result
        (       (
                show_joinable_critical_pairs;
                (not show_joinable_critical_pairs, Result == "UNJOINABLE")
                )
        ->      log(Loginfo, critical_pairs,
                        [nl, "This CRITICAL PAIR is ", t, "!", nl, nl, nl, nl, nl, flush],
                        [Result])
        ;       true
        ),

        % examine next critical pair
        fail.

% say good bye

ipartial(Name/Arity, Loginfo) :-
        (var(Name), var(Arity)
        ->      (
                (handler Handler),
                log(Loginfo, user_info,
                        ["No (more) (un)joinable critical pairs for handler ", t, ".", nl],
                        [Handler])
                )
        ;       log(Loginfo, user_info,
                        ["No (more) (un)joinable critical pairs for ", t, ".", nl],
                        [Name/Arity])
        ).


% ipartial with a list interface

ipartial([], _Loginfo) :- !.
ipartial([Name/Arity | Tail], Loginfo) :-
        ipartial(Name/Arity, Loginfo),
        ipartial(Tail, Loginfo).


feasible(Guard, User_delayed, Loginfo) :-
        DEBUG(log(Loginfo, feasible,
                ["** feasible: examining ", t, nl],
                [Guard])),
        update(state([], [], Guard, []), state([], [], Delayed, []), Loginfo),
        (       consult_user
        ->      consult_user(Delayed, User_delayed)
        ;       User_delayed = Delayed
        ),
        DEBUG(log(Loginfo, feasible,
                ["** feasible: returning ", t, nl],
                [User_delayed])),
        User_delayed \== [false].
```

```
joinable(S, T, _) :- S == T, !.

joinable(S, T, Loginfo) :-

        % avoid side effects
        copy_term(S, U),
        copy_term(T, V),

        % decide joinability
        expand(U, W, Loginfo),
        expand(V, X, Loginfo),
        DEBUG(log(Loginfo, joinable,
                ["** joinable: comparing ", nl, t, nl, "with", nl, t, nl],
                [W, X])),
        equivalent_states(W, X),

        !.


expand(S, T, Loginfo) :-
        DEBUG(expansion_starts(S, Loginfo)),
        introduce(S, A, Loginfo),
        idempotence(A, B, Loginfo),
        solve(B, C, Loginfo),
        expand(C, T, [], Loginfo).

expand(S, T, History, Loginfo) :-
        unfold(S, A, History, New_history, Loginfo),
        introduce(A, B, Loginfo),
        idempotence(B, C, Loginfo),
        solve(C, D, Loginfo),
        expand(D, T, New_history, Loginfo)
        *->     true
        ;       (T = S, DEBUG(expansion_finishes(T, History, Loginfo))).

#ifdef DEBUG_ON
expansion_starts(State, Loginfo) :-
        log(Loginfo, expand,
                ["** expand: expanding ", nl, t, nl],
                [State]).

expansion_finishes(State, History, Loginfo) :-
        log(Loginfo, expand,
                ["** expand: expansion finishes in ", nl, t, nl, "History:", nl, t, nl, nl],
                [State, History]).
#endif


introduce(S, T, DEBUGV(Loginfo)) :-
        DEBUG(log(Loginfo, introduce,
                ["** introduce: transformed", nl, t, nl],
                [S])),
        S = state(Goals0, Userdef0, Builtin, Global_vars),
        filter(is_userdef_constraint, Goals0, Userdef1, Goals1),
        append(Userdef0, Userdef1, Userdef2),
        T = state(Goals1, Userdef2, Builtin, Global_vars),
        DEBUG(log(Loginfo, introduce,
```

```
                ["into", nl, t, nl],
                [T])).


solve(S, T, DEBUGV(Loginfo)) :-
        DEBUG(log(Loginfo, solve,
                ["** solve: transformed", nl, t, nl],
                [S])),
        S = state(Goals0, Userdef, Builtin0, Global_vars),
        append(Goals0, Builtin0, Builtin1),
        update(state([], Userdef, Builtin1, Global_vars), T, loginfo([], _, _)),
        DEBUG(log(Loginfo, solve,
                ["into", nl, t, nl],
                [T])).


idempotence(S, T, DEBUGV(Loginfo)) :-
        use_idempotence,
        !,
        DEBUG(log(Loginfo, idempotence,
                ["** idempotence: transformed", nl, t, nl],
                [S])),
        S = state(Goals, Userdef0, Builtin, Global_vars),
        sort(Userdef0, Userdef1),
        T = state(Goals, Userdef1, Builtin, Global_vars),
        DEBUG(log(Loginfo, idempotence,
                ["into", nl, t, nl],
                [T])).

idempotence(S, S, _).


unfold(state(_, _, [false], _), _, _, _, _) :- !, fail.

unfold(S, T, History, New_history, Loginfo) :-

        % split initial state
        S = state([], Userdef, Builtin, Global_vars),

        % rule selection and matching
        rule(DEBUGV(Rule), Normalized_rule, Name, Head, Guard, Body, Surviving_atoms),
        DEBUG(log(Loginfo, noisy_unfold,
                ["** unfold: trying rule ", t, nl],
                [Name])),
        matches(Head, Userdef, Unmatched_atoms),
        DEBUG(log(Loginfo, noisy_unfold,
                ["** unfold: head of rule ", t, " matches,", nl,
                "instance of rule: ", t, nl],
                [Name, Rule])),

        % avoid trivial non-termination for propagation rules
        (       Normalized_rule = (_ @ _ ==> _)
        ->      not member(==, (Name, Head), History),
                DEBUG(log(Loginfo, noisy_unfold,
                        ["* unfold: instance of rule ", t, " has not yet been applied", nl],
                        [Name]))
        ;       true
```

```
        ),

        % guard check
        entails(state(Head, [], Builtin, Global_vars),
                state(Head, [], Guard,   Global_vars), Loginfo),

        % fire
        append(Unmatched_atoms, Surviving_atoms, Userdef1),
        T = state(Body, Userdef1, Builtin, Global_vars),
        DEBUG(log(Loginfo, unfold,
                ["** unfold: rule ", t, " fires on", nl, t, nl, "resulting in",
                nl, t, nl],
                [Name, S, T])),

        % record application of instance of propagation rule
        (       Normalized_rule = (_ @ _ ==> _)
        ->      New_history = [(Name, Head) | History],
                DEBUG(log(Loginfo, noisy_unfold,
                        ["** unfold: application of instance of rule has been recorded",
                        nl, "new history: ", t, nl],
                        [New_history]))
        ;       New_history = History
        ).

unfold(State, State, History, History, DEBUGV(Loginfo)) :- !,
        DEBUG(log(Loginfo, unfold,
                ["** unfold: no more rule applies to ", nl, t, nl],
                [State])),
        fail.


matches(K, L, R) :-
        length(K, M),
        length(L, N),
        M =< N,
        select(L, M, S, R),
        instance(S, K),
        K = S.


entails(S, T, Loginfo) :-

        DEBUG(log(Loginfo, entails,
                ["** entails: testing wheter ", nl, t, nl, "entails ", nl, t, nl],
                [S, T])),

        % avoid side effects
        copy_term(S, W),
        copy_term(S-T, U-V),

        % decide entailment
        update(W, X, Loginfo),
        U = state(_, _, U_builtin, _),
        V = state(V_goals, V_userdef, V_builtin, V_global_vars),
        append(U_builtin, V_builtin, Builtin),
        update(state(V_goals, V_userdef, Builtin, V_global_vars), Y, Loginfo),
        equivalent_states(X, Y),
```

```
        DEBUG(log(Loginfo, entails,
                ["** entails: ", nl, t, nl, "entails ", nl, t, nl],
                [S, T])).


update(S, T, DEBUGV(Loginfo)) :-
        DEBUG(log(Loginfo, update,
                ["** update: transformed", nl, t, nl],
                [S])),
        S = state(Goals, Userdef, Builtin0, Global_vars),
        (metacall(Builtin0, Delayed) -> Builtin1 = Delayed; Builtin1 = [false]),
        T = state(Goals, Userdef, Builtin1, Global_vars),
        DEBUG(log(Loginfo, update,
                ["into", nl, t, nl],
                [T])).


equivalent_states(state(_, _, [false], _), state(_, _, [false], _)) :- !.

equivalent_states(S, T) :-

        % avoid side effects
        copy_term(S, U),
        copy_term(T, V),

        % compare results for global variables
        U = state(_, _, _, U_global_vars),
        V = state(_, _, _, V_global_vars),
        variant(U_global_vars, V_global_vars),

        % identify global variables of U and V
        U_global_vars = V_global_vars,
        Global_vars = U_global_vars,
                % variable lists are the same now, so no need for two names!

        % sort constraint stores
        sort_state(U, W),
        sort_state(V, X),

        % split up constraint stores
        W = state(U_goals, U_userdef, U_builtin, _),
        filter(has_local_variables(Global_vars),
                U_goals, U_goals_local, U_goals_global),
        filter(has_local_variables(Global_vars),
                U_userdef, U_userdef_local, U_userdef_global),
        filter(has_local_variables(Global_vars),
                U_builtin, U_builtin_local, U_builtin_global),
        X = state(V_goals, V_userdef, V_builtin, _),
        filter(has_local_variables(Global_vars),
                V_goals, V_goals_local, V_goals_global),
        filter(has_local_variables(Global_vars),
                V_userdef, V_userdef_local, V_userdef_global),
        filter(has_local_variables(Global_vars),
                V_builtin, V_builtin_local, V_builtin_global),

        % compare constraints without local variables
```

```
        state(U_goals_global, U_userdef_global,
                U_builtin_global, Global_vars) ==
        state(V_goals_global, V_userdef_global,
                V_builtin_global, Global_vars),

        % compare permutations of lists of constraints with local variables
        copy_term((U_goals_local, U_userdef_local,
                U_builtin_local, Global_vars),
                (U_goals_local_copy, U_userdef_local_copy,
                U_builtin_local_copy, Global_vars_copy)),
                % arguments to variant/2 must not share variables!
        length(U_goals_local,   M),
        length(V_goals_local,   M),
        select(V_goals_local,   M, V_goals_local_permutation, _),
        variant((V_goals_local_permutation, Global_vars),
                (U_goals_local_copy, Global_vars_copy)),
        length(U_userdef_local, N),
        length(V_userdef_local, N),
        select(V_userdef_local, N, V_userdef_local_permutation, _),
        variant((V_userdef_local_permutation, Global_vars),
                (U_userdef_local_copy, Global_vars_copy)),
        length(U_builtin_local, O),
        length(V_builtin_local, O),

        select(V_builtin_local, O, V_builtin_local_permutation, _),
        variant((V_builtin_local_permutation, Global_vars),
                (U_builtin_local_copy, Global_vars_copy)),
        !.


sort_state(S, T) :-
        S = state(Goals0, Userdef0, Builtin0, Global_vars),
        msort(Goals0, Goals1),
        msort(Userdef0, Userdef1),
        msort(Builtin0, Builtin1),
        T = state(Goals1, Userdef1, Builtin1, Global_vars).


consult_user([], []) :- !.

consult_user(Input, Output) :-
        term_variables(Input, VarList),
        list2conj(Input, Input_conj),
        nl, write("Guard:      "), write(VarList : Input_conj), nl,
        write("accept (a.) or simplify (s.)?"),
        read(s),
        !,
        nl,
        write("Enter new guard, beginning with the list of free vars:"), nl,
        write("Use the format [_, _, ..., _, X1, X2, ..., Xn] : NewGuard."), nl,
        write("Replace anonymous variables with _."), nl,
        read(UserVarList : Output_conj),
        conj2list(Output_conj, Output),
        UserVarList = VarList.

consult_user(Input, Input).
```

```
%---------------------------------------------------------------------------
% Total Confluence Test


total(Name/Arity) :- !, total(Name/Arity, '/dev/tty').
total(Name) :- total(Name/_).
total(Name/Arity, Filename) :- !,
        complete_constraint_specification(Name/Arity),
        open(Filename, write, Stream),
        build_message_list(Messages),
        itotal(Name/Arity, loginfo(Messages, Filename, Stream)),
        close(Stream).
total(Name, Filename) :- total(Name/_, Filename).


itotal(Name/Arity, Loginfo) :-

        % say what we are doing
        platform(Platform),
        log(Loginfo, user_info,
                ["Performing total confluence test for ", t, " on platform ", t, nl, nl, flush],
                [Name/Arity, Platform]),

        % build list of constraints to undergo partial confluence test
        based_upon(Name/Arity, Constraints0),
        sort([Name/Arity | Constraints0], Constraints),

        % test partial confluence
        ipartial(Constraints, Loginfo).


based_upon(Name/Arity, Bases) :- !, based_upon(Name/Arity, [], Bases).
based_upon([], Bases, Bases) :- !.
based_upon([Name/Arity | Tail], Bases0, Bases2) :- !,
        based_upon(Name/Arity, Bases0, Bases1),
        based_upon(Tail, Bases1, Bases2).

based_upon(Name/Arity, Bases0, Bases3) :-       % Bases0 must be a set!

        % get a rule and make a list of constraint specifications from its body
        rule(Name/Arity, _, _, _, Body, _, _, _),
        maplist(name_and_arity, Body, Names),
        filter(is_userdef_constraint_symbol, Names, Userdef0),
        sort(Userdef0, Userdef),

        % make a list of new base constraints (subtract needs sets as input)
        subtract(Userdef, Bases0, New_bases),
        New_bases \== [],
        !,

        /* make a new set of base constraints and compute
        the base constraints of the new base constraints */
        append(New_bases, Bases0, Bases1),
        sort(Bases1, Bases2),
        based_upon(New_bases, Bases2, Bases3).

based_upon(_, Bases, Bases).
```

```
%-----------------------------------------------------------------------------
% Test All Constraints for Confluence


confluent :- confluent('/dev/tty').
confluent(Filename) :- !,
        open(Filename, write, Stream),
        build_message_list(Messages),
        ipartial(_, loginfo(Messages, Filename, Stream)),
        close(Stream).
```

```
%------------------------------------------------------------------------------
% Prolog Meta Interpreter


% debugging
#ifdef DEBUG_METACALL
metacall(G, _) :- write("metacall: "), write(G), nl, flush, fail.
#endif


% special case: conjunction with one variable as goal
metacall(G, [G]) :- var(G), !.


% metacall with a list interface
metacall([], D) :- !, D = [].
metacall([H | T], D) :- !,
        metacall(H, D0),
        metacall(T, D1),
        append(D0, D1, D2),
        sort(D2, D).


% metacall with a conjunction interface
% necessary for handling of conjunctions in conjunctions
metacall((H, T), D) :- !,
        metacall(H, D0),
        metacall(T, D1),
        append(D0, D1, D2),
        sort(D2, D).


% now we are sure that G is not a conjunction of goals

% improve performance by taking a short cut for special cases
metacall(A = B, []) :- !, A = B.
metacall(fail, _) :- !, fail.
metacall(false, _) :- !, fail.
metacall(true, []) :- !.
metacall(!, []) :- !.   % evil cuts can be ignored :-)


% user-defined constraints are not allowed by the language definition
metacall(G, _) :-
        is_userdef_constraint(G), !,
        write("User-defined constraint met while proving Prolog goal!"), nl,
        write("Syntax error!"), nl,
        abort.


% type-testing predicates

metacall(var(T), _) :- nonvar(T), !, fail.
metacall(var(T), [var(T)]) :- !.

metacall(not var(T), D) :- !, metacall(nonvar(T), D).
metacall(nonvar(T), []) :- nonvar(T), !.
```

```
metacall(nonvar(T), [nonvar(T)]) :- !.

metacall(ground(T), []) :- ground(T), !.
metacall(ground(T), [ground(T)]) :- !.

metacall(not ground(T), D) :- !, metacall(nonground(T), D).
metacall(nonground(T), _) :- ground(T), !, fail.
metacall(nonground(T), [nonground(T)]) :- !.

metacall(not G, D) :-
        functor(G, Name, Arity),
        is_typetest_built_in_symbol(Name/Arity), !,
        arg(1, G, A),
        (var(A) -> D = [not G]; (not G, D = [])).

metacall(G, D) :-
        functor(G, Name, Arity),
        is_typetest_built_in_symbol(Name/Arity), !,
        arg(1, G, A),
        (var(A) -> D = [G]; (G, D = [])).


% arithmetics

metacall(A is B, []) :- ground(B), !, A is B.
metacall(A is B, [A is B]) :- !.

metacall(not A =\= B, D) :- !, metacall(A =:= B, D).
metacall(A =:= B, []) :- A == B, !.
metacall(A =:= B, []) :- ground(A), ground(B), !, A =:= B.
metacall(A =:= B, [A =:= B], B =:= A) :- !.

metacall(not A =:= B, D) :- !, metacall(A =\= B, D).
metacall(A =\= B, []) :- A == B, !, fail.
metacall(A =\= B, []) :- ground(A), ground(B), !, A =\= B.
metacall(A =\= B, [A =\= B, B =\= A]) :- !.

metacall(not A >= B, D) :- !, metacall(A < B, D).
metacall(A > B, D) :- !, metacall(B < A, D).
metacall(A < B, []) :- number(A), number(B), !, A < B.
metacall(A < B, _) :- A == B, !, fail.
metacall(A < B, [A < B]) :- !.

metacall(not A > B, D) :- !, metacall(A =< B, D).
metacall(A >= B, D) :- !, metacall(B =< A, D).
metacall(A =< B, []) :- number(A), number(B), !, A =< B.
metacall(A =< B, []) :- A == B, !.
metacall(A =< B, [A =< B]) :- !.


% unification

metacall(A ~= B, D) :- !, metacall(A \= B, D).
metacall(not A = B, D) :- !, metacall(A \= B, D).
% not A \= B has a special meaning!
metacall(A \= B, []) :- A \= B, !.
metacall(A \= B, _) :- A == B, !, fail.
```

```
metacall(A \= B, [A \= B, B \= A]) :- !.


% syntactic comparisions

metacall(not A == B, D) :- !, metacall(A \== B, D).
normalize_atom(not A == B, [A \== B, B \== A]).
metacall(A == B, []) :- A == B, !.
metacall(A == B, []) :- A \= B, !, fail.
metacall(A == B, [A == B, B == A]) :- !.

metacall(not A \== B, D) :- !, metacall(A == B, D).
metacall(A \== B, _) :- A == B, !, fail.
metacall(A \== B, []) :- ground(A), ground(B), !, A \== B.
metacall(A \== B, [A \== B, B \== A]) :- !.

metacall(A @>= B, D) :- !, metacall(B @=< A, D).
metacall(A @> B, D) :- !, metacall(B @< A, D).
metacall(not A @>= B, D) :- !, metacall(A @< B, D).
metacall(not A @> B, D) :- !, metacall(A @=< B, D).


% disjunction

%metacall((A; B), [A; B]) :- A \= (_ -> _), !.

metacall((A; B), D) :-
        (var(A) -> true; A \= (_ -> _)), !,
        (       copy_term(A, A_copy), metacall(A_copy, _),
                copy_term(B, B_copy), metacall(B_copy, _)
        ->      D = [A; B]
        ;       (       metacall(A, D)
                ->      true
                ;       metacall(B, D)
                )
        ).


% if - then - else

metacall(_ -> T; E, D) :- T == E, !, metacall(T, D).
metacall(C -> T; E, D) :- !,
        (       metacall(C, D_for_C)
        ->      (       D_for_C == []
                ->      metacall(T, D)
                ;       D = [D_for_C -> T; E]
                )
        ;       metacall(E, D)
        ).

metacall(C -> T, D) :- !, metacall(C -> T; fail, D).


% negation
metacall(nonmember(E, L), D) :- !, metacall(not member(E, L), D).
metacall(once(T), D) :- !, metacall(T, D).
metacall(call(T), D) :- !, metacall(T, D).
```

```
metacall(\+ T, D) :- !, metacall(not T, D).
metacall(~ T, D) :- !, metacall(not T, D).
metacall(fail_if(T), D) :- !, metacall(not T, D).
metacall(not G, []) :- not metacall(G, _), !.
metacall(not G, []) :- ground(G), !, fail.
metacall(not G, [not G]) :- !.


% subcall/2
metacall(subcall(G, _), _) :- not metacall(G, _), !, fail.
metacall(subcall(G, D), []) :- var(D), !, metacall(G, D).
metacall(subcall(G, D), []) :-
        metacall([G], D0), msort(D, D1), variant(D0, D1).
metacall(subcall(G, D), [subcall(G, D)]) :- !.


% user-defined predicate
metacall(P, D) :-
        is_userdef_predicate(P), !,
        (       ground(P)
        ->      (P, D = [])
        ;       (       deterministic(P)
                ->      (clause(P, B), metacall(B, D))
                ;       D = [P]
                )
        ).

deterministic(P) :-
        findall(_, clause(P, _), S),
        (S == []; S = [_]).


% builtin-predicate
metacall(P, D) :- !,
        (       delays(P)
        ->      D = [P]
        ;       (P, D = [])
        ).


% when to delay predicates
delays(T =.. L) :- var(T), var(L).
delays(append(A, B, C)) :-
        ((var(A), var(B)); (var(A), var(C))).
delays(delete(A, B, C)) :-
        ((var(A), var(B)); (var(A), var(C)); (var(B), var(C))).
delays(checklist(P, L)) :- (var(P); var(L)).
delays(flatten(N, _)) :- var(N).
delays(intersection(_, _, _)).
delays(length(L, N)) :- var(L), var(N).
delays(maplist(P, O, N)) :- (var(P); (var(O), var(N))).
delays(member(_, _)).
delays(subset(S, L)) :- (var(S); var(L)).
delays(subtract(K, L, _)) :- (var(K); var(L)).
delays(union(K, L, _)) :- (var(K); var(L)).
delays(bagof(_, _, _)).
delays(coverof(_, _, _)).
```

```
delays(findall(_, _, _)).
delays(setof(_, _, _)).
delays(sort(L, _)) :- var(L).
delays(sort(K, O, R, _)) :- (var(K); var(O); var(R)).
delays(instance(A, B)) :- not (ground(A), ground(B)).
delays(A @=< B) :- A \== B, not (ground(A), ground(B)).
delays(A @< B) :- A \== B, not (ground(A), ground(B)).
```

```
%------------------------------------------------------------------------------
% List and Conjunction Support Predicates


select([], N, [], []) :- !, N = 0.
select(L, 0, [], L) :- !.
select(L, N, [X | T], R) :-
        delete(X, L, L0),
        M is N - 1,
        select(L0, M, T, R).


enumeration(From, From, [From]) :- !.
enumeration(From, To, [From | Tail]) :-
        N is From + 1, enumeration(N, To, Tail).


for_one(P, [H | _]) :- apply(P, [H]), !.
for_one(P, [_ | T]) :- for_one(P, T).


filter(P, L, F) :- filter(P, L, F, _).

filter(P, L, F) :- filter(P, L, F, _).
filter(_P, [], [], []) :- !.
filter(P, [E | T0], [E | T1], L) :-
        apply(P, [E]), !, filter(P, T0, T1, L).
filter(P, [E | T0], L, [E | T1]) :-
        filter(P, T0, L, T1).


list2conj([], true) :- !.
list2conj([X], X) :- !.
list2conj([H | T0], (H, T1)) :- list2conj(T0, T1).


conj2list(true, []) :- !.
conj2list((H, T0), [H | T1]) :- !, conj2list(T0, T1).
conj2list(X, [X]).


member(P, X, [Y | _]) :- apply(P, [X, Y]).
member(P, X, [_ | T]) :- member(P, X, T).

% for use with checklist, etc.
amember(P, [Y | _], X) :- apply(P, [X, Y]).
amember(P, [_ | T], X) :- amember(P, T, X).


% append for conjunctions
cappend((A, B), C, (A, D)) :- cappend(B, C, D), !.
cappend(A, B, (A, B)).
```

```
%-------------------------------------------------------------------------------
% Output Support Predicates


/*
name_variables(+Term)

Names the free variables appearing in Term.
Replacement for numbervars/3.
*/

name_variables([], _Names) :- !.
name_variables(_Vars, []) :- !. % not enough names
name_variables([Var | Vars_tail], [Name | Names_tail]) :-
        Var = Name,
        name_variables(Vars_tail, Names_tail).

name_variables(Term) :-
        term_variables(Term, Vars),
        name_variables(Vars,
                ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L",
                 "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X",
                 "Y", "Z",
                 "AA", "AB", "AC", "AD", "AE", "AF", "AG", "AH", "AI", "AJ",
                 "AK", "AL", "AM", "AN", "AO", "AP", "AQ", "AR", "AS", "AT",
                 "AU", "AV", "AW", "AX", "AY", "AZ"]).


/*
log(loginfo(+Classes, +Filename, +Stream), +Class, +Format, +Arguments)

log prints the message specified by Format and Arguments on stream Stream
if the message class Class occurs in the list Classes.
*/

log(loginfo(Classes, _, Stream), Class, Format, Arguments) :-
        member(Class, Classes), !,
        nprintf(Stream, Format, Arguments),
        flush(Stream).

log(_, _, _, _).


/*
nprintf(+Stream, +Format, +Arguments)

Prints a formatted string specified by Format and Arguments on Stream.

Format is a list containing strings and directives:
- nl begins a new line.
- t prints a term from Arguments. Term may also be a string.
- flush performs a flush operation on Stream.

Objects are taken from Arguments in the order they are needed.
*/

nprintf(_, [], _) :- !, true.
```

```
nprintf(Stream, [t | Format_tail], [Term | Arguments_tail]) :- !,
#ifdef SWI
        write(Stream, Term),
#endif
#ifdef ECLIPSE
        /*
        + use portray/1,2 if defined (default)
        + print metaterm attributes using user-defined handlers (default)
        + print variables as unique numbers (don't confuse the user!)
        */
        printf(Stream, "%Pmvw", [Term]),
#endif
        nprintf(Stream, Format_tail, Arguments_tail).

nprintf(Stream, [nl | Format_tail], Arguments) :- !,
        nl(Stream),
        nprintf(Stream, Format_tail, Arguments).

nprintf(Stream, [flush | Format_tail], Arguments) :- !,
        flush(Stream),
        nprintf(Stream, Format_tail, Arguments).

nprintf(Stream, [String | Format_tail], Arguments) :- !,
        write(Stream, String),
        nprintf(Stream, Format_tail, Arguments).
```

```
%------------------------------------------------------------------------------
% Miscellaneous


/*
list_constraint_symbols(?Name/?Arity)

Prints a list of constraint specifications.
The constraint specification may be used as a filter.
*/

list_constraint_symbols(Name/Arity) :-
        is_userdef_constraint_symbol(Name/Arity),
        write(Name/Arity), nl,
        fail.

list_constraint_symbols(_).




/*
complete_constraint_specification(+Name/-Arity)

Succeeds if there is only one constraint with name Name. In this case the
arity of the constraint is returned in Arity. Else prints a completion list
and fails.
*/

complete_constraint_specification(Name/Arity) :-
        one_solution(is_userdef_constraint_symbol(Name/Arity)),
        is_userdef_constraint_symbol(Name/Arity), !.

complete_constraint_specification(Name/Arity) :-
        not is_userdef_constraint_symbol(Name/Arity), !,
        write("The constraint you specified is unknown."), nl,
        write("However, I have definitions for:"), nl,
        list_constraint_symbols(_/_),
        fail.

complete_constraint_specification(Name/Arity) :-
        write("Your constraint specification is not sufficient."), nl,
        write("I have definitions for:"), nl,
        list_constraint_symbols(Name/Arity),
        fail.




/*
rule_geq(+Rule1, +Rule2)

Copies the rules, assigns each free variable a dummy value and succeeds if
the copy of Rule1 is greater or equal than the copy of Rule2 in the standard
order of terms. Introduces a kind of order on rules.
*/
```

```
rule_geq(Rule1, Rule2) :-
        copy_term(Rule1-Rule2, Rule1_copy-Rule2_copy),
        term_variables(Rule1_copy-Rule2_copy, Vars),
        checklist(setvar(dummy), Vars),
        Rule1_copy @>= Rule2_copy.

% little helper
setvar(Y, Y).




is_userdef_constraint_symbol(Name/Arity) :- !,
        constraints(C),
        member(Name/Arity, C).


is_userdef_constraint(A) :-
        functor(A, Name, Arity),
        is_userdef_constraint_symbol(Name/Arity).


is_userdef_predicate_symbol(Name/Arity) :-
        (platform(eclipse) -> is_predicate(Name/Arity); true),
        not is_built_in_predicate_symbol(Name/Arity),
        not is_library_predicate_symbol(Name/Arity).


is_userdef_predicate(A) :- !,
        functor(A, Name, Arity),
        is_userdef_predicate_symbol(Name/Arity).


name_and_arity(A, Name/Arity) :- functor(A, Name, Arity).
name_and_arity(Name/Arity, A) :- functor(A, Name, Arity).


% supplement for is_built_in/1
is_library_predicate_symbol(Name/Arity) :-
        member(Name/Arity, [
                append/3, checklist/2, delete/3, flatten/2,
                intersection/3, length/2, maplist/3,
                member/2, memberchk/2, nonmember/2,
                subset/2, subtract/3, union/3,
                sort/2, sort/3, prune_instances/2]).


is_typetest_built_in_symbol(Name/Arity) :-
        member(Name/Arity, [atom/1, atomic/1, compound/1,
                integer/1, number/1, rational/1, real/1, string/1]).


has_local_variables(V, T) :-
        not only_global_variables(V, T).

only_global_variables(V, T) :-
        term_variables(T, TV),
```

```
        checklist(amember(==, V), TV).



/*
one_solution(+Goal)

Succeeds if Goal can be executed deterministically. Goal must be
executable!

How it works: Basically, the copy of G is called at most two times, then
one_solution/2 evaluates the facts satisfied_once/1 and satisfied_twice/2
asserted by satisfied/1. The failure driven loop is implemented by
satisfied/1, which will fail after the first satisfaction. Note that
safisfied/1 succeeds after the second satisfaction to avoid an infinite loop
for certain goals, e. g. append(A, B, C) with A, B, C variables. Further
note that Eclipse will not resatisfy a goal that has been delayed. Thus
Eclipse will remove the delayed goal. If Eclipse would resatisfy a goal that
has been delayed, the delayed goal would not be removed because satisfied/1
succeeds for the second satisfaction.
*/

% dynamic predicates for one_solution
:- dynamic satisfied_once/1, satisfied_twice/1.

one_solution(G) :-
        copy_term(G, G_copy), G_copy, satisfied(G), !,
        one_solution(decide, G).
one_solution(G) :- one_solution(decide, G).
one_solution(use_metacall, G) :-
        copy_term(G, G_copy), metacall(G_copy, _), satisfied(G), !,
        one_solution(decide, G).
one_solution(use_metacall, G) :- one_solution(decide, G).

one_solution(decide, G) :- not satisfied_once(G), !.
one_solution(decide, G) :- not satisfied_twice(G), !, retract(satisfied_once(G)).
one_solution(decide, G) :- !,
        retract(satisfied_once(G)), retract(satisfied_twice(G)),
        fail.


% record a satisfaction of the goal G

satisfied(G) :- not satisfied_once(G), !, assert(satisfied_once(G)), fail.
satisfied(G) :- not satisfied_twice(G), !, assert(satisfied_twice(G)).
```

```
%-------------------------------------------------------------------------------
% Package to run confluence tester on Eclipse

platform(eclipse).


ground(X) :- not nonground(X).


flush :- flush(user).


is_built_in_predicate_symbol(Name/Arity) :- is_built_in(Name/Arity).
```

```
%------------------------------------------------------------------------------
% Package to run confluence tester on SWI-Prolog


platform(swi).


% make SWI-Prolog recognize Prolog string objects
:- style_check(+string).


% operators used in CHR programs

% Eclispe stuff
:- op(700, xfx, ~=).
:- op(1200, xfx, if).
:- op(900, fy, ~).

% CHR stuff
:- op(1200, fy, handler).
:- op(1200, fy, constraints).
:- op(1200, fy, [label_with]).
:- op(1190, xfx, [==>, <=>]).
:- op(1200, xfx, :--).
:- op(1200, xfx, @).
:- op(1100, xfx, |).
:- op(1100, xfx, \ ).
:- op(700, xfx, flag).


% nothing like is_builtin is available in SWI-Prolog
is_built_in_predicate_symbol(Name/Arity) :-
        member(Name/Arity, [
                fail/0, false/0, true/0,
                (not)/1, (\+)/1, (~)/1, fail_if/1,
                call/1, once/1, subcall/2,
                !/0, ;/2, (->)/2,
                atom/1, atomic/1, compound/1,
                integer/1, number/1, real/1, rational/1, string/1,
                nonground/1, ground/1, var/1, nonvar/1,
                (==)/2, (\=)/2, (\==)/2, (~=)/2, (=)/2,
                (@<)/2, (@=<)/2, (@>)/2, (@>=)/2,
                instance/2, variant/2,
                (is)/2, (>)/2, (<)/2, (=<)/2, (>=)/2, (=:=)/2,
                (=..)/2,
                append/3, checklist/2, delete/3, flatten/2, last/2,
                intersection/3, length/2, maplist/3,
                member/2, memberchk/2, nonmember/2,
                subset/2, subtract/3, union/3,
                sort/2, sort/3,
                findall/3, bagof/3, coverof/3, setof/3]).


% implementation of Eclipse (*->)/2 (soft cut operator :-)
% soft cut will appear in release 2.7.14
:- op(1100, xfy, *->).
(A *-> _ ; C) :- \+ A, !, C.
```

```
(A *-> B ; _) :- A, B.


% implementation of Eclipse min/3
min(A, B, C) :- C is min(A, B).


% implementation of Eclipse delete/3
:- redefine_system_predicate(delete(_, _, _)).
delete(X, [X | T], T).
delete(X, [Y | T0], [Y | T1]) :- delete(X, T0, T1).


% implementation of Eclipse numbervars/3
numbervars(Term, From, To) :- numbervars(Term, '$VAR', From, To).


% implementation of Eclipse flush/1
flush(Stream) :- flush_output(Stream).


% implementation of Eclipse term_variables/2
term_variables(T, V) :- free_variables(T, V).


% implementation of Eclipse variant/2
variant(T, U) :- T =@= U.


/*
implementation of Eclipse instance/2
Eclipse performs occur check!
*/
instance(T, U) :-
        copy_term(T, V),
        copy_term(U, W),
        free_variables(V, FV_V),
        V = W,
        checklist(var, FV_V),
        free_variables(W, FV_W),
        length(FV_V, N),
        length(FV_W, M),
        N =< M.


% implementation of Eclipse retract_all/2
retract_all(Term) :- retract(Term), fail.
retract_all(_Term).


% implementation of Eclipse concat_atoms/3
concat_atoms(A, B, C) :- concat(A, B, C).
```