Implementierung eines kontrollmengengesteuerten Default Schlußfolgerungsmechanismus mit bedingten Klauseln

Diplomarbeit

von

Michael Grabarczyk

Professur für Künstliche Intelligenz

Prof.Dr.M.Schmidt-Schauß

Johann Wolfgang Goethe-Universität

Frankfurt am Main, Juli 1994

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Aufgabenstellung dieser Diplomarbeit sieht die Implementierung eines Deduktionssystems im Bereich des Default Reasoning vor.
Dieses Deduktionssystem entspricht der Implementierung des Kontrollmengengesteuerten Resolutionsprinzips in Form des Prototypen *KoResB* (*KoResB* steht für Kontrollmengengesteurter Resolutions Beweiser). Das Kontrollmengengesteuerte Resolutionsprinzip wurde in [Panitz,1993] vorgestellt.
Mit dem Begriff des Default Reasoning - ins Deutsche als „vages Schließen" übertragen - bezeichnet man im Bereich der Künstlichen Intelligenz (kurz: KI) alle Folgerungsmethoden, die über das monotone Schließen hinausgehen. Die Motivation im Gebiet der Künstlichen Intelligenz ist es hierbei, beliebige Sachverhalte darzustellen und in einem „hohen level" - sprich intelligent, also auch kreativ und flexibel - mit diesem Sachverhalt umzugehen.
Diese Kreativität und Flexibilität ist vor allem dann gefragt, wenn nur unvollständige - oder im Extremfall sogar gegensätzliche - Informationen über den ausgewählten Problembereich existieren. Und genau dies ist in den meisten der Problembereiche wie sie im täglichen Leben auftreten der Fall: Aufgrund zu komplexer Zusammenhänge ist eine vollständige Erfassung der problemspezifischen Daten nicht möglich. Man weiß zwar, daß alle Vögel fliegen, aber einige fliegen eben doch nicht.

Der Prototyp *KoResB* ist in Gofer geschrieben, und Gofer wiederum ist an die funktionale PS Haskell angelehnt. Das Ziel funktionaler PS ist die Loslösung des vom „von Neumannschen Flaschenhals" dem Programmierer aufgezwungene „Pipeline Programming“. Aufgrund des streng Funktionalen Aufbaus ist ein „funktionales Programm“ leicht modulierbar und die einzelnen Module auch im Sinne mathematischer Prinzipien verifizierbar.

In der vorliegenden Diplomarbeit beschäftige ich mich zunächst ganz allgemein mit Fragen des Automatischen Schließens und diskutiere verschiedene in diesem Zusammenhang durchgeführte Arbeiten.
So kläre ich als erstes in Kapitel 2 den Begriff des Deduktionssystems und gehe im weiteren auf das Schichtenmodell ein. Von besonderem Interesse ist hierbei das klassische Robinsonsche Resolutionsprinzip, da dieses Resolutionsprinzip modifiziert in *KoResB* Anwendung findet.

Nachdem ich Einblick in das Feld der Deduktionssysteme und deren spezifischer Problematik gegeben habe, wende ich mich dem Gebiet der Default Reasoning Systeme zu.
Die in Kapitel 3 gesetzten Schwerpunkte sind neben einer Motivation des Nichtmonotonen Schließens die zu dieser Thematik durchgeführten Arbeiten von Poole und Reiter.

Ziel des nächsten Kapitels ist es, verschiedene Constraint Klauselkonzepte vorzustellen.

In Kapitel 5 formuliere ich die theoretischen Grundlagen des von mir in der funktionalen Programmiersprache Gofer geschriebenen Prototypen *KoResB*. Schlüsselbegriffe dieses Kapitels sind Constraint Klauseltheorie, Konsistenzeigenschaft einer Constraint Klauseltheorie, Kontrollmenge, Constraint Resolutionsprinzip der Kontrollmengengesteuerten Constraint Resolution und Kontrollmengengesteuerte Constraint Resolution.

Im folgenden Kapitel stelle ich *KoResB* vor und gebe das abschließende Resultat der Umsetzung der theoretischen Grundlagen der Kontrollmengengesteuerten Constraint Resolution in *KoResB*.

Kapitel 2

Deduktionssysteme

In diesem Kapitel beschäftige ich mich ganz allgemein mit dem Gebiet der Deduktionssysteme, indem ich eine kurze Einleitung über einige zugrundeliegende Begriffe und Konzepte gebe.
Zu Beginn dieser Einleitung (Absch.1.1) steht der Begriff des Schichtenmodells für Deduktionssysteme. Das Schichtenmodell umfaßt vier Schichten, die ich im folgenden näher betrachte. Als erste Schicht wähle ich hierbei die Logikschicht (Absch.1.2). Die hier beschriebene Logik ist die in Anwendungen der künstlichen Intelligenz (KI) dominierende Prädikatenlogik erster Stufe (PL1), deren Ursprung in Freges Begriffsschrift wurzelt.
Die Kalkülschicht ist gleichbedeutend mit den Kalkülregeln, die zur Verfügung stehen. Im Abschnitt 2.3 beschreibe ich das Robinsonsche Resolutionskalkül. In den meisten Kalkülen tauchen zumindest Varianten der beiden Robinsonschen Regeln Resolutionsschritt und Faktorisierung auf.
In den letzten beiden Abschnitten behandle ich die Schichten Repräsentation und Steuerung zusammen. Diese Schichten sind besonders stark voneinander abhängig.

2.1 Das Schichtenmodell

Die Erforschung und Entwicklung von Deduktionssystemen gehört zu den Grundlagendisziplinen der KI. Hierbei wird ein Deduktionssystem ganz allgemein als ein programmierbares Verfahren angesehen, das aus vorhandenen Daten Schlussfolgerungen zieht und damit neue Formeln herleitet. Der Begriff Deduktionssystem impliziert das Vorhandensein von logischen Schlussfolgerungskomponenten. Ein Programm welches nur arithmetische Anweisungen ausführen kann wird somit nicht als Deduktionssystem betrachtet. Die beiden Vorraussetzungen für die Realisierung von Deduktionssystemen waren zum einen leistungsfähige elektronische Rechner und zum anderen mechanisierbare Logikkalküle. Zu Beginn der fünfziger Jahre realisierte Davis das erste Deduktionssystem. Sein Programm wurde von einem „Johniac“ bearbeitet, der zur Generation der Röhrencomputer zählt. Es war ein grosser Augenblick, als eine Maschine erstmalig für eine logische Anwendung genutzt werden konnte. Der erste von einem Computer durchgeführte Beweis war hierbei der Nachweis, das die Summe zweier gerader Zahlen ebenfalls gerade ist. Die Vorarbeit leistete Presburger [Presburger,1930], dessen Entscheidungsverfahren für die Arithmetik positiver Zahlen von Davis als Vorlage genutzt wurde.

Ein Deduktionssystem besteht im wesentlichen aus den Komponenten Logik, Kalkül, Repräsentation und Steuerung.

Abbildung 2.1: Schichtenmodell eines Deduktionssystems

Die Logik eines Deduktionssystems entspricht der zur Verfügung stehenden formalen Sprache. In ihr wird die zulässige Struktur (Syntax) und die Bedeutung (Semantik) der Aussagen definiert. Diese Aussagen sind gleichbedeutend mit den Formeln der Logik. Ein Deduktionssystem muß in der Lage sein, die von der Logik vorgegebenen Folgerungsrelationen zwischen den einzelnen Aussagen nachzubilden.

Im Logikkalkül werden nun losgelöst von der Bedeutung einzelner Symbole allein auf der Syntax basierende Ableitungsregeln (den Inferenzregeln) definiert. Kalküle werden aufgrund der Eigenschaften korrekt (sound) und vollständig (complete) bewertet. Ein Kalkül ist korrekt, wenn alle in ihm ableitbare Aussagen auch semantisch ableitbar sind. Können in einem korrekten Kalkül zusätzlich alle semantisch ableitbaren Formeln hergeleitet werden, so ist der Kalkül vollständig. Falls es für eine Logik einen vollständigen Kalkül gibt, so ist die Logik vollständig.

In der Repräsentationsschicht werden die Regeln zur Darstellung einzelner Formeln und Formelmengen festgelegt. Im dynamischen Ableitungsprozess des Deduktionssystems gibt die Repräsentation außerdem eine Darstellung der Beziehungen der Objekte züinander und eine Bewertung des Systemzustands. Vorhandene Reduktionsregeln werden auf Anwendbarkeit geprüft und gegebenenfalls durchgeführt.

Die Steuerung entspricht den Strategien und Heuristiken nach denen die Priorität der einzelnen Ableitungsschritte bestimmt wird. Falls das Deduktionssystem konfluent ist, wird eine nicht revidierende Steuerungsschicht gewählt. Die Konfluenz eines Systems bedeutet, dass das Lösungsverhalten aller auf einen beliebigen Systemzustand folgenden Alternativen identisch ist und somit beliebig zwischen den alternativen Ableitungsmöglichkeiten gewählt werden kann.

2.2 Prädikatenlogik

Die in Logikanwendungen vorherrschende Prädikatenlogik erster Stufe (PL1) ist eine Weiterentwicklung der von G.Boole entwickelten Aussagenlogik. Frege gilt aus heutiger Sicht als Wegbereiter der PL1. Er entwickelte in seiner Begriffsschrift [Frege,1879] eine graphisch orientierte Syntax für PL1 unter der Prämisse einer Aussagenlogischen Interpretation. Als Inferenzregel benutzte er den Modus Ponens.

In diesem Abschnitt beschäftige ich mich mit den heute gebräuchlichen Darstellungen der Syntax und Semantik von PL1. Diese Darstellungen sind Ergebnisse aus den Arbeiten von [Peano,1889] und [Tarski,1953].

2.2.1 Die Syntax von PL1

Die heute gebräuchliche Syntax der PL1 stammt von [Peano,1889]. Die Syntax einer Logik bestimmt die Regeln, nach denen Formeln in dieser Logik aufgebaut werden. Die einzelnen PL1 Formeln erhalten ihre syntaktische Definition wiederum durch Ableitung von einer zunächst festgelegten Signatur und den darauf basierenden Termen.

Definition 2.1 (Signatur) *Eine* Signatur Σ *ist ein Tupel* $\Sigma = (F,P)$ *bestehend aus den disjunkten Mengen F und P. Hierbei ist F eine Menge von Funktionssymbolen und P eine Menge von Prädikatensymbolen. Funktionssymbole und Prädikatsymbole besitzen eine Stelligkeit. Funktionssymbole der Stelligkeit 0 sind Konstante. Prädikatsymbole haben eine Stelligkeit von mindestens 1.*

Bemerkung: In der Signatur einer Logik werden die zur Verfügung stehenden primitiven Datenobjekte, die Funktions- und Prädikatzeichen definiert.
Beispiel (eine Signatur Σ):
$\Sigma = (\{f_\delta, g, abs\}, \{<, >\})$
Die Funktions- und Prädikatensymbole der Signatur Σ verfügen alle über eine sogenannte Stelligkeit, die durch eine Stelligkeitsfunktion s bestimmt werden kann. Es gilt:
$s(f_\delta) = s(g) = 1, s(abs) = 2, s(<) = S(>) = 2$ [1]

Definition 2.2 (Term) *Sei* $\Sigma = (F,P)$ *eine Signatur und V eine abzählbare Menge von Variablen so ist die Menge aller Terme* T_Σ *über* Σ *definiert durch:*

- $V \in T_\Sigma$
- *alle Konstanten sind in* T_Σ
- $\{f(t_1, \ldots, t_n) | f \in F \wedge s(f) = n \wedge t_1, \ldots, t_n \in T_\Sigma\} \in T_\Sigma$
- *keine weiteren Elemente sind in* T_Σ *(Minimalität)*[2]

Beispiel: Verschiedene Terme in PL1 Syntax
$x, f_\delta(\epsilon), abs(g(x), g(y))$;
hierbei sind x und y Variable und ϵ ist eine Konstante (also s(ϵ) = 0).

Definition 2.3 (Formel) *Sei* $\Sigma = (F,P)$ *eine Signatur. Die Menge* F_Σ *aller Formeln über* Σ *ist definirt durch:*

[1] Die Stelligkeitsfunktion s mit $s : \{F, P\} \mapsto \mathbb{N}$ gibt losgelassen auf ein Funktionssymbol aus F oder eine Prädikatensymbol aus P dessen Stelligkeit an.

[2] durch diese Regel wird die Minimalität und Korrektheit der Terme gesichert

- $\{R(t_1,\dots,t_n)|R\in P\wedge s(R)=n\wedge t_1,\dots,t_n\in T_\Sigma\}\in F_\Sigma$
- $A\in F_\Sigma \Rightarrow \neg A\in F_\Sigma$
- $A,B\in F_\Sigma \Rightarrow (AopB)\in F_\Sigma$, *mit:*
$op\in\{\wedge,\vee,\rightarrow,\leftrightarrow\}$
- $A\in F_\Sigma \Rightarrow \forall A\in F_\Sigma$
- $A\in F_\Sigma \Rightarrow \exists A\in F_\Sigma$
- *keine weiteren Objekte sind in A (Minimalität)*

Beispiel: Eine Formel in PL1 Syntax
$\forall\epsilon(>(\epsilon,0)\Rightarrow\exists\delta(>(\delta,0)\wedge\forall x,y(<(abs(x,y),\delta)\Rightarrow<(abs(g(x),g(y)),\epsilon)))$

2.2.2 Die Semantik von PL1

Die heute übliche Methode um syntaktisch korrekten Termen und Formeln eine semantische Bedeutung zu verleihen, basiert auf einer Veröffentlichung von Tarski [Tarski,1953]. Tarski benutzte hierbei eine Abbildung der vorhandenen syntaktischen Objekte auf geeignete mathematische Objekte. In Anwendungen der KI geht man oft die umgekehrte Richtung: Ein Ausschnitt der realen Welt wird hierbei auf entsprechende syntaktische Konstrukte abgebildet. Die drei in diesem Zusammenhang auftretenden Schlüsselbegriffe Σ-Algebra, Σ-Struktur und Interpretation werden in den folgenden Definitionen behandelt.

Definition 2.4 (Σ-Algebra) *Für eine Signatur Σ=(F,P) ist* $\mathcal{A}_\Sigma=(D_\Sigma,F_\Sigma)$ *eine Σ-Algebra gdw.*

- D_Σ *ist eine nichtleere Menge (Trägermenge von A)*
- F_Σ *enthält für jedes* $f\in F$ *eine totale Funktion* [3] f_n *, so dass*
$F_\Sigma=\{f_n: D_\Sigma^n\rightarrow D_\Sigma|f\in F\wedge s(f)=n\}$

Definition 2.5 (Σ-Struktur) *Für eine Signatur Σ=(F,P) ist* $\mathcal{S}_\Sigma=(D_\Sigma,F_\Sigma,P_\Sigma)$ *eine Σ-Struktur gdw.*

[3] eine Funktion f heißt total, wenn sie für jedes Element des Argumentbereichs einen Wert liefert.

- (D_Σ, F_Σ) *ist eine* Σ*-Algebra*
- P_Σ *enthält für jedes* $p \in P$ *eine Relation* $p_\Sigma \subseteq D_\Sigma^n$

Definition 2.6 (Interpretation) *Für eine Signatur* Σ*=(F,P) und eine Menge von Variablen V ist* $\mathcal{I} = (\mathcal{S}, \mathcal{V})$ *eine Interpretation, wenn* $\mathcal{S}$ *eine* Σ*-Struktur ist und* $\mathcal{V}$ *einer Abbildung* $\mathcal{V} : V \mapsto D_\mathcal{S}$ *entspricht, die jeder Variablen aus V einen Wert aus* $D_\mathcal{S}$ *zuordnet.* $D_\mathcal{S}$ *ist die Trägermenge von* $\mathcal{S}$*.*

Eine geschlossene Formel [4] kann unter einer ihr zugeordneten Interpretation als Aussage über den semantischen Problemkreis angesehen werden und besitzt einen der beiden *Wahrheitswerte* WAHR oder FALSCH. Eine Interpretation I heisst *Modell* für eine Formelmenge F, falls jede Formel aus F unter der Interpretation I den Wahrheitswert WAHR annimmt. Eine Formel F heisst *erfüllbar*, wenn sie ein Modell hat. Eine Formel F ohne Modell ist *unerfüllbar*. Eine Formelmenge F heisst *allgemeingültig*, wenn deren Negation ¬F unerfüllbar ist.

Definition 2.7 (logische Folgerung) *Eine Formel F folgt logisch aus einer Formelmenge* $\mathcal{F}$ $(\mathcal{F} \models F)$*, falls jedes Modell für* $\mathcal{F}$ *auch ein Modell für F ist.*

Bemerkung: Eine logische Folgerung wird auch semantische Folgerung genannt.

2.3 Robinsons Resolutionskalkül

In vielen Deduktionssystemen werden Varianten des von Robinson in seiner 1965 erschienenen Arbeit „A machine oriented logic based on the resolution principle“ vorgestellten Resolutionskalküls benutzt [Robinson,1965].

Mit Hilfe des Resolutionskalküls ist es möglich, die Allgemeingültigkeit einer beliebigen PL1 Formel nachzuweisen. In diesem Sinne wird das Resolutionsprinzip als Beweiskalkül für PL1 Formeln angesehen. Ein Beweisalkül heißt negativ, wenn die Allgemeingültigkeit einer Formel F durch Widerlegung des Negats der Formel F gezeigt wird. Ein Kalkül heißt Testkalkül, wenn man von der zu überprüfenden Formel

[4] Eine Formel die keine freien Variablen enthält, ist eine geschlossene Formel. Eine Variable ist frei, wenn sie von keinem Quantoren gebunden wird

durch Anwendung der Inferenzregeln zu den anfangs definierten Axiomen gelangen will. Da beide Eigenschaften auf das Resolutionsprinzip zutreffen, handelt es sich beim Resolutionsprinzip um einen negativen Testkalkül. Das Resolutionskalkül verwendet die Inferenzregeln Resolutionsschritt und Faktorisierung (Absch. 2.3.3). Ziel ist es, den fundamentalen Widerspruch in Form der leeren Klausel herzuleiten.

Bevor die Inferenzregeln auf die zu widerlegende Formel angewendet werden, wird diese zunächst in die Klauselnormalform (kurz: KNF) [Davis,1963 und Chinlund et al.,1964] gebracht (Absch. 2.3.1). Um einen Inferenzschritt auf den einzelnen Elementen der KNF - den Klauseln - auszuführen ist es im allgemeinen notwendig, die Frage nach der Unifizierbarkeit zweier Terme (Absch. 2.3.2) zu klären. Gelingt es durch Anwendung der Inferenzregeln die leere Klausel abzuleiten, so gilt die zu widerlegende Formel als widerlegt. Falls ein Resolutionsbeweis terminiert, dann entweder mit der Herleitung der leeren Klausel oder wenn keine neuen Ableitungen mehr gemacht werden können. Da PL1 nicht mehr entscheidbar ist [Church, 1936], terminiert der Robinsonsche Resolutionskalkül für PL1 nicht zwingend; er terminiert aber mit der leeren Klausel für alle allgemeingültigen PL1 Formeln. Er ist sozusagen ein lebender Beweis für die Vollständigkeit von PL1 [Gödel,1930].

2.3.1 Klauselnormalform

Die Klauselnormalform und die Klauselmengendarstellung sind nur zwei von vielen Normalformen, die es für PL1 Formeln gibt. Sollen Formeln mit einem Resolutionsbeweiser bearbeitet werden, so werden diese Formeln gewöhnlich in die Klauselmengendarstellung gebracht, bevor das Resolutionsprinzip angewendet werden kann. Für diesen vorbereitenden Schritt gibt es verschiedene Transformationsalgorithmen, die beliebige PL1 Formeln in beliebiger Darstellung in die Klauselform bringen. Der Aufwand einer solchen Transformation benötigt im Standardverfahren exponentiellen Operationsaufwand.

Ich behandle nun als erstes die Definitionen der für den Aufbau der Klauselmengendarstellung benötigten Begriffe Atom, Literal und disjunktive Klausel. Als nächstes zeige ich einige Darstellungsvarianten für ein und dieselbe PL1 Formel.

Definition 2.8 (Atom) *Ein Atom ist ein Konstrukt $P(t_1, \ldots, t_n)$ wobei P ein Prädikatensymbol der Stelligkeit n ist und t_i ein Term.*

Beispiel:
Unter Verwendung der beiden zweistelligen Prädikatsymbole $<$ und $>$ lassen sich zum Beispiel die Atome $<(a,b)$ und $>(g(a),a)$ bilden.

Definition 2.9 (Literal) *Es gibt positive und negative Literale. Ein positives Literal ist ein Atom ohne vorangestelltes $\neg$. Ein negatives Literal ist ein Atom mit vorangestelltem $\neg$.*

Definition 2.10 (Disjunktive Klausel) *Eine endliche Disjunktion K von Literalen L_i, also $K = L_1 \vee \ldots \vee L_n$ ist eine disjunktive Klausel.*[5] *Schreibt man K in der Notation $K = \{L_1, \ldots, L_n\}$, so wird K Klausel genannt.*

Beispiel: Disjunktive Klausel
$\neg$ *größer* $(\epsilon, 0) \vee$ *größer* $(\epsilon, 0)$

Definition 2.11 (Klauselnormalform) *Eine Klausel in Klauselnormalform (kurz: KNF) ist eine endliche Konjunktion $\mathcal{K}$ von disjunktiven Klauseln K_i: $\mathcal{K} = K_1 \wedge \ldots \wedge K_n$. $\mathcal{K}$ wird, wenn es als $\mathcal{K} = \{K_1, \ldots, K_n\}$ geschrieben wird Klauselmenge genannt.*[6]

Als Beispiel für verschiedene Darstellungsweisen von PL1 Formeln werde ich nun die in 2.2.1 eingeführte Formel (es handelt sich um die aus der Analysis bekannte Definition der Stetigkeit einer Funktion), ausgehend von der mathematischen Notation die noch Quantoren und Junktoren enthält, in drei äquivalenten Formelrepräsentationen aufführen.
Ausgehend von der mathematischen Darstellung
$\forall \epsilon > 0 \exists \delta > 0 \forall x, y : \mid x - y \mid < \delta \Rightarrow \mid \delta(x) - g(y) \mid < \epsilon$
werden im ersten Schritt die mathematischen Symbole $<$ und $>$ durch die zweistelligen Prädikatsymbole $<$ und $>$ ersetzt und zusätzlich noch einige Klammerungsveränderungen ausgeführt:
$\forall \epsilon (> (\epsilon, 0) \Rightarrow \exists \delta (> (\delta, 0) \wedge \forall x, y (< (\mid x - y \mid, \delta) \Rightarrow < (\mid g(x), g(y) \mid, \epsilon)))$
Im nächsten Schritt bleibt die Klammerung erhalten und das zweistellige natürlichsprachige Prädikat abs wird eingeführt.

[5]Der Vollständigkeit halber sei erwähnt, daß eine aus nur einem Literal bestehende disjunktive Klausel ebenfalls eine konjunktive Klausel ist.

[6]Eine einelementige Klauselmenge ist ebenfalls in KNF.

$\forall \epsilon \geq (\epsilon, 0) \Rightarrow \exists \delta \geq (\delta, 0) \wedge \forall x, y \leq (abs(x, y), \delta) \Rightarrow$
$kleiner\ (abs(g(x), g(y)), \epsilon)))$

Nun werden Allquantoren und Existenzquantoren aus der Formel genommen, was die Klammerung beeinflußt und zusätzlich - im Fall des Existenzquantoren - die Skolemfunktion f_δ einführt. Auf diee Weise erhält man die Klauselnormalform:

$(\neg größer(\epsilon, 0) \vee größer(f_\delta(\epsilon), 0)) \wedge$
$(\neg größer(\epsilon, 0) \vee kleiner(abs(x, y), f_\delta(\epsilon)) \vee kleiner(abs(g(x), g(y)), \epsilon)))$

Um aus dieser Klauselnormalform die Klauselform zu erhalten, ist es nur noch notwendig die logischen Verknüpfungen $\wedge$ und $\vee$ durch die entsprechenden Klammerungen zu ersetzen.
Die Klauselmengendarstellung der PL1-Formel für die Stetigkeit einer Funktion sieht somit folgendermaßen aus:

$\{\{\neg größer(\epsilon, 0), größer(f_\delta(\epsilon), 0)\},$
$\{\neg größer(\epsilon, 0), \neg kleiner(abs(x, y), f_\delta(\epsilon)), kleiner(abs((g(x), g(y)), \epsilon)\}\}$

2.3.2 Unifikation

Zwei Klauseln gelten als resolvierbar, wenn sie Literale mit unterschiedlichem Vorzeichen besitzen und deren Atome identisch sind oder deren Atome durch die simultane Anwendung einer geeigneten Substitution identisch gemacht werden können. Solche besonderen Substitutionen werden Unifikatoren genannt.

Definition 2.12 (Substitution) *Eine Substitution σ ist eine Abbildung $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. Es gilt $x_i \neq x_j$ und $x_i \notin t_i \forall i, j$. Ein einzelnes Paar $x_i \leftarrow t_i$ wird Zuweisung genannt. Hierbei ist x_i die Zuweisungsvariable und t_i der Zuweisungsterm. Die Substitution σ auf einen Ausdruck anwenden heißt, daß jede Zuweisungsvariable x_i im Ausdruck durch den zugehörigen Zuweisungsterm simultan t_i ersetzt wird.*

Definition 2.13 (Unifikator) *Seien A_1 bis A_n entweder Terme oder Atome. Eine Substitution σ heißt Unifikator für $\{A_1, \ldots, A_n\}$ falls $\sigma(A_1) = \ldots = \sigma(A_n)$.*

Wenn es darum geht einen Unifikator für zwei oder mehrere Terme beziehungsweise Atome zu bestimmen, wird nicht nur irgendein Unifikator, sondern ein allgemeinster Unifikator gesucht.

Definition 2.14 (Allgemeinster Unifikator) *Seien A_1 bis A_n entweder Terme oder Atome. σist ein allgemeinster Unifikator für $\{A_1, \ldots, A_n\}$, falls für alle Unifikatoren σ' für $\{A_1, \ldots, A_n\}$ eine Substitution σ" existiert, sodaß σ"$\sigma\{A_1, \ldots, A_n\} = \sigma'\{A_1, \ldots, A_n\}$ gilt.*

2.3.3 Robinsons Inferenzregeln

Im Mittelpunkt des Robinsonschen Resolutionskalküls steht der folgende Resolutionsschritt:

Definition 2.15 (Resolutionsschritt) *Seien K_1 und K_2 Klauseln mit $K_1 = \{L, M_1, \ldots, M_n\}$ und $K_2 = \{\neg L', N_1, \ldots, N_m\}$ und sei σ ein Unifikator für $\{L, L'\}$. So ist $\sigma(\{M_1, \ldots, M_n, N_1, \ldots, N_m\})$ Resolvente von K_1 und K_2. L und L' werden Resolutionsliterale, K_1 und K_2 werden Elternklauseln genannt. Das Bilden einer Resolvente aus zwei Elternklauseln wird als Resolutionsschritt bezeichnet.*

Dieser Resolutionsschritt ist eine Verallgemeinerung der klassischen logischen Schlußfolgerung, des Modus Ponens [7]. Zu den meistzitiertesten Anwendungsbeispielen des Modus Ponens zählt wohl folgender logischer Beweisschluß:

Alle Menschen sind Sterblich.
Sokrates ist ein Mensch.
$\Rightarrow$ Sokrates ist sterblich.

Die folgende Faktorisierungsregel kann auch in die Resolutionsregel integriert werden. Durch die Darstellung als eigenständige Regel wird allerdings die Abtrennung der

[7]Eine heute geläufige Notation der Syntax des Modus Ponens lieferte der Logiker G.Gentzen gen35. Er nannte den Modus Ponens „Folgt Beseitigung“ und integrierte ihn als $F, F \Rightarrow G \parallel G$ in seinen Schlußfolgerungsregeln.

Prinzipien „zwei Elternklauseln“ (Resolventenbildung) „eine Elternklausel“ (Faktorbildung) unterstrichen.

Definition 2.16 (Faktorisierung) *Sei $K = \{L, L', M_1, \ldots, M_n\}$ eine Klausel und σ ein Unifikator für $\{L, L'\}$. Dann ist $\sigma(\{L, M_1, \ldots, M_n\})$ Faktor von K. K wird Elternklausel genannt. Das Bilden eines Faktors wird Faktorisierung genannt.*

Als Inferenzregeln kennt der Robinsonsche Resolutionskalkül nur den Resolutionsschritt und die Faktorisierung.

Definition 2.17 (Robinsonsches Resolutionsprinzip) *Das Robinsonsche Resolutionsprinzip ist ein Kalkül bei dessen Anwendung auf eine gegebene Klauselmenge $\mathcal{K}$ es das Ziel ist, durch Anwendung der beiden Regeln Resolutionsschritt und Faktorisierung die leere Klausel herzuleiten.*

Die Sterblichkeit von Sokrates kann mit dem Resolutionskalkül bewiesen werden, indem die Klauselmenge $\mathcal{K}$ widerlegt wird, was in zwei Schritten möglich ist:

$$\begin{aligned}
&\mathcal{K} = \{K_1, K_2, K_3\}\\
&K_1 = \{Mensch(Sokrates)\},\\
&K_2 = \{\neg Mensch(x), Sterblich(x)\},\\
&K_3 = \{\neg Sterblich(Sokrates)\}\}
\end{aligned}$$ [8]

Der erste Resolutionsschritt mit K_1 und K_2 und Unifikator $\sigma_1 = \{x \leftarrow Sokrates\}$ ergibt

$$K_4 = \{sterblich(Sokrates)\}$$

Im zweiten Resolutionsschritt mit K_4 und K_3 und Unifikator $\sigma_2 = \{\}$ erhält man als Resolvente die leere Klausel

[8] $\neg sterblich(Sokrates)$ ist das Negat der zu beweisenden Aussage.

$K_5 = \square$.

Somit wurde die Aussage

> „Sokrates ist sterblich, weil alle Menschen sterblich sind und Sokrates ein Mensch ist“

als allgemeingültig nachgewiesen, da die Annahme das Sokrates unsterblich sein könnte zum Widerspruch (in Form der leeren Klausel) geführt werden konnte.[9]

2.3.4 Widerlegungsvollständigkeit des Resolutionskalküls

Die wichtigste Eigenschaft des Robinsonschen Resolutionskalküls (RRK) ist dessen Widerlegungsvollständigkeit. Die Widerlegungsvollständigkeit eines Kalküls besagt, daß jede widerlegbare Formel im Rahmen der im Kalkül vorgesehenen Mechanismen widerlegt werden kann.

Theorem 2.18 (Widerlegungsvollständigkeit des RRK) *Für jede unerfüllbare Klauselmenge $\mathcal{K}$ kann mittels einer geeigneten Steuerungsstrategie, welche die Anwendung der beiden Inferenzregeln Resolutionsschritt und Faktorisierung bestimmt, die leere Klausel hergeleitet werden.*

Die Vollständigkeit beziehungsweise Widerlegungsvollständigkeit eines Kalküls für PL1 entspricht dem maximalen Resultat, das ein Kalkül für PL1 leisten kann, da PL1 vollständig (rekursiv aufzählbar) jedoch nicht entscheidbar (rekursiv) ist. [10]

Im folgenden werden die Begriffe isoliert, tautologisch und subsumiert geklärt, damit die zugehörigen Löschregeln definiert werden können:

[9] Eine Sonderform des Resolutionsprinzips, die unäre Resolution verlangt, das bei jedem Resolutionsschritt mindestens eine Elternklausel einelementig (unär) sein muß. Beweise die nur mit Modus Ponens Aussagen geführt werden können, können mit dieser „unären Resolution“ geführt werden.

[10] In der theoretischen Informatik sind die Begriffe rekursiv und rekursiv aufzählbar im Bereich der Sprachkomplexität zu finden. Aus diesem Blickwinkel betrachtet entsprechen alle in PL1 konstruierbaren Formeln den Wörtern, die über einem Alphabet Σ gebildet werden können. Eine Sprache L ist eine Teilmenge aller Wörter über dem Alphabet Σ. Die Frage, ob ein Wort w aus dem Komplement $\neg L$ ist, kann i.a. nicht entschieden werden.

Ein Literal L der Klauselmenge $\mathcal{K} = \{K_1, \ldots, K_n\}$ heißt *isoliert*, falls es nirgendwo in $\mathcal{K}$ eine Klausel K_i mit einem Literal L' gibt, so daß L und L' ein unterschiedliches Vorzeichen haben und deren Atome unifizierbar sind.
Eine Klausel $K = (L_1, \ldots, L_n)$ ist eine *Tautologie*, falls es zwei Literale L_i und L_j in K gibt, mit $L_i = \neg L_j$.
Eine Klausel K' wird von einer Klausel K *subsumiert*, falls es eine Substitution σ gibt, mit $\sigma(K) \subseteq K'$.

Theorem 2.19 (Widerlegungsvollständigkeit RRP mit LR) *Das Robinsonsche Resolutionskalkül zusammen mit den drei Löschregeln (LR)*

- *entferne Klauseln mit isolierten Literalen*
- *entferne tautologische Klauseln*
- *entferne Klauseln die subsumiert werden können*

ist widerlegungsvollständig.

2.4 Repräsentation

Bevor es das eigenständige Forschungsfeld der KI gab, beschäftigten sich verschiedene Wissenschaftler aus Linguistik, Philosophie und Psychologie mit den Fragen der Eigenschaften des menschlichen Sprechens und Denkens [Minsky,1975]. Wichtige Fragen betreffen in diesem Zusammenhang das Entstehen der menschlichen Sprachen, den Transferprozeß von Gedachtem zu Geäußertem und vice versa, sowie das „abspeichern“ von gemachten Erfahrungen im Gehirn. So wurde u.a. mittels Semantischer Netzwerke (SNW) [Woods,1975] versucht, natürlichsprachliche Sätze darzustellen.

Repräsentationsschichten in Deduktionssystemen stellen nicht den Anspruch natürlichsprachliche Sätze zu simulieren. Sie dienen zur Bereitstellung der Datenobjekte, auf denen die im Kalkül festgelegten Operationen ausgeführt werden. Zu solchen im Rahmen der Deduktionssysteme vertretenen Repräsentationsformen gehören:

- Klauselgraphen [Kowalski,1975]

- Konnektionsmethode [Bibel,1982]
- Tableaus [Smullyan,1968]

2.5 Steuerung

Beim „Bau " eines Deduktionssystems entscheidet man sich in aller Regel zunächst einmal für die dem Deduktionssystem zugrunde liegende Logik und anschließend für ein entsprechendes Kalkül. Somit sind die Grenzen, in denen sich die beiden noch ausstehenden Schichten Repräsentation und Steuerung noch bewegen können weitestgehend abgesteckt.
Für Repräsentation und Steuerung ergibt sich eine gegenseitige Abhängigkeit. Diese gegenseitige Abhängigkeit tritt bereits bei der Konzeption der beiden Schichten zu Tage. Hierbei gibt es verschiedene Ansätze. Man kann sowohl mit der Repräsentation als auch mit der Steuerung beginnen. Ein mögliches Verfahren ist hierbei eine 3-stufige wechselseitige Konstruktion beider Schichten. Erster Schritt ist hierbei das Entwickeln einer Idee für eine mögliche Steuerung. Mit dieser Steuerungsidee im Hinterkopf folgt die Entwicklung der Repräsentation und hierauf die Konkretisierung der Steuerung anhand der gefundenen Repräsentation. Dieser 3. Schritt beschäftigt sich aber keineswegs nur mit der Steuerung. Er stellt vielmehr eine „Feinjustierung" beider Schichten dar. Bei der Steuerungsdefinition können unter Umständen noch Verbesserungsmöglichkeiten der Repräsentationsschicht offengelegt werden, nach deren Umsetzung sich wieder Auswirkungen auf die Steuerungsdefinition ergeben und so fort. Eine Modifikation einer der beiden Schichten kann also eine Kette weiterer Möglichkeiten initiieren. Eine alternative Konzeption ergibt sich, wenn man sich endgültig für eine Darstellungsschicht entschieden hat und nur noch die Steuerung optimieren möchte. Hierbei gibt es im allgemeinen mehrere sinnvolle Steuerungsalternativen die alle über gewisse Vor- und Nachteile verfügen. Bei der Konzeption der Steuerungsschicht gibt es die Wahl zwischen einer Ordnungsstrategie oder einer Restriktionsstrategie. Hierbei wird die Entscheidung getroffen, ob die für die Kalküloperationen nötigen Objekte sachbezogen (Restriktionsstrategie) oder strukturbezogen (Ordnungsstrategie) ausgewählt werden. Bei einer Ordnungsstrategie spielt die Art und Weise der Objektverwaltung die bestimmende Rolle. Ein Beispiel hierfür ist die Ordnungsstrategie „Resolution in Breitensuche", bei der zu einer anfangs ausgewählten Klausel syste-

matisch die restlichen Klauseln gemäß deren Verwaltungsstruktur auf Anwendbarkeit geprüft werden. Anders verhält es sich bei Restriktionsstrategien. Hier erhalten die einzelnen Objekte, beziehungsweise Objektgruppen eine restriktionsabhängige Präferenz zur Bestimmung des Abarbeitungszeitpunktes. So können einzelne Objekte oder Objektgruppen von der momentanen Bearbeitung ausgeschlossen werden. Objekte mit entsprechender Restriktionskennung werden bevorzugt. Ein Beispiel hierfür ist die Restriktions-Strategie „Unäre Resolution“. Hierbei entscheidet die Restriktionskennung zwischen Klauseln mit nur einem Literal (unäre Klauseln) und Klauseln mit mehreren Literalen. Die unären Klauseln werden bevorzugt.

Kapitel 3

Nichtmonotones Schließen

Durch das Robinsonsche Resolutionsprinzip gibt es eine effektive Methode, um semantische Folgerungen syntaktisch nachzubilden. Ein neuer Impuls resultierte nun aus dem Wunsch, Beweisverfahren auf Situationen aus dem Alltagsleben zu übertragen. Im Gegensatz zu mathematischen Anwendungsgebieten gibt es im Alltagsleben eine Fülle von nichtmonotonen Sachverhalten. Um nun auch solche nichtmonotonen Sachverhalte in das Resolutionskalkül involvieren zu können, machte man sich auf die Suche nach geeigneten Mechanismen zur Lösung dieser Darstellungsfrage. Schlußfolgerungsverfahren die über solche Mechanismen verfügen, werden Default-Reasoning Systeme genannt. In solchen Systemen werden nichtmonotone Aussagen als Defaults gekennzeichnet.

Die nächste Überlegung ging nun nicht in Richtung eines geeigneten Kalküls für solche Defaults, sondern führte u.a. zur grundsätzlicheren Frage nach der zu benutzenden Logik, um die Defaults effektiv einsetzen zu können. So kam Reiter zu dem Schluß, daß dies nur unter Einsatz nichtmonotoner Logiken der Fall sei:

> „... any logic which presumes to formulize default reasoning must be non monotonic“ ([Reiter,1980]S.85).

Anstatt sich dieser Theorie anzuschliessen, entschied sich Poole bei der monotonen PL1 zu bleiben und lediglich den Kalkül dahingehend zu erweitern, daß er die Möglichkeit hypothetische Schlüsse zu ziehen unterstützt. Poole postulierte:

„ ... rather than being a problem with logic, nonmonotonicity is a problem of how logic is used“ ([Poole,1988],S.28).

In diesem Kapitel beschäftige ich mich mit der Fragestellung des nichtmonotonen Schließens und anderen, im Zusammenhang mit Darstellung und „Berechnung“ sogenannter Systeme der KI stehenden Fragestellungen. Nachdem in Abschnitt 3.1 für KI Systeme typische Merkmale aufgeführt werden, behandle ich im Abschnitt 3.3 den Begriff der Monotonie. Die hierauf folgenden beiden Abschnitte beschreiben Arbeiten von Reiter und Poole und geben deren Lösungsansätze für die Realisierung des nichtmonotonen Schließens.

3.1 Eigenschaften von Systemen in der KI

Problembereiche der KI Anwendungen sind weniger konstruiert, als vielmehr Ausschnitte einer „Anwendung“ wie sie in der „Realen Welt“ vorkommen kann. Die spezielle Problematik solcher Systeme der KI liegt in ihrer Komplexität begründet. Nur die wenigsten KI Systeme lassen sich vollständig charakterisieren. Typische Merkmale von Objekten oder Objektgruppen solcher Anwendungen sind:

- die Erfordernis flexibler, nichtmathematischer Mengenbezeichner wie „die meisten“,„fast alle“ oder „einige“.
- der auf dem Grundkonzept der WAHR - FALSCH Philosophie beruhende Dualismus, der als *closed world assumption* [Reiter,1978] bekannt ist. In einem System mit „closed world assumption“ müssen nur positive Informationen über den Anwendungsbereich aufgelistet werden. Die negativen Informationen können geschlußfolgert werden. Dies ist sinnvoll, da es oft wesentlich mehr negative als positive Informationen gibt. Als Beispiel stelle man sich die Darstellung einer Schulklasse mit einem Schüler Peter vor, dessen Vater der Klassenlehrer ist. Man könnte dies mit
  $SohnvomLehrer(Peter)$ als positive Information darstellen, und in einer „closed world assumption“ Regel der Form
  $\neg SohnvomLehrer(x) \parallel x \neq Peter$ für alle anderen Schüler ableitbar machen, daß sie nicht der Sohn des Lehrers sind.

- sogenannte *frame defaults*, die innerhalb von dynamischen Welten die jeweiligen gesetzmäßigen Invarianzen umschreiben. So ändert sich beispielsweise niemals die Farbe eines normalen Telefons, wenn daß Licht ausgeht.

Als weitere Probleme solcher Systeme der KI sind noch die Darstellung von Zeitabhängigkeiten und unvollständigem Wissen zu nennen. Als Darstellungsmöglichkeit hierfür betrachte ich in **??** die Constraint Klauseln.

3.2 Monotonie

Systeme der KI sollten i.A. die Eigenschaft haben, daß sie innerhalb einer Anwendungsbearbeitung durch neue Einzelheiten ergänzt werden können. Durch solche Ergänzungen mit neuen Fakten und Regeln kann es der Fall sein, daß bisher gültige Folgerungen nun nicht mehr gültig sind. Ein KI System sollte also jederzeit in der Lage sein, flexibel auf Ergänzungen der Datenobjekte reagieren zu können. Falls eine Logik über diese Flexibilität verfügt, spricht man von einer nichtmonotonen Logik. [1]

Definition 3.1 (Nichtmonotone Logik) *Seien $\mathcal{A}, \mathcal{B}$ Formelmengen einer Logik $\mathcal{L}$ und F eine Formel. $\mathcal{L}$ heißt nichtmonoton,*[2] *falls* $(\mathcal{A} \models F) \not\Rightarrow (\mathcal{A} \cup \mathcal{B} \models F)$

3.3 Reiters Default Logic

In seiner Arbeit „ A Logic for Default Reasoning“ [Reiter,1980] liefert Reiter sein Konzept der Default Logic (DL). Motivation der DL ist die Unterstützung von nichtmonotonem Schließen. In den einzelnen Teilen dieses Abschnitts gebe ich eine Zusammenfassung dieser DL.

Die syntaktischen Konstrukte Reiters entsprechen zum einen der üblichen PL1 Syntax (Absch. 2.2)und zum anderen den als Defaults bekannten Regeln, mit denen prinzipiell ein im Sinne der Nichtmonotonie-Eigenschaft dynamisches Schließen ermöglicht

[1] Systeme die diesen Gedanken unterstützen, werden „belief revision Systems“ oder Default Reasoning Systems genannt.

[2] Eine Logik heißt monoton, falls durch hinzunahme beliebiger Annahmen nicht weniger folgt, als vor hinzunahme dieser Annahmen.

wird.[3] Diese Datenobjekte: Fakten (in Form von bekannten PL1 Formeln) und Regeln (in Form von Defaults bilden nun auch die Berechnungsgrundlage, anhand derer die verschiedenen „Beliefs“ und hierauf die jeweiligen Extensionen (Absch.3.3.2) bestimmt werden können.

Soll eine Formel (Aussage) β in einer Default Theorie Δ bewiesen werden, so muß innerhalb eines Beweises eine Extension E definiert werden, in der die Formel β konsistent eingebettet ist (Absch3.3.3).

3.3.1 Default und Default Theorie

Die für sämtliche Folgerungsableitungen zugrundeliegenden „Basisdaten“ sind in Form einer Default Theorie enthalten. Es gibt zwei Gruppen von Basisdaten: die Fakten und die Regeln.

Definition 3.2 (Default Theorie) *Eine Default Theorie (DT) $\Delta = (W, D)$ besteht aus einer Menge W von PL1 Formeln und einer Menge D von Default Regeln.*
Eine Default Regel $d_i = \alpha_i : M\beta_{i1}, \ldots, M\beta_{in} \parallel w_i$ besteht aus den 3 Teilen
- $vor(d_i) = \alpha_i$, der Default Vorbedingung
- $ann(d_i) = M\beta_{i1}, \ldots, M\beta_{in}$, den Default Annahmen, sowie
- $kon(d_i) = w_i$, der Default Konseqünz.
Hierbei sind $\alpha_i, \beta_{i1}, \ldots, \beta_{in}$ und w_i PL1 Formeln.

Als Beispiel für eine Default Regel, die den „dehnbaren“ Mengenbezeichner *die meisten* aufnimmt, könnte man
(Student(x): Mschläft-gerne-lang(x)|| schläft-gerne-lang(x))
formulieren und hiermit ein weit verbreitetes Vorurteil über Studenten ausdrücken.

[3]Prinzipiell deshalb, weil u.a. die bekannten Probleme wie bspw. die unvollständigkeit von PL1 auftreten (hierzu Absch. refmep).

3.3.2 Belief und Extension

Zentraler Punkt in der Reiterschen DL Beweis Prozedur sind die aus einer DT $\Delta = (W, D)$ herleitbaren *beliefs* (in folgenden werden beliefs als DL-Folgerungen bezeichnet). Allgemein können aus einer DT unendlich viele DL-Folgerungen hergeleitet werden. Diese DL-Folgerungen entsprechen allen möglichen Schlußfolgerungen, die aus einer unvollständigen Sammlung von Fakten W in Verbindung mit Default Regeln aus D hergeleitet werden können.

Aufgrund des hypothetischen Charakters, den die DL-Folgerungen durch vorgehende Anwendung von Default Regeln erhalten, ist die Gesamtheit einer Menge von DL-Folgerungen bzgl. Δ in sich nicht konsistent.
Reiter bildet nun konsistente Mengen solcher DL-Folgerungen: die Extensionen.

Definition 3.3 (Extension) *Sei S eine Menge von Formeln.* $\Delta = (W, D)$ *eine Default Theorie.*

$\Gamma(S)$ *ist definiert als die kleinste Menge, für die gilt:*
eine Extension

1. $W \subseteq \Gamma(S)$
2. $\Gamma(S) \models F \Rightarrow F \in \Gamma(S)$
3. Falls $(\alpha : \beta_1, \ldots, \beta_n \parallel \omega) \in D \wedge \alpha \in \Gamma(S) \wedge \beta_1, \ldots, \beta_n \notin S \Rightarrow \omega \in \Gamma(S)$

Eine Extension E ist eine Menge mit $\Gamma(E) = E$

3.3.3 Beweisverfahren in Reiters DL

Um eine Formel β in einer DT $\Delta = (W, D)$ zu beweisen, gibt Reiter die Definition eines Default Beweises.

Definition 3.4 (Default Beweis) *Sei* $\Delta = (W, D)$ *eine DT. Eine Formel* β *hat einen Default Beweis B bzgl.* Δ*, wenn es ausgehend von W eine Kette von Defaults* $D_0, \ldots, D_k, D_i \in D$ *gibt, so daß gilt:*
1. $W \cup kons(D_0) \ldots \cup kons(D_k) \vdash \beta$,

2. $\forall D_i \in D_0, \ldots, D_k : \beta_j \in ann(D_i) \Rightarrow W \cup kon(D_0) \cup \ldots \cup kon(D_{i-1}) \not\vdash \beta_i$

Vergleicht man nun diese Default-Beweise mit den Extensionen, (speziell mit dem Γ-Operator) so fällt eine Teilmengenbeziehung auf:
Falls es eine Default-Kette D gibt, die ein Default Beweis für eine Formel β ist, dann existiert auch eine Extension E, so daß:
1. $E \vdash \beta$, und
2. die Default Kette D ist in E enthalten.

Reiter verschärft diese Aussage sogar noch:

Theorem 3.5 ([Reiter,1980) *,S.103 Theorem 4.8] Für eine beliebige DT* Δ *und eine Formel* β *sind Aussage 1 und 2 äquivalent:*
1. Es gibt eine Extension E, so daß $E \vdash_\Delta \beta$
2. Es gibt einen Default-Beweis für β

Aufgrund dieser Erkenntnis sind zwei weitere Fragen von Interesse:

1. Wie kann ein Default Beweis algorithmisch berechnet werden, sowie
2. Kann man für alle Paare β, Δ beantworten, ob es eine Extension E gibt, mit $E \vdash \beta$ (extension membership problem).

Im Abschnitt 5 seiner Arbeit beschreibt Reiter einen Resolutionsbeweiser für Aussagenlogik, der ausgehend von dem Negat $\neg\beta$ der zu beweisenden Formel β mit der Steuerungsstrategie lineare Resolution[4] [Loveland,1970] arbeitet.
Dieser lineare Top Down Resolutions-Beweiser (Top Down weil er mit $\neg\beta$ als Anfangsklausel beginnt) findet imer die leere Klausel, wenn eine Extension E Existiert, mit $E \vdash_\Delta \beta$, da LR für widersprüchliche Klauselmengen vollständig ist. Und somit ist die LR eine mögliche Lösungsstrategie, um die es in der ersten Frage geht.
Die Problematik der zweiten Frage, des Extension membership Problems, behandle ich in dem nächsten Abschnitt.

[4]Die Lineare Resolution LR geht von der zu widerlegenden Klausel aus und bildet eine Resolventenkette, d.h. eine Elternklausel jedes LR-Schrittes - mit Ausnahme des ersten - ist immer die Resolvente des vorigen Resolutionsschrittes.

3.3.4 Resume zu Reiters DL

Zu Beginn dieses Abschnitts, in welchem ich abschließend die grundlegenden Möglichkeiten des nichtmonotonen Schließens, die in Reiters DL eröffnet werden, zusammenfasse, möchte ich Reiter selbst über die Zielsetzung seiner Arbeit zu Wort kommen lassen [5]:

> „...our position is that the purpose of default reasoning is to determine *one* consistent set of beliefs about a world, i.e. *one* extension, and to reason within this extension until such time as the evidence at hand forces a revision of those beliefs, in which case a switch to a new extension may be called for."

Die in dieser Aussage beschriebene Festlegung auf eine Extension und ggf. das wechseln (switching) der Extension ist eine Konseqünz der Nichtentscheidbarkeit des membership Problems.
Das Extension Membership Problem - also die Frage ob für beliebige Δ und β eine Extension E existiert, mit $E \vdash_{\Delta} \beta$ ist nicht entscheidbar, da ...?
dies bedeutet, daß die Theoreme der Reiterschen DL nicht rekursiv aufzählbar sind.
Also selbst wenn es default Beweise für β in Δ gibt, können diese Beweise i.a. nicht erstellt werden.
Dieses unangenehme Resultat ist für fast alle Beweisverfahren im Zusammenhang mit Default Theorien übertragbar: an irgendeiner Stelle eines beliebigen Verfahrens wird ein nicht entscheidbarer Prozeß auftauchen.

Zu einem Wechsel der gerade betrachteten Extension kommt es, falls eine der beiden folgenden Situationen vorliegt:

1. Beim Beweisversuch von β wurde zu einem gegebenen Δ eine Extension E bestimmt aber es gilt nicht $E \vdash_{\Delta} \beta$. Es muß eine neue Extension bestimmt werden.
2. Zu einer gegebenen DT Δ wurde eine Extension E bestimmt und $\Delta = (D, W)$ wird zu $\Delta' = (W \cup W', D)$ erweitert. Falls E keine Extension

[5]([Reiter,1980],S.94)

für Δ' ist, muß eine neue Extension bestimmt werden.[6]

Genau diese in Punkt 2. aufgeführte Eweiterung macht den nichtmonotonen Character in Reiters DL aus.

Im Gegensatz hierzu ist eine Erweiterung der DT $\Delta = (W, D)$ mit einer Menge D' von Defaults immer monoton. Es gilt:

$$(W, D) \vdash \beta \Rightarrow (W, D \cup D') \vdash \beta.$$

3.4 Pooles Konzept für DR

Die im letzten Abschnitt diskutierte DL Reiters wurde nach ihrer Veröffentlichung im Bereich der KI mit großem Interesse aufgenommen und hat verschiedenartige Diskussionen ausgelöst. Hierbei wurde die Reiters DL motivierende These, daß nämlich nichtmonotones schließen nur mit nichtmonotonen Logiken realisierbar sei als erster klärungsbedürftiger Punkt angesehen.

In diesem Abschnitt betrachte ich die von Poole 1988 veröffentlichte Arbeit „A logical Framework for Default Reasoning", in der Poole in der Poole Einblick in seine im Bereich des DR gibt und darin auch Bezug nimmt auf das 1987 veröffentlichte Konzept des logischen Schlußfolgerungssystems THEORIST [Poole et al.,1987], einem u.a. für Diagnose Programme vorgesehenen Prototyp, in dem DR-Mechanismen zur Verfügung gestellt werden.
Poole zeigt in seinem Papier, daß nichtmonotones Schlußfolgern nicht eine nichtmonotone Logik zugrunde liegen muß.

[6] Was nicht heißt, das alle Formeln die mit der alten Extension bewiesen werden konnten nun nicht mehr beweisbar (gültig) sind. Trotzdem wird durch die Neubestimmung der Extension wieder ein globales Beweisverfahren initiiert, was eigentlich vermeidbar sein sollte, wenn man eine in beiden Extensionen beweisbare Formel β vorliegen hat.

3.4.1 Beweisen mit Pooles Logikkonzept

Als logischen Rahmen für nichtmonotones Schließen wählt Poole die PL1. Der Benutzer gibt die Datenbasis in Form von zwei Mengen vor:

- $\mathcal{F}$, eine Menge von Grundformeln.[7] $\mathcal{F}$ ist die Sammlung aller *Fakten* über die Anwendung.
- Δ, eine Menge von Formeln, den *möglichen Hypothesen*.

Aus diesen Fakten und möglichen Hypothesen lassen sich nun die Hypothesen sowie die komplexeren Formelmengen Szenarium (scenario), Erklärung (explanation) und Ausprägung (extension) aufbaün. *Hypothesen* sind konsistente Grundformeln, die nach entsprechender Variableninstantiierung aus den möglichen Hypothesen resultieren. Ein *Szenarium* ist jede konsistente Vereinigung von Hypothesen und der Menge der Fakten (Def.2.1). Eine *Erklärung* für eine Grundformel g ist jedes Szenarium S, mit S $\vdash$ g (Def.2.2). Für gegebene Formelmengen $calF, \Delta$ ist eine *Ausprägung* die Menge aller logischen Folgerungen eines maximalen Szenarios von $calF, \Delta$ (Def.2.3).

Mit Hilfe dieser Definitionen erreicht Poole folgendes Resultat (Theorem 2.6):

Theorem 3.6 *Es gibt eine Erklärung für w, gdw. w in einer Ausprägung vorkommt.*

3.4.2 Pooles Constraints

Bei der Umsetzung seines „logical framework“ in das DR System THEOREM benutzt Poole neben den Konzepten von Fakten (Fakt) und möglichen Hypothesen (Default) noch eine speziell auf Defaults zugeschnittene Sperrvorrichtung ein, die er *Constraint* nennt.
Um Defaults durch Constraints sperren zu können, ist eine Kennung (naming) der einzelnen Defaults nötig.

Betrachtet man die beiden Formeln

Default $d : c \leftarrow b$

[7] eine Grundformel enthält keine Variablen.

Constraint $\neg d \leftarrow a$

wobei d die Kennung für das Default $c \leftarrow b$ ist, so kann, falls a gilt, das Default

Default $d : c \leftarrow b$

nicht in den Schlußfolgerungsprozeß integriert werden.

Durch die drei Datenobjekte Fakt, Default und Constraint ist es in THEORIST möglich Reiters „general defaults“
$\alpha(x) : M\beta_1(x), \ldots, M\beta_n(x) \parallel \gamma(x)$
folgendermaßen darzustellen:

Default $M\beta_1(x)$, ...,**Default** $M\beta_n(x)$
Constraint $\neg M\beta_1(x) \leftarrow \beta_1(x)$, ...,**Constraint** $\neg M\beta_n(x) \leftarrow \neg\beta_n(x)$
Fakt $\gamma(x) \leftarrow M\beta_1(x) \wedge \ldots \wedge M\beta_n(x) \wedge \alpha(x)$

Kapitel 4

Resolutionsverfahren mit Constraint Klauseln

Die syntaktischen Objekte, auf die das Resolutionsprinzip zuallererst angewendet wurde, sind wie in Abschnitt 2.3.1 beschrieben, die Klauseln. Die Hyperresolution und Theorieresolution (Absch. 4.1) sind zwei der zahlreichen Resolutionsverfahren, die ausschließlich diese Klauseln benutzen.

Neben diesen klassischen Klauseln etablierte sich im laufe der Zeit ein erweitertes Klauselkonzept: Die Constraint Klausel (Absch.4.2).
Mit Hilfe dieses Constraint Klauselkonzepts wurde zunächst eine Aufgabenteilung in: erstens das Herleiten einer leeren Klausel und zweitens die folgende Berechnung eines dafür nötigen Unifikators unterstützt (Absch.4.3).
Bürckert nutzte den Constraint Teil der Constraint Klauseln um damit eine eigenständige Theorie zu beschreiben (Absch.4.4).
In Abschnitt 4 betrachte ich einige Vorschläge zur Darstellung nichtmonotoner Sachverhalte durch Constraint Klauseln, die gleichzeitig die Basisdaten für die kontrollmengengesteuerte Constraint Resolution [Panitz,1993] darstellen.

4.1 Resolution mit einfachen Klauseln

Ziel der 1965 von Robinson beschriebenen Hyperresolution ist es, mehrere einfache Resolutionsschritte in Form eines einzelnen Makro-Resolutionsschrittes auszuführen. Hierbei werden alle positiven Literale der Nukleus Klausel als Resolutionsliterale mit entsprechenden Elektronen-Klauseln genutzt (Abb.4.1). Die auf diese Art entstehende Resolvente ist die Hyperresolvente.

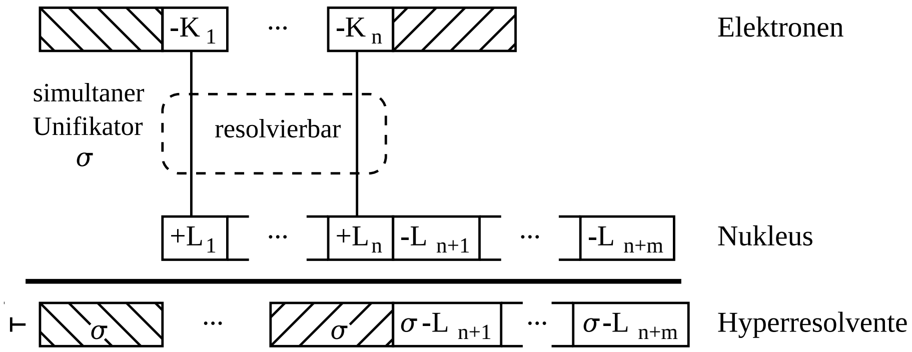

Abbildung 4.1: Makro-Resolutionsschritt bei der Hyperresolution

Die von Stickel 1985 veröffentlichte Theorieresolution sieht einen Resolutionsschritt - i.a. ebenfalls ein Makroschritt - vor, wenn aus einer beliebigen Anzahl von Klauseln bzgl. der definierten Theorie Δ eine Menge $L_1, \ldots, L_n$ von Literalen bestimmt werden kann, so daß die Konjunktion dieser Literale in der Theorie Δ unerfüllbar ist (Abb.4.1). Die Definition einer Theorie erfolgt dabei durch speziell für Prädikaten- und Funktionssymbole definierte Schlußfolgerungsregeln.

4.2 Constraint Klauseln

Grundsätzlich werden Berechungsverfahren, die das Robinsonsche Resolutionskalkül für Anwendbarkeit auf Constraint Klauseln modifizieren, als Constraint Resolutionsprinzipien bezeichnet. Da die Bedeutung des Constraints einer Constraint Klausel vom jeweiligen Constraint Resolutionsprinzip abhängt, muß man bei dem Versuch diese Constraint Klauseln zu definieren, einen größtmöglichen gemeinsamen Nenner finden, dem die Constraint Klauseln der verschiedenen Constraint Resolutionsprinzipien genügen. So gesehen kommt es zu folgender Definition für eine Constraint Klausel:

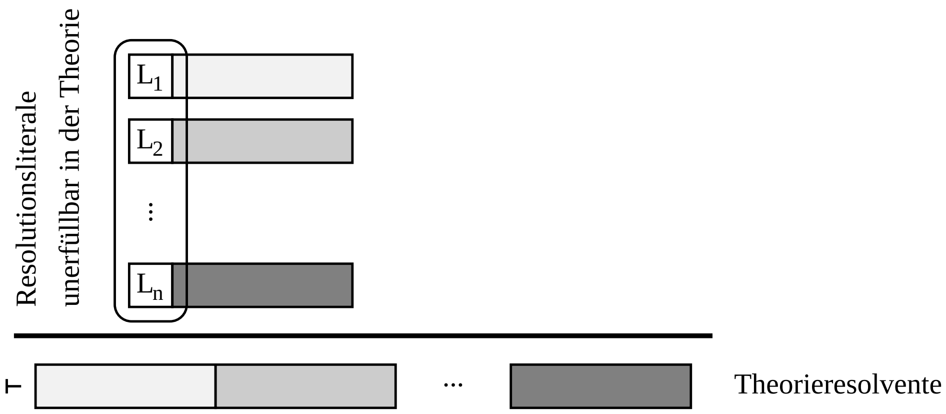

Abbildung 4.2: Makro-Resolutionsschritt bei der Theorieresolution

Definition 4.1 (Constraint Klausel eines CRP) *Sei $\mathcal{R}$ ein Constraint Resolutionsprinzip. Eine in $\mathcal{R}$ resolvierbare Constraint Klausel $K = F \parallel B$ besteht aus zwei Teilen:*
1. Dem Kern F. Hier handelt es sich um eine Klausel, mit der resolviert werden kann.
2. Dem Constraint B. Die Funktion des Constraint wird vom jeweiligen Resolutionsprinzip $\mathcal{R}$ bestimmt. Sie ist i.d.R. mit der Funktion des Kerns verknüpft.

Beispiele für spezielle Constraint Anwendungen sind zu finden in:

- Constraint zur Darstellung eines Gleichungssystems [Huet,1972] (Absch.4.3)
- Constraint zur Beschreibung einer eigenständigen Theorie [Bürckert,1991](Absch.4.4)
- Constraint mit Kontrollmengendefinierender Funktion [Panitz,1993](Absch.**??**)

Verschiedene Interpretationen für Constraint Resolutionsprinzipien folgen in den nächsten beiden Abschnitten.

4.3 Constraint Klauseln für Unifikationsberechnung

[Huet,1972] teilte die klassische Klausel in einen für die Resolventenbildung vorgesehenen Teil (Kern) und einen Constraintteil, in welchem er die für die jeweilige Resolventenbildung nötige Information aufsammelte. Hierdurch ist es möglich, solange mit der Unifikationsberechnung zu warten, bis eine leere Klausel abgeleitet werden kann; falls im Anschluän die Bestimmung der leeren Klausel für den begleitenden Constraint Teil in Form eines Gleichungssystems ein Unifikator $\mathcal{U}$ betimmt werden kann, so gilt die leere Klausel als der gesuchte Widerspruch mit Unifikator $\mathcal{U}$.

4.4 Constraint Klauseln für Resolution mit eingeschränkten Quantoren

Bürckerts Constraint Resolutionsprinzip ist eine konseqünte Weiterführung der Idee die Semantik einer beliebigen Anwendungsfrage - in Form einer entsprechenden Hintergrundtheorie - in den Berechnungsvorgang zu integrieren. Bürckert nutzt hierbei das Constraint Klauselkonzept um mit dem Constraint einer CK eine beliebige Hintergrundtheorie in den Resolutionsvorgang zu integrieren. Diese Hintergrundtheorie kann entweder als Menge von Axiomen definiert werden, oder auch ganz einfach als eine beliebige Menge von Modellen verstanden werden. Betrachtet man die Constraint Resolventenbildng

$C_1 = P(t_1, \ldots, t_n) \cup C \parallel R$
$C_2 = \neg P(t'_1, \ldots, t'_n) \cup D \parallel S$
$C_{Res} = C \cup D \parallel R \wedge S \wedge \Gamma$
wobei Γ Unifikator für $\{P(t_1, \ldots, t_n), P(t'_1, \ldots, t'_n)$ ist.

bei Bürckert, so sind zwei Voraussetzungen zu nennen:

1. Die Kerne müssen über potentiell resolvierbare Literale verfügen [1]
2. Der Constraint $R \wedge S \wedge \Gamma$ muß bzgl. der Constraint Theorie erfüllbar sein.

Um eine Menge von Constraint Klauseln zu widerlegen,ist es i.a. nötig nicht nur eine, sondern mehrere leere Constraint Klauseln herzuleiten, da eine Menge von Constraint

[1] Zwei Literale sind potentiell resolvierbar, wenn die zugehörigen Prädikatsymbole identisch sind, und die Literale gegensätzliche Vorzeichen haben.

Klauseln genau dann unerfüllbar ist, wenn für jedes Modell $\mathcal{M}$ der Constraint Theorie eine leere Klausel C resolviert werden kann, deren Constraint von diesem Modell $\mathcal{M}$ erfüllt wird.

Kapitel 5

Die Kontrollmengengesteuerte Constraint Resolution

In diesem Kapitel beschreibe ich die in [Panitz,1993] vorgestellte Kontrollmengengesteuerte Constraint Resolution (KCR).
Ausgangsstellung dieses DR-Systems ist die Darstellung eines beliebigen Problembereichs der realen Welt in Form einer Constraint Klausel Theorie (Absch.5.1.1).
Ziel der KCR ist es, für eine Anfrage $\mathcal{A}$ einen Widerlegungsbeweis in der Constraint Klauseltheorie (CKT) Δ zu finden.
Wie ein solcher Widerlegungsbeweis für $\mathcal{A}$ bzgl. Δ aussieht, hängt zunächst von dem Konsistenztyp der Theorie Δ ab.
Eine CKT Δ hat genau einen der drei Konsistenztypen „inkonsistent", „konsistent mit nichtleerer Kontrollmenge", oder „konsistent mit leerer Kontrollmenge".

Falls eine Theorie vom Konsistenztyp „konsistent mit nichtleerer Kontrollmenge" ist, wird im Beweisverfahren eine Kontrollmenge aufgebaut (Absch.5.3).

Die von mir getroffene Entscheidung, für eine gegebene Constraint Klauseltheorie zuallererst den Konsistenztypen zu bestimmen, macht eine vorbereitende Resolution notwendig. Damit diese vorbereitende Resolution terminiert (Ziel ist das auffinden sämtlicher leerer Constraint Klauseln - und im Fall einer inkonsistenten Theorie sogar das herleiten der leeren Klausel - um hierauf nötigenfalls eine Kontrollmenge zu bestimmen), definiere ich gleich zu Beginn eine grundlegende Reklementierung der in Klauseln und Constraint Klauseln auftretenden Literale, indem ich als Argumente innerhalb der Argumentlisten der Prädikate nur Variable und Konstante erlaube.

5.1 Constraint Klauseltheorie und Constraint Resolutionsprinzip

In diesem Abschnitt gebe ich die Definition einer Constraint Klauseltheorie (Punkt 5.1.1) und beschreibe, wie die vom klassischen Resolutionsprinzip her bekannten Techniken auf das in der kontrollmengengesteuerten Constraint Resolution benutzte Constraint Resolutionsprinzip übertragen werden (Punkt5.1.2)

5.1.1 Constraint Klauseltheorien

Dem Prinzip Pooles folgend, übernimmt Panitz die PL1 als Logik seiner KCR. Um nichtmonotone Sachverhalte darzustellen, nutzt er das Konzept der Constraint Klausel. Diesem Konzept schließe ich mich prinzipiell an. Um allerdings die Terminierungseigenschaft des Constraint Resolutionsprinzips zu erzwingen, schränke ich die PL1 als Logik ein, indem ich die Literale der PL1 folgendermaßen reklementiere:

Definition 5.1 (reduziertes Literal) *Ein Literal $P(a_1, \ldots, a_n)$ bzw. $-P(a_1, \ldots, a_n)$ heißt reduziert, falls für alle i gilt:*
a_i ist entweder eine Variable oder eine Konstante.

Bemerkung: Die auf diese Weise definierten reduzierten Literale büßen an Ausdrucksstärke ein. Prädikate der Form $P(a_1, \ldots, f(b_1, \ldots, b_n), \ldots, a_n$, in denen ein Funktionssymbol f auftaucht sind nicht mehr zulässig.

Die Teillogik der PL1, die nur solche reduzierte Literale zuläßt, ist als Folgerung der Entscheidbarkeit von PL0 ebenfalls emtscheidbar.

Diese Reklementierung ist somit grundlegend für das Konzept der Constraint Klauseltheorie und zuallererst für deren Klauseln und Constraint Klauseln.

Definition 5.2 (Constraint Klausel und Klausel) *Eine Constraint Klausel (CK) C besteht aus zwei Komponenten $C = F \parallel B$. F ist der Kern und B der Constraint* *der Constraint Klausel C. Hierbei ist der Kern F eine Klausel, deren Literale alle reduzierte Literale sind und der Constraint B eine Klauselmenge, deren Literale alle reduzierte Literale sind.*

Eine Constraint Klausel $C = \square \parallel B$ *mit leerem Kern und nichtleerem Constraint wird leere Constraint Klausel (kurz: LCK für Empty Constraint Clause) genannt.*

Eine Klausel K ist eine Disjunktin von reduzierten Literalen.

Aufgabe des Benutzers ist es zunächst, durch Nutzung der Konzepte Klausel und Constraint Klausel eine Beschreibung des ausgewählten Problembereichs auszuwählen. Eine solche Sammlung von Beobachtungen entspricht einer Constraint Klauseltheorie (CKT) Δ:

Definition 5.3 (Constraint Klausel Theorie) *Eine Constraint Klauseltheorie* Δ *(kurz: CKT) ist ein Tupel* $\Delta = (W, D)$*. W ist endliche Menge von Klauseln, D ist eine endliche Menge von Constraint Klauseln. Die Klauseln werden Fakten und die Constraint Klauseln werden Regeln genannt.*

5.1.2 Das Constraint Resolutionsprinzip bei der Kontrollmengengesteuerten Constraint Resolution

Das Constraint Resolutionsprinzip bei der Kontrollmengengesteuerten Constraint Resolution benutzt eine Erweiterung der in den Abschnitten 2.3.3 und **??** vorgestellten Inferenz- und Löschregeln. Falls innerhalb des des Constraint Resolutionsprinzips zwei klassische Klauseln miteinander resolvieren, bleibt es beim klassischen Resolutionsschritt. Bei einem Constraint Resolutionsschritt ist mindenstens eine der beiden Klauseln eine Constraint Klausel, und der Resolutionsschritt ist folgendermaßen definiert:

Definition 5.4 (Constraint Resolutionsschritt) *Seien* $C_1 = L, M_1, \ldots, M_n \parallel B_1$ *und* $C_2 = \neg L', N_1, \ldots, N_m \parallel B_2$ *Constraint Klauseln und sei* $C_3 = \neg L', N_1, \ldots, N_m$ *eine Klausel. Sei außerdem* σ *ein Unifikator für* $\{L, L'\}$*, so gilt:*

1. $\sigma(M_1, \ldots, M_n, N_1, \ldots, N_m) \parallel \sigma(B_1 \cup B_2)$ *ist Constraint Resolvente mit den Eltern Klauseln* C_1 *und* C_2*.*
2. $\sigma(M_1, \ldots, M_n, N_1, \ldots, N_m) \parallel \sigma(B_1)$ *ist Constraint Resolvente mit den Elternklauseln* C_1 *und* C_3*.*

Der Constraint ist so ausgelegt, daß die Kerne der miteinander resolvierenden CK den aktiven Part bestimmen und die Constraints eigentlich nur aufgesammelt und gegebenenfalls die für Resolutionsliterale der Kern notwendige Substitution σ auf sie angewendet wird.
Ähnlich verhält es sich mit der Faktorisierung. Falls der Kern faktorisiert werden kann, wird er faktorisiert und die hierfür notwendige Substitution σ auf den Constraint angewendet.

Definition 5.5 (Constraint Faktorisierung) *Sei* $C = \{L, L', M_1, \ldots, M_n\} \parallel B$ *eine beliebige Constraint Klausel. Falls* $\sigma(L) = \sigma(L')$ *so ist* $C' = \sigma(\{L, M_1, \ldots, M_n\} \parallel \sigma(B))$ *ein Constraint Faktor von* C*. Das bilden eines Constraint Faktors von* C *heißt Constraint Faktorisierung von* C*.*

Somit sind die Inferenzregeln des Resolutionskalküls für die Anwendung auf Constraint Klauseln entsprechend modifiziert.
Als nächstes werden die Definitionen für tautologische Constraint Klauseln, Constraint Klausel mit einem isolierten Literal und Subsumtion zweier Constraint Klauseln gegeben:

Definition 5.6 (tautologische Constraint Klausel) *Eine Constraint Klausel* $C = L_1, \ldots, L_n \parallel B$ *ist eine Tautologie, falls* $\exists L_i, L_j \in C$ *so daß* $L_i = \neg L_j$*.*

Definition 5.7 (Constraint Klausel mit isoliertem Literal) *Sei* $\mathcal{C} = \{C_1, \ldots, C_n\}$ *eine Menge von Constraint Klauseln. Die Constraint Klausel* $C_i = L, M_1, \ldots, M_n \parallel B_i$ *hat ein isoliertes Literal* L*, falls:*
$\nexists C_j \in \mathcal{C}$ *deren Kern ein Literal L' enthält, so daß ein* σ *existiert, mit:* $\sigma(L') = \sigma(\neg L)$*.*

Wie gesehen können tautologische Constraint Klauseln und Constraint Klauseln mit isolierten Literalen unabhängig vom Constraint bestimmt werden. Dies ist bei der

Subsumtionsregel nicht der Fall:
Die Einbeziehung der Constraints beider Constraint Klauseln ist nötig. Es ergibt sich folgender Sachverhalt:

Definition 5.8 (Subsumtionsregel für Constraint Klauseln) *Seien* $C_1 = (A \parallel B_1 \wedge \ldots \wedge B_n)$ *und* $C_2 = (A' \parallel C_1 \wedge \ldots \wedge C_m)$ *beliebige Constraint Klauseln, dann gilt:* C_2 *subsumiert* C_1 *gdw.*
$\exists \sigma$ *mit:*
1. $\sigma(A') \subseteq A$ *und*
2. $\forall B_i \in B_1, \ldots, B_n \exists C_j \in C_1, \ldots, C_m$*, so daß* $\sigma C_j \subseteq B_i$

Wie in Kapitel 4 beschrieben, sind Constraint Resolutionsverfahren variantenreich. In der nächsten Definition wird das Constraint Resolutionsprinzip der Kontrollmengengesteuerten Constraint Resolution vorgestellt:

Definition 5.9 (Das Constraint Resolutionsprinzip bei der KCR) *Sei* Ω *eine Menge von Klauseln und Constraint Klauseln. Das Constraint Resolutionsprinzip bei der Kontrollmengengesteuerten Constraint Resolution (kurz:* $\mathcal{CRP}_{KCR}$*) besteht aus folgenden drei Komponenten:*

- *den beiden Inferenzregeln*
  - *Resolutionsschritt bzw. Constraint Resolutionsschritt und*
  - *Faktorisierung bzw. Constraint Faktorisierung*
- *den drei Löschregeln*
  - *löschen der tautologischen Klauseln bzw. der tautologischen Constraint Klauseln*
  - *löschen der subsummierten Klauseln bzw. der subsummierten Constraint Klauseln und*
  - *löschen der Klauseln und Constraint Klauseln die ein isoliertes Literal enthalten*
- *einer vollständigen Steuerungsstrategie für eine dynamische Menge von Objekten (in unserem Fall Klauseln und Constraint Klauseln).*

Man bezeichnet $\mathcal{CRP}_{KCR}(\Omega)$ als die Menge Ω vereinigt mit allen aus Ω mit dem $\mathcal{CRP}_{KCR}$ ableitbaren Resolventen.

Unter einer vollständigen Steuerungsstrategie sei eine Steuerungsstrategie verstanden, die sämtliche möglichen Paare von Klauseln bildet und damit die Inferenz- und Löschregeln durchführt. Die Dynamik der Liste kommt durch die Anwendung von Inferenz- und Löschregeln zustande, da entweder Elemente hinzukommen oder wegfallen.

5.2 Konsistenztyp einer CKT

Nachdem im letzten Abschnitt das Constraint Resolutionsprinzip für Constraint Klauseltheorien beschrieben wurde, geht es nun darum, erste Aussagen über die vom Benutzer vorgegebene Constraint Klauseltheorie zu treffen. Diese Aussagen betreffen das Konsistenzverhalten der Constraint Klauseltheorie:

Definition 5.10 (Konsistenzeigenschaften einer CKT) *Im folgenden werden drei Konsistenztypen für CKTs definiert:*

- *Eine CKT Δ =(W,D) heißt* **inkonsistent***, wenn W inkonsistent ist.*
- *Seien $\Delta = (W, D)$ eine CKT,*
  $W = \{C_1, \ldots, C_n\}$ die Menge der Fakten und
  $D = \{F_1 \parallel B_1, \ldots, F_n \parallel B_n\}$ die Menge der Constraint Klauseln.
  Δ heißt **konsistent mit nichtleerer Kontrollmenge** *wenn Δ nicht inkonsistent ist und die Klauselmenge $W \cup F_1 \ldots \cup F_n$ inkonsistent ist.*
- *Eine CKT Δ =(W,D) heißt* **konsistent mit leerer Kontrollmenge***, wenn $W \cup F_1 \ldots \cup F_n$ konsistent ist.*

Bemerkung (Konsistenzeigenschaften einer CKT):
Die drei möglichen Konsistenzeigenschaften einer CKT Δ sind deren eigentlicher Charakter. So ist im Fall einer inkonsistenten CKT Δ ein vorzeitiges Beweisende erreicht: Mit Δ als inkonsistenter Datenbasis liesse sich beliebiges folgern.

Das Konsistenzverhalten einer Constraint Klauseltheorie Δ ist grundlegend für den weiteren Beweisverlauf. Wie im nächsten Kapitel geschildert, ist eine Kontrollmengenbestimmung nur nötig, falls Δ konsistent mit nichtleerer Kontrollmenge ist.

5.3 Beweisen mit Kontrollmengensteuerung

Ausgangspunkt beim Beweisen mit Kontrollmengensteuerung ist immer eine Conctraint Klauseltheorie Δ, die konsistent mit nichtleerer Kontrollmenge ist. Für eine solche Theorie muß zunächst eine Kontrollmenge bestimmt werden, um die auftretenden Widersprüche in Form von leeren Constraint Klauseln zu neutralisieren. Eine Kontrollmenge wird fogendermaßen definiert:

Definition 5.11 (Kontrollmenge) *Sei $\Delta = (W, D)$ eine CKT die konsistent mit nichtleerer Kontrollmenge ist. Eine Menge $K_\Delta = \{L_1, \ldots, L_n\}$ von Literalen L_i ist eine Kontrollmenge für Δ, falls gilt:*
$\forall C = \Box \parallel \{\{B_{11}, \ldots, B_{1n_1}\}, \ldots, \{B_{m1}, \ldots, B_{mn_m}\}\}$ *mit* $C \in \mathcal{CRP}_{KCR}(\Delta)$ *folgt:*
$\exists j$ *so daß für jedes* $B_{jk} \in \{B_{j1}, \ldots, B_{jn}\}$ *gilt:*
$\exists L_i \in K_\Delta$ *mit* $B_{jk} = \neg L_i$

Zur Kontrollmengenbestimmung selbst ist aufgrund der aus einer Constraint Klausel Theorie Δ resolvierbaren leeren Constraint Klauseln folgende Beobachtung zu machen:

- Es gilt: $\Box \parallel \{\{B_1, \ldots, B_n\}\} \in \mathcal{CRP}_{KCR}(\Delta) \Rightarrow \forall K_\Delta : \{\neg B_1, \ldots, \neg B_n\} \in K_\Delta$
- Es gilt: $\Box \parallel \{\{L_{11}, \ldots, L_{1n_1}\}, \ldots, \{L_{m1}, \ldots, L_{mn_m}\} \in \mathcal{CRP}_{KCR}(\Delta) \Rightarrow$
  $\forall K_\Delta : \exists j \wedge \{\neg L_{j1}, \ldots, \neg L_{jn_j}\} \in K_\Delta$

Nachdem alle Resolventen der Theorie Δ ermittelt sind, hat man in der Regel mehrere leere Constraint Klauseln (mindestens eine) gefunden. Diese LCK deuten gewisse Inkonsistenzen in der Theorie Δ an. So bedeutet beispielsweise die LCK

$\Box \parallel \neg UMandat(A)$das im hypothetischen Fall von $\mathcal{O}(\neg UMandat(A)) = WAHR$ ein Widerspruch in Δ auftreten würde, da $\Box \parallel TAUT = \Box$ [1]

Um nun aber die Konsistenz von Δ zu gewährleisten, muß man an dieser Stelle UMandat(A) (als Kurzform für $\neg\neg UMandat(A)$) fordern. Außerdem merkt man sich diese Forderung an einer Sammelstelle: Der Kontrollmenge.

Eine konsistenzerhaltende Kontrollmenge für eine Theorie Δ kann demnach folgendermaßen bestimmt werden: Alle LCK die von Δ aus ableitbar sind werden der Reihe nach besucht und eines der Konjunktionsglieder [2] im Constraint wird ausgewählt, negiert und in die Kontrollmenge aufgenommen. Auf diese Weise wird eine Konsistenzvariante für Δ erzielt.

Diese Überlegungen lassen sich folgendermaßen zusammenfassen:

Theorem 5.12 *Sei Δ konsistent mit nichtleerer Kontrollmenge und $\mathcal{K}_\Delta = \{B_1 \wedge \ldots \wedge B_n\}$ eine Kontrollmenge von Δ, dann gilt:*

$$\forall C = \Box \parallel C_1, \ldots, C_n \wedge C \in \mathcal{CRP}_{KCR}(\Delta) \Rightarrow$$
$$C_1, \ldots, C_n, B_1, \ldots, B_m \models \Box$$

Beweis:
Es gibt ein i und ein j mit $\neg C_i = B_j$ aufgrund der Konstruktion von K_Δ und es gilt $C_i \wedge B_j = \perp$

Definition 5.13 (Bedingungslos Konsistent) *Sei $\mathcal{K}_\Delta = \{B_1, \ldots, B_n\}$ Kontrollmenge für Δ und $C = \Box \parallel C_1 \wedge \ldots \wedge C_m$ mit $C_i = c_{i1} \vee \ldots \vee c_{in_i}$.*
C ist bedingungslos konsistent mit $\mathcal{K}_\Delta$ falls: $\forall C_i : \exists$ Literal c_{ij}, so daß $\nexists B_k \in \mathcal{K}_\Delta$ mit

[1] $\mathcal{O} : F \mapsto \{WAHR, FALSCH\}$ wird in diesem Zusammenhang als eine Orakel-Funktion verstanden (Nichtdeterminismus) und bedeutet daß es an dieser Stelle prinzipiell möglich ist eine Entscheidung bezüglich des Wahrheitswertes der Formel F zu treffen. So könnte es beispielsweise möglich sein, daß der Benutzer die Formel F einzuschätzen weiß (wie und warum ist nebensächlich) oder aufgrund der Datenbasis eine Entscheidung (im aufwendigsten Fall mit Resolution, leichtestenfalls durch ablesen) festgestellt werden kann. Die Entscheidung an dieser Stelle zu treffen ist allerdings garnicht relevant.

[2] Sei $\mathcal{K} = K_1 \wedge \ldots \wedge K_n$ eine Klausel in Klauselnormalform. K_i wird Konjunktionsglied von $\mathcal{K}$ genannt.

1. *Das Prädikat von* c_{ij} *und* B_k *ist identisch und*
2. *Die Literale* c_{ij} *und* B_k *haben verschiedene Wertigkeit* [3]

Definition 5.14 (Kontrollmengengesteuerte Constraint Resolution) *Seien* $\Delta = (W, D)$ *eine CKT die konsistent mit nichtleerer Kontrollmenge ist,* $\mathcal{CRP}_{KCR}$ *das Constraint Resolutionsprinzip und* $\mathcal{A}$ *eine Menge von Klauseln und Constraint Klauseln.*

Die kontrollmengengesteuerte Constraint Resolution besteht aus den folgenden Schritten:

- **Schritt 1**: *Bilde* $\mathcal{CRP}_{KCR}(\Delta)$
  Merke : $a = \mathcal{CRP}_{KCR}(\Delta)$
- **Schritt 2**: *Nimm alle LCK aus* $\mathcal{CRP}_{KCR}(\Delta)$*.*
  Merke : $b = \{c_i | c_i \in \mathcal{CRP}_{KCR}(\Delta) \wedge c_i = \Box \parallel B_1, \ldots, B_n\}$
- **Schritt 3**: *Nimm die nichtleeren Constraint Klauseln aus* $\mathcal{CRP}_{KCR}(\Delta)$*.*
  Merke : $c = \{c_j | c_j \in \mathcal{CRP}_{KCR}(\Delta) \setminus b\}$
- **Schritt 4**: *Bestimme eine Kontrollmenge für* Δ*.*
  Merke : $d = K_\Delta$
- **Schritt 5**: *Bilde* $\mathcal{CRP}_{KCR}(\Delta \cup \mathcal{A}$*.*
  Merke : $e = \mathcal{CRP}_{KCR}(\Delta \cup \mathcal{A})$
- **Abfrage 6.1**: *Falls leere Klausel in e, dann f.*
  Ausgabe : $\mathcal{A}$ *widerlegt in* Δ *mit* K_Δ
- **Abfrage 6.2**: *Falls keine Leere Constraint Klausel in e, dann g.*
  Ausgabe : $\mathcal{A}$ *nicht widerlegbar in* Δ*.*
- **Schritt 6.3**: *Sammle alle neuen Leeren Constraint Klauseln aus e.*
  Merke : $h = \{c_i | c_i \in e \wedge c_i = \Box \parallel D_1, \ldots, D_n\}$

[3] Zwei Literale haben verschiedene Wertigkeit, wenn genau eins der Literale negiert ist.

- **Schritt 7**: *Bilde Konsistenzaussage von Kontrollmenge und den Constraints der einzelnen Leeren Constraint Klauseln.*
  Merke : $i = \{(K_\Delta, D_1, \ldots, D_n)|\Box \parallel D_1 \in h$

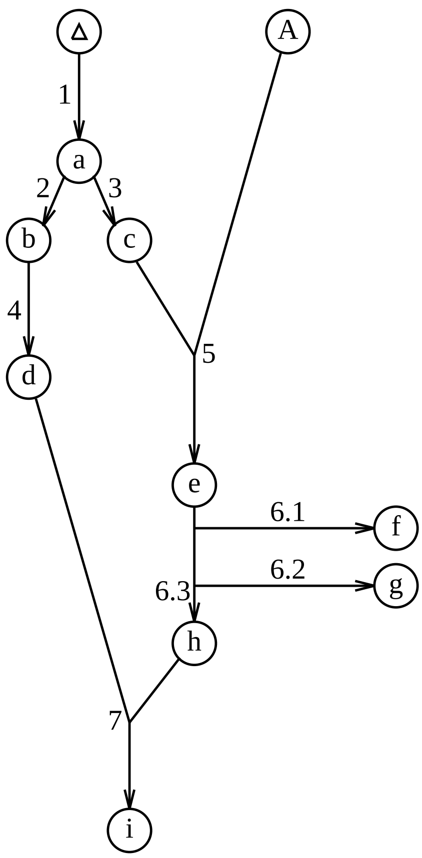

Abbildung 5.1: Kontrollfluß für Constraint Klauseltheorien, die konsistent mit *nichtleerer* Kontrollmenge sind

Definition 5.15 (Widerlegbarkeit von $\mathcal{A}$) *Sei $\Delta = (W, D)$ eine CKT die konsistent mit nichtleerer Kontrollmenge ist. Sei $K_\Delta = \{K_1, \ldots, K_n\}$ eine Kontrollmenge für Δ.*

Eine Menge $\mathcal{A}$ von Klauseln und Constraint Klauseln ist widerlegbar in Δ mit K_Δ und einem Constraint $\{\{c_{11}, \ldots, c_{1n_1}\}, \ldots, \{c_{m1}, \ldots, c_{mn_m}\}\}$, falls
$\exists C = \Box \parallel \{\{c_{11}, \ldots, c_{1n_1}\}, \ldots, \{c_{m1}, \ldots, c_{mn_m}\}\}$, *mit*

- $C \notin \mathcal{CRP}_{KCR}(\Delta)$ *und*
- $C \in \mathcal{CRP}_{KCR}(\Delta \cup \mathcal{A})$ *und*
- $\nexists\{c_{i1}, \ldots, c_{in_i}\}(i \in 1, \ldots, m)$, *so daß* $\{c_{i1}, \ldots, c_{in_i}, k_1, \ldots, k_n\}$ *inkonsistent ist.*

Bevor nun die Zusammenführung der bisher getroffenen Definitionen folgt, gebe ich noch die folgende Definition:

Definition 5.16 (Konsistenz für K_Δ und Constraint) *Seien* $K_\Delta = \{K_1, \ldots, K_n\}$ *eine Kontrollmenge und* $C = \{\{c_{11}, \ldots, c_{1n_1}\}, \ldots, \{c_{m1}, \ldots, c_{mn_m}\}\}$ *ein Constraint.* K_Δ *und* C *sind konsistent, falls* $/\exists\{c_{i1}, \ldots, c_{in_i}\}(i \in 1, \ldots, m)$, *so daß* $\{c_{i1}, \ldots, c_{in_i}, k_1, \ldots, k_n\}$ *inkonsistent ist.*

Theorem 5.17 *Sei* $\Delta = (W, D)$ *eine CKT die konsistent mit nichtleerer Kontrollmenge ist. Sei* $\mathcal{A}$ *eine Menge von Klauseln und Konstraint Klauseln.*
Falls es eine Kontrollmenge K_Δ *gibt, so daß* $\mathcal{A}$ *ist widerlegbar in* Δ *mit* K_Δ, *dann findet man für* $\Delta = (W, D)$ *und* $\mathcal{A}$ *eine solche Kontrollmenge* K_Δ *durch Anwendung der kontrollmengengesteuerten Constraint Resolution.*

Beweis:
Die folgenden drei Punkte ergeben die Skizzierung des Beweises:

1. Zu einer gegebenen Constraint Klauseltheorie die konsistent mit nichtleerer Kontrollmenge ist, können nur endlich viele leere Constraint Klauseln berechnet werden. Ein Constraint einer solchen leeren Constraint Klausel hat endlich viele Konjunktionsglieder. Die Folgerung hieraus ist, daß nur endlich viele Kontrollmengen bestimmt werden können.

2. Nun kann man $\mathcal{A}$ zusammen mit Δ resolvieren und aus dieser Resolventenmenge *neue* leere Constraint Klauseln herausnehmen. Da nach Voraussetzung $\mathcal{A}$ in Δ mit einer Kontrollmenge widerlegbar ist, muß mindestens eine solche *neue* leere Constraint Klausel existieren.

3. Wie schon in Punkt 2 wird auch an dieser Stelle die Voraussetzung, daß $\mathcal{A}$ in Δ mit einer Kontrollmenge widerlegbar ist, genutzt. Diesmal schließe ich, daß zwar nicht zu dem Constraint jeder *neuen* leeren Constraint

Klausel eine konsistente Kontrollmenge existiert, daß es aber aufgrund der Voraussetzung mindestens eine der *neuen* leeren Constraint Klauseln gibt, deren Constraint mit mindestens einer der Kontrollmengen konsistent ist. Da es nur endlich viele *neue* leere Constraint Klauseln und auch nur endlich viele Kontrollmengen gibt, kann mindestens eine *neue* leere Constraint Klausel zusammen mit einer Kontrollmenge bestimmt werden, so daß Kontrollmenge und der Constraint dieser *neuen* leeren Constraint Klausel konsistent sind.

5.4 Beweisen ohne Kontrollmengensteuerung

Abbildung 5.2 zeigt den Kontrollfluss um die Widerlegbarkeit einer Klauselmenge $\mathcal{A}$ in einer CCT Δ zu bestätigen. Δ ist vom Konsistenztyp „konsistent mit nichtleerer Kontrollmenge".

- **Schritt 1**: Bilde $\mathcal{CRP}_{KCR}(\Delta)$.
  Merke : $a = \mathcal{CRP}_{KCR}(\Delta)$

- **Schritt 2**: Bilde $\mathcal{CRP}_{KCR}(a \cup A)$.
  Merke : $b = \mathcal{CRP}_{KCR}(a \cup A)$

- **Abfrage 3.1**: Falls leere Klausel in b, dann c.
  Ausgabe : $\mathcal{A}$ widerlegt in Δ

- **Abfrage 3.2**: Falls weder eine Leere Klausel noch eine bedingte leere Klausel in b ist, dann d.
  Ausgabe : $\mathcal{A}$ nicht widerlegbar in Δ

- **Schritt 3.3**: Alle gefundenen Leeren Constraint Klauseln kommen nach e.
  Merke : $e = \{\Box \parallel B | \Box \parallel B \in b\}$

- **Schritt 4** (Transformationsschritt): Eine LCK aus e wird genommen und deren Constraint als neue Klausel nach f gebracht.
  Merke : $f = B$, falls $\Box \parallel B \in e$, und
  $e_{neu} := e \setminus \Box \parallel B$

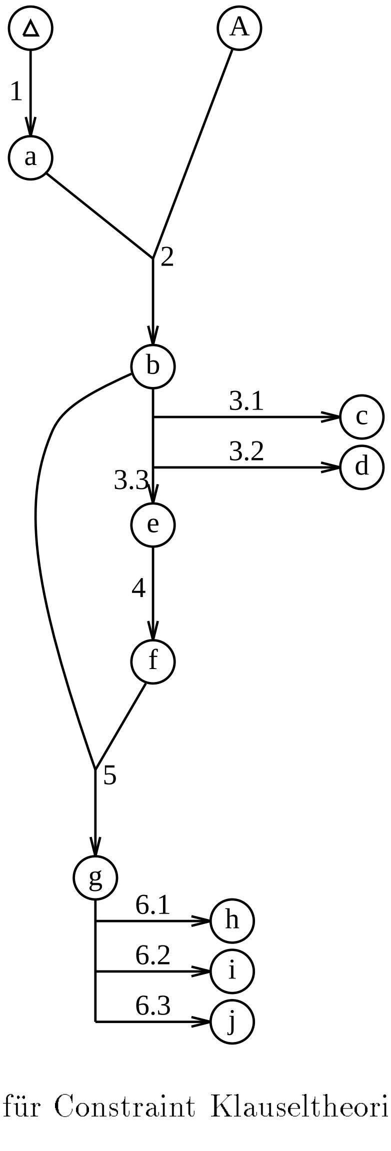

Abbildung 5.2: Kontrollfluß für Constraint Klauseltheorien, die konsistent mit *leerer* Kontrollmenge sind

- **Schritt 5**: Resolviere b und f.
  Merke : $g = \mathcal{CRP}_{KCR}(b \cup f)$

- **Abfrage 6.1**: Falls leere Constraint Klausel in g vorkommt, so konnte $\mathcal{A}$ nicht in Δ widerlegt werden.
  Falls e nichtleer, gehe zurück zu Schritt 4.
  Falls e leer, dann h
  Ausgabe : $\mathcal{A}$ konnte nicht in Δ widerlegt werden.

- **Abfrage 6.2**: Falls weder eine leere Klausel noch eine leere Constraint Klausel in g ist, dann kann $\mathcal{A}$ in Δ widerlegt werden. Erweitere i mit der Bedingung: f = TRUE.
  Falls e leer ist, gehe zu Schritt 6.3;
  Falls e nichtleer ist, gehe zu Schritt 4.
  Merke : $i = f = TRUE$

- **Schritt 6.3**: $\mathcal{A}$ konnte in Δ widerlegt werden.
  Ausgabe : $j = \mathcal{A}$ konnte in Δ widerlegt werden. Jede Bedingung aus i ist hierfür anwendbar.

Kapitel 6

Das Programm KoResB

Im Mittelpunkt dieses Kapitels steht der von mir implementierte Beweiser *KoResB*. Hierfür benutzte ich den in [Jones,1991] vorgestellten funktionalen Interpreter *Gofer*. *KoResB* knüpft an das in [Panitz,1993] vorgestellte Constraint Resolutionsprinzip mit Kontrollmengensteuerung an. Gleich zu Beginn dieses Kapitels (Absch.6.1) beschreibe ich den Umfang des von *KoResB* lieferbaren Ergebnisses. Es wird nur eine der möglichen Kontrollmenge bestimmt und auch nur - falls vorhanden - eine neue leere Constraint Klausel, deren Constraint mit der Kontrollmenge konsistent ist.

In den weiteren Abschnitten behandle ich die Form der Eingabe, gebe eine ausführliche Beschreibung für die Unifikation zweier Argumentlisten und betrachte drei kleinere Anwendungsbeispiele.

6.1 *KoResB*

KoResB steht für Kontrollmengengesteuerter Resolutionsbeweiser. Die Bezeichnung weist auf den Kontrollmengengesteuerten Beweisverlauf im Zusammenhang mit Constraint Klauseltheorien hin, die konsistent mit nichtleerer Kontrollmenge sind.

Für die Schichten von *KoResB* gilt folgendes:

- Die Logikschicht ist eine Teilmenge der PL1, bei der alle Literale reduzierte Literale sind.

- Das Kalkül ist das im letzten Kapitel vorgestellte $\mathcal{CRP}_{KCR}$.
- Die Repräsentation der Formeln erfolgt als Constraint Klauseln und Klauseln.
- Die benutzte Steuerung ist die einfache Breitensuche.

Folgender Satz beschreibt das Ein- und Ausgabeverhalten von *KoResB*:

Theorem 6.1 *Sei $\Delta = (W, D)$ eine Constraint Klausel Theorie und sei Δ konsistent mit nichtleerer Kontrollmenge. Sei $\mathcal{A}$ eine Menge von Klauseln und Constraint Klauseln. KoResB erfüllt die folgenden beiden Bedingungen:*

1. *KoResB bestimmt eine beliebige Kontrollmenge $K_\Delta = \{k_1, \ldots, k_n\}$*

2. *Falls $\mathcal{A}$ widerlegbar in Δ mit der von KoResB bestimmten Kontrollmenge K_Δ, so findet KoResB eine leere Constraint Klausel $C = \Box \parallel \{\{c_{11}, \ldots, c_{1n_1}\}, \ldots, \{c_{m1}, \ldots, c_{mn_m}\}\}$ mit:*

   - $C \notin \mathcal{CRP}_{KCR}(\Delta)$
   - $C \in \mathcal{CRP}_{KCR}(\Delta \cup \mathcal{A})$
   - $\nexists j \in \{1, \ldots, m\} : \{c_{j1}, \ldots, c_{jn_j}, k_1, \ldots, k_n\} \vdash FALSCH$

Dieses Ergebnis erhält man aufgrund der Konstruktion von *KoResB*. Notwendig ist zuallererst das Terminieren von *KoResB* für alle Eingaben.
Dies ist gewährleistet, da die Literale der für das Constraint Resolutionsprinzip $\mathcal{CRP}_{KCR}$ zugrundeliegenden Constraint Klauseln *reduziert sind* (Definition 5.1). Da *KoResB* mit $\mathcal{CRP}_{KCR}$ eine Erweiterung des Resolutionskalküls ist, für das die Entscheidbarkeit des Resolutionskalküls vererbt wird, ist das beschriebene Verhalten einzusehen.

6.2 Die Eingabe

Die Eingabe von *KoResB* ist eine Constraint Klauseltheorie Δ und eine auf Widerlegbarkeit in dieser Constraint Klauseltheorie zu überprüfende Menge von Klauseln und Constraint Klauseln. Beides wurde in Kapitel 5 bereits beschrieben.

6.2.1 Literale

Literale bestehen aus den Komponenten

- dem Vorzeichen „-" (optional)

- einem Atom bestehend aus
  - dem Prädikatnamen
  - einer Argumentliste mit der dem Prädikatnamen entsprechenden Länge

Die von mir hierfür gewählte Gofer - Darstellung sieht folgendermaßen aus:

```
> data Arg   = X String Int | K String
> data Pred  = P String [Arg]
> data Lit a = Pos a | Neg a
```

Hieraus ergibt sich für Lit Pred ein Ausdruck der Form

`Pos P String [Arg] bzw. Neg P String [Arg]`

mit *String* als Prädikatnamen und der dazugehörigen Argumentliste.

Im folgenden Beispiel stelle ich ein vom Benutzer eingegebenes Literal seiner internen Gofer - Repräsentation gegenüber:

- `"-Nachbar(Tisch3,X2)"`

- `Neg P "Nachbar" [K "tisch3", X "x" 2]`

6.2.2 Klauseln und Constraint Klauseln

Nachdem die Literale feststehen, wird in den folgenden beiden Syntaxdiagrammen beschrieben, wie diese Literale zu Kauseln und Constraint Klauseln kombiniert werden.

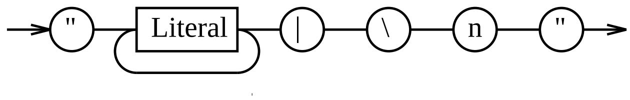

Abbildung 6.1: Darstellung einer Eingabe Klausel

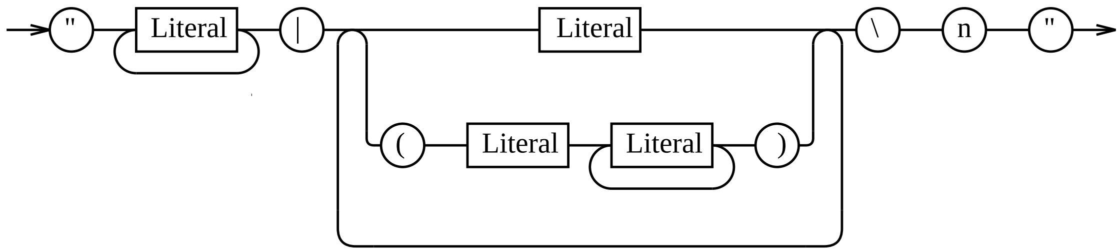

Abbildung 6.2: Darstellung einer Eingabe Constraint Klausel

Folgendes Beispiel zeigt die vom Benutzer eingegebene String - Darstellung einer Constraint Klausel und die interne Darstellung bzgl. *KoResB*:

```
? parseCK "-Abgeordneter(X1)Wahlkreis(X1,X2)Landesliste(X1,X3)|
  -UMandat(X1)\n"

[([Neg (P "Abgeordneter" [X "x" 1]),
  Pos (P "Wahlkreis" [X "x" 1, X "x" 2]),
  Pos (P "Landesliste" [X "x" 1, X "x" 3])],
  [[Neg (P "UMandat" [X "x" 1])]])]

(2497 reductions, 3391 cells)
```

6.2.3 Parser für die CKT

Die Parserfunktion „parseCKT" liest die Constraint Klausel Theorie (CKT) ein. Hierbei unterteilt die im Standard Prelude definierte Funktion „lines" den String der Eingabedatei anhand des Zeilenumbruchzeichens „bs n" in Zeilen. Um die Eingabe der einzelnen Klauseln benutzerfreundlicher zu gestalten, wurde darauf verzichtet nach jedem Literal ein Blank zu setzen (siehe XXXXXXXXX). Dies hat zur Folge das ,,word" (standard prelude) nicht genutzt werden kann.

```
> parseCKT::String->[CCL]
> parseCKT ""  = []
> parseCKT ein = map parseccl (lines ein)
> parseccl::String->CCL
> parseccl as
>     | last as == '|' = (parselitli as,[])
>     | otherwise      = (parselitli (takeWhile (/= '|') as),
>                         parsecon (tail (dropWhile (/= '|') as)))
```

Der bei den Eingabeklauseln als Markierungssymbol gesetzte vertical tab ,,|" übergibt den Kernel und Constraint einer Klausel an ,,parselitli" beziehungsweise ,,parsecon".

```
> parsecon::String->[[Lit Pred]]
> parsecon ""  = []
> parsecon as
>     |  head as == '('
>             = (parselitli (tail (kopflitli as))):
>              (parsecon (restlitli as (length (kopflitli as) +1)))
>     | (head as == '-') || (isUpper (head as))
>             = [parselit (takeUntil (== ')') as)]:
>              (parsecon (tail (dropWhile (/= ')') as)))
> kopflitli::String->String
> kopflitli (a:as)
>     | a==')' && head as == ')' = [a]
>     | otherwise                = a:(kopflitli as)
> restlitli::String->Int->String
> restlitli as 0 = as
> restlitli as n = restlitli (tail as) (n-1)
> parselitli::String->[Lit Pred]
> parselitli "|" = []
> parselitli ""  = []
```

```
> parselitli as  = (parselit (takeUntil (== ')') as)):
>                  (parselitli (tail (dropWhile (/= ')') as)))
> parselit::String->Lit Pred
> parselit lit
>  | head lit == '-' && isUpper (head (tail lit))
>       = Neg (parsepred (tail lit))
>  | isUpper (head lit)
>       = Pos (parsepred lit)
> parsepred::String->Pred
> parsepred pred  =
>  (P (takeWhile (/= '(') pred)
>        (parseargli (tailh (dropWhile (/= '(') pred))))

> parseargli::String->[Arg]
> parseargli ""  = []
> parseargli argli
>   | elem ',' argli
>              = (parsearg (takeWhile (/= ',') argli )):
>                 (parseargli (tailh (dropWhile (/= ',') argli)))
>   | otherwise = [parsearg (takeWhile (/= ')') argli)]
> parsearg::String->Arg
> parsearg arg
>   | isUpper (head arg)  && length arg == 2 =
>  X "x" (ord (last arg) - 48)
>   | isLower (head arg) = K arg
> tailh::String->String
> tailh [] = ""
> tailh as = tail as
```

6.2.4 Eingabetest

Die Funktion „eingabetest" erhält die geparste CKT als Eingabeargument. Falls es Prädikate gibt, die mit unterschiedlich langen Argumentlisten auftauchen, wird eines davon als einelementige Liste ausgegben. „main" liefert daraufhin den Namen des Prädikats. Nach Korrektur durch den Benutzer wird die Eingabe erneut nach Prädikaten mit fehlerhaften Argumentlisten abgesucht. Falls alle Argumentlisten die korrekten Längen besitzen, ist die Ausgabe die leere Liste. Nun kann die Eingabeberechnung starten.

```
> eingabetest::[CCL]->[Pred]
> eingabetest [] = []
> eingabetest as = eingabetesth (allepred as)
> allepred::[CCL]->[Pred]
> allepred [] = []
> allepred ((a,b):as) = (allepredh a)++(allepredcon b)++(allepred as)
> allepredh::[Lit Pred]->[Pred]
> allepredh [] = []
> allepredh ((Pos a):as) = a:(allepredh as)
> allepredh ((Neg a):as) = a:(allepredh as)
> allepredcon::[[Lit Pred]]->[Pred]
> allepredcon []     = []
> allepredcon (a:as) = (allepredh a)++(allepredcon as)
> eingabetesth::[Pred]->[Pred]
> eingabetesth [] = []
> eingabetesth (a:as)
>    | testeargli a as /= [] = [a]
>    | otherwise = eingabetesth as
> testeargli::Pred->[Pred]->[Pred]
> testeargli _ [] = []
> testeargli (P a b) ((P c d):bs)
>    | a==c && length b /= length d = [P a b]
>    | otherwise = testeargli (P a b) bs
```

6.3 Bestimmen eines Unifikators zweier Argumentlisten

Zentraler Punkt des von *KoResB* durchgeführten Constraint Resolutionsprinzips ist das Bestimmen eines Unifikators zweier potentieller Resolutionsliterale. Da als Argumente der Argumentlisten nur Variable und Konstante zugelassen sind, kann durch einmaliges simultanes Durchlaufen der beiden Argumentlisten durch *findunifargli* der Unifikator - falls beide Argumentlisten unifizierbar sind - bestimmt werden. Die Beim Funktionsaufruf von *findunifargli* umgesetzte Vorgehensweise veranschaulicht Abbildung 6.3.

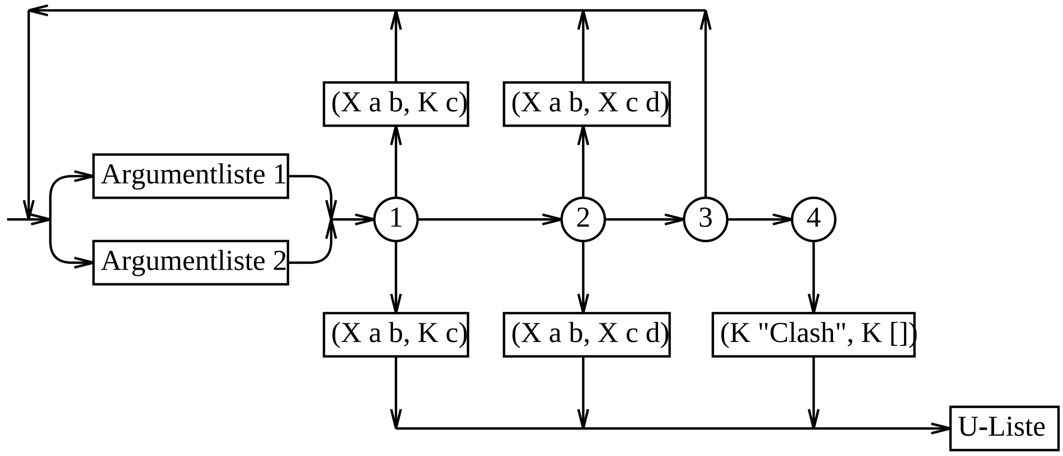

Abbildung 6.3: Unifizieren zweier Argumentlisten

Die Argumentlisten der beiden Literale werden paarweise abgearbeitet. Für jedes der Argumentpaare trifft genau einer der Fälle 1 bis 4 zu.

- **Fall 1**: Eines der Argumente ist eine Konstante, das andere eine Variable. Es werden folgende Aktionen ausgeführt:

  1. Das Paar wird als Substitution in die Unifikatorliste aufgenommen.
  2. Das Paar wird als Substitution auf die beiden Argumentlisten angewendet.
  3. Das nächste Argumentpaar wird auf Anwendbarkeit für die Fälle 1 bis 4 bereitgestellt.

- **Fall 2**: Zwei ungleiche Variable treffen aufeinander. Die hierauf durchzuführenden Aktionen sind denen des ersten Falles identisch, die Substitution ist diesmal ein Variablenpaar.

- **Fall 3**: Die Argumente des Argumentpaars sind identisch. Das nächste Argumentpaar der Ausgangslisten wird bearbeitet.

- **Fall 4**: Zwei nicht identische Konstanten bilden das Argumentenpaar. Dieser Fall entspricht dem Konstantenclash, und somit sind beide Argumentlisten nicht unifizierbar. *findunifargli* terminiert. Die Unifikatorliste erhält die Marke (K „Clash",K []) als letzten Eintrag.

findunifargli terminiert also entweder mit (K „Clash",K []) als letztem Listeneintrag, oder indem er eine Unifikatorliste berechnet hat und beide Argumentlisten vollständig abgearbeitet sind.

```
> findunifargli::[Arg]->[Arg]->[(Arg,Arg)]
> findunifargli [] [] = []
> findunifargli ((K a):as) ((K b):bs)
>   | a==b = findunifargli as bs
>   | otherwise = [(K "Clash",K [])]
> findunifargli ((X a b):as) ((K c):bs) =
>       (X a b,K c):(findunifargli (ersetze (X a b) (K c) as)
>                                  (ersetze (X a b) (K c) bs))
> findunifargli ((K a):as) ((X b c):bs) =
>       (X b c,K a):(findunifargli (ersetze (X b c) (K a) as)
>                                  (ersetze (X b c) (K a) bs))
> findunifargli ((X a b):as) ((X c d):bs)
>   | (X a b)/=(X c d) = (X a b,X c d):
>                         (findunifargli (ersetze (X a b) (X c d) as)
>                                        (ersetze (X a b) (X c d) bs))
>   | otherwise = findunifargli as bs
>
> ersetze::Arg->Arg->[Arg]->[Arg]
> ersetze a b [] = []
```

```
> ersetze a b (c:cs)
>   | a==c = b:(ersetze a b cs)
>   | otherwise = c:(ersetze a b cs)
```

6.4 Drei Anwendungsbeispiele

In diesem Abschnitt stelle ich zuerst die Überhangmandat Aussage im Zusammenhang mit dem Einzug in den deutschen Bundestag vor.
Es folgt die Wahl einer Constraint Klauseltheorie, die diese Regel enthält.
Hierauf folgen drei verschiedene Anwendungsbeispiele.

6.4.1 Die Überhangmandat Aussage

Bei den Wahlen zum deutschen Bundestag gibt es eine Sonderregelung, die es erlaubt, daß Wahlkandidaten in den Bundestag einziehen, obwohl sie nach dem prozentualen Anteil an Zweitstimmen *nicht* mehr berufen würden, aber durch den Sieg im Wahlkreis ein sogenanntes Überhangmandat erhalten. Da im Regelfall die Wahlkandidaten entweder direkt über den Wahlkreis oder die Landesliste in den Bundestag berufen werden und ein Überhangmandat selten erzielt wird, handelt es sich bei diesem Sachverhalt um eine Ausnahmeregelung, der man in Form der folgenden Constraint Klausel gerecht werden kann:

```
-Abgeordneter(X1)Wahlkreis(X1,X2)Landesliste(X1,X3)|-UMandat(X1)
```

6.4.2 CKT mit Überhangmandat - Regel

Mit der in 6.4.1 vorgestellten Überhangmandat Aussage und einigen fiktiven Bundestagsabgeordneten läßt sich bereits eine anschauliche Constraint Klausel Thaeorie realisieren. Die Klauseln und Constraint Klauseln gebe ich bereits als Strings an:

```
d0 = "-Abgeordneter(X1)Wahlkreis(X1,X2)Landesliste(X1,X3)|-UMandat(X1)\n"
```

```
a1 = "Abgeordneter(Schmidt)|Abgeordneter(Schmidt)\n"
a2 = "Abgeordneter(Nadoll)|Abgeordneter(Nadoll)\n"
a3 = "Abgeordneter(Schulze)|Abgeordneter(Schulze)\n"
a4 = "Abgeordneter(Appold)|Abgeordneter(Appold)\n"
a5 = "Abgeordneter(Kirchner)|Abgeordneter(Kirchner)\n"
a6 ="Abgeordneter(Uhl)|Abgeordneter(Uhl)\n"
akom = a1++a2++a3++a4++a5++a6

w1 = "Wahlkreis(Schmidt,Soest)|\n"
w2 = "Wahlkreis(Nadoll,Viersen)|\n"
w3 = "-Wahlkreis(Schulze,X1)|\n"
w4 = "-Wahlkreis(Appold,X1)|\n"
w5 = "-Wahlkreis(Kirchner,X1)|\n"
w6 ="-Wahlkreis(Uhl,X1)|\n"
wkom = w1++w2++w3++w4++w5++w6

l1 = "Landesliste(Schulze,Bayern)|\n"
l2 = "Landesliste(Appold,Bayern)|\n"
l3 = "Landesliste(Kirchner,Bayern)|\n"
l4 = "-Landesliste(Schmidt,X1)|\n"
l5 = "-Landesliste(Nadoll,X1)|\n"
l6 = "-Landesliste(Uhl,X1)|\n"
lkom = l1++l2++l3++l4++l5++l6
```

6.4.3 Ein Beispiel mit inkonsistenter CKT

Die in 6.4.2 aufgeführten Klauseln und Constraint Klauseln ergeben eine Constraint Klauseltheorie mit Konsistenztyp *konsistent mit nichtleerer Kontrollmenge*, da keine leere Klausel aus ihnen resolviert werden kann, aber sehr wohl eine leere Constraint Klausel, da man durch Anwendung des $\mathcal{CRP}_{KCR}$ auf

```
d0 = "-Abgeordneter(X1)Wahlkreis(X1,X2)Landesliste(X1,X3)|
```

```
        -UMandat(X1)\n"
  a6 ="Abgeordneter(Uhl)|Abgeordneter(Uhl)\n"
  w6 ="-Wahlkreis(Uhl,X1)|\n"
  l6 = "-Landesliste(Uhl,X1)|\n"
```

folgende leere Constraint Klausel erhält:

```
  Abgeordneter(Uhl)-UMandat(Uhl)
```

In diesem Beispiel möchte ich jedoch eine inkonsistent CKT von *KoResB* bearbeiten lassen, und füge aus diesem Grund noch folgende Klausel ein:

```
  w4n = "Wahlkreis(Appold,X1)|\n"
```

KoResB wird mit

```
koresb::String->String->String
```

aufgerufen, wobei der erste String die CKT ist und der zweite die Anfrage in Form einer Menge von Klauseln und Constraint Klauseln.

Ich bilde

```
delta1 = d0++akom++wkom++lkom++w4n
anfrage1 = l6 (Sie hat keine Auswirkung auf das Ergebnis)
```

und erhalte folgendes Reultat:

```
? koresb delta1 anfrage1
 Die Constraint Klauseltheorie ist inkonsistent.
(442939 reductions, 794085 cells, 10 garbage collections)
```

6.4.4 Beispiel CKT konsistent mit nichtleerer Kontrollmenge aber keine neue LCK resolvierbar

Ich definiere delta2 und anfrage1 wie folgt:

```
delta2 = d0++akom++wkom++lkom
anfrage1 = "Landesliste(Schulze,Bayern)|\n"
```

und gebe folgenden Funktionsaufruf:

? koresb delta2 anfrage1

der das Resultat

> Die Constraint Klauseltheorie ist konsistent mit nichtleerer Kontrollmenge. Als Kontrollmenge wurde ausgewaehlt: -Abgeordneter(Uhl) Es konnte keine neue Constraint Klausel gefunden werden, A kann also nicht widerlegt werden.

liefert.

6.4.5 Beispiel CKT konsistent mit nichtleerer Kontrollmenge und einer neu resolvierten LCK

In diesem Beispiel übernehme ich delta1 und bilde eine neue Anfrage:

```
anfrage2 = "-Abgeordneter(Appold)|Abgeordneter(Appold)\n"
```

Das Resultat ist:

```
? crp delta2 anfrage2
Die Constraint Klauseltheorie ist konsistent mit
 nichtleerer  Kontrollmenge.
 Als Kontrollmenge wurde ausgewaehlt: -Abgeordneter(Uhl)
 Es konnte genau eine neue LCK bestimmt werden, diese ist:

|Abgeordneter(Appold)
```

Anhang A

KoResB Quellcode

Zunächst folgt die Übersicht über komplexere Funktionen, die in KoResB auftreten, und schließlich folgt der Quellcode.

A.1 Funktionenübersicht

```
DATENTYPEN
CK f"ur Constraint Klausel,
KM f"ur Kontrollmenge

INSTANZEN
Definitionen der ben"otigten Mengenrelationen

PARSECKT
parseCKT::String->[CK]

PARSEDNF
parseDNF::String->[CK]

EINGABETEST
eingabetest::[CK]->[Pred]
```

```
ALLERESOLV
alleresolv::[CK]->[CK]

AUFXY
aufx,aufy::CK->CK

RESZWEICK
reszweick::CK->CK->[CK]

FINDUNIFARGLI
findunifargli::[Arg]->[Arg]->[(Arg,Arg)]

SUBSTLIT
substlit::Lit Pred->[(Arg,Arg)]->Lit Pred

SUBSTCK
substck::CK->[(Arg,Arg)]->CK

KK
kk::CK->CK->CK

ALLERESOLVVONMIT
alleresolvvonmit::[CK]->[CK]->[CK]

NEUE
neue,neueh::[CK]->[CK]->[CK]

TAUT
taut::[CK]->[CK]

SUBSUM
subsum::[CK]->[CK]

SUNIF
```

```
sunif::Lit Pred->Lit Pred->[(Arg,Arg)]

FAKTOR
faktor::[CK]->[CK] -- faktorisierbare ck bleiben erhalten.

BENUTZER KM
benutzerKM::(KM,[CK])->[Lit Pred]->(KM,[CK])

AUSGABE
cklialsstring::[CK]->String

RESOLV
resolv::String->String
resolv sys = cklialsstring (sos [] (parseCKT sys))

KORESB
koresb::String->String->String
koresb theorie anfrage

SOS
sos::[CK]->[CK]->[CK]

FINDBED
findbed::[[Lit Pred ]]->KM->[[[(Arg,Arg)]]]

OPTUNDPERMUT
optundpermut,optundpermuth ::[[[(Arg,Arg)]]]->[[(Arg,Arg)]]

STRINGAUSGABE

FINDKM
findKM::[[[Lit Pred]]]->KM

KMKONS
```

```
kmkons::KM->Bool

TEST KM -DNF LCK
Dieser Test gilt nur fuer konsistente KM
test::KM->[CK]->Bool
```

A.2 Quellcode KoResB

```
-- ----------------------------DATENTYPEN
type CK   = ([Lit Pred],[[Lit Pred]])
type KM    = [Lit Pred]

data Arg   = X String Int | K String
data Pred  = P String [Arg]
data Lit a = Pos a | Neg a
-- ----------------------------DATENTYPEN END
-- ----------------------------INSTANZEN
-- Auf folgenden Datentypen und Datenstrukturen wird die
-- Gleichheitsrelation definiert:
-- Arg, Pred, Lit Pred, [Lit Pred], [[Lit Pred]],
-- ([Lit Pred],[[Lit Pred]])

instance Eq Arg where
  X a b == X c d = a==c && b==d
  K a == K b     = a==b
  _ == _         = False
-- ---------
instance Eq Pred where
  P a b == P c d = a==c && b==d
-- ---------
instance Eq Pred => Eq (Lit Pred) where
  Pos a == Pos b = a==b
```

```
  Neg a == Neg b = a==b
  _ == _         = False

-- ---------
instance Eq (Lit Pred) => Eq [Lit Pred] where -- Mengengleichheit
  a==b = equal a b a b
--  ([x|elem x a,not (elem x b)]==[]) &&
--  ([x|elem x b,not (elem x a)]==[])

equal::Eq (Lit Pred) => [Lit Pred]->[Lit Pred]->
                        [Lit Pred]->[Lit Pred]->Bool
equal [] [] [] [] = True
equal [] [] a b = True
equal [] _ _ _ = False
equal _ [] _ _ = False
equal (a:as) (b:bs) c d = elem a d &&
                          elem b c &&
                          equal as bs c d

litinli::Eq (Lit Pred) => Lit Pred->[Lit Pred]->Bool
litinli a [] = False       -- elem a d
litinli a (b:bs) = a==b || litinli a bs
-- ---------

instance Eq ([Lit Pred]) => Eq ([Lit Pred],[[Lit Pred]]) where
 (a,b)==(c,d) = a==c && b==d

instance Eq ([Lit Pred]) => Eq ([[Lit Pred]]) where
 a==b = equal2 a b a b

equal2:: Eq [Lit Pred] => [[Lit Pred]]->[[Lit Pred]]->
                          [[Lit Pred]]->[[Lit Pred]]->Bool
equal2 [] [] [] [] = True
equal2 [] [] a b   = True
```

```
equal2 [] _ _ _     = False
equal2 _ [] _ _     = False
equal2 (a:as) (b:bs) c d = elem a d &&
                           elem b c &&
     equal2 as bs c d
-- ------------------------------------INSTANZEN END
litname::Lit Pred->String
litname (Pos (P a b)) = a
litname (Neg (P a b)) = a
-- ---------
constraint::CK->[[Lit Pred]]
constraint (a,b) = b

instance Eq Arg => Eq (Arg,Arg) where -- wird nicht gebraucht
  (a,b)==(c,d) = a==c && b==d || a==d && b==c


-- ------------------------------------------PARSESYS

-- EINGABE: Die CK der Systemklauselmenge werden in eine Datei
-- "SysCK" eingegben. Jede CK wird mit \n abgeschlossen.
-- Beim Lesen wird diese Datei durch "lines" (Standard Prelude)
-- als eine Liste von Strings Interpretiert. Diese Strings werden
-- sukzessive in die interne CK-Darstellung gebracht.

parseCKT::String->[CK]
parseCKT ""  = []
parseCKT ein = map parseck (lines ein)


parsemengevonCK::String->[CK]
parsemengevonCK ""  = []
parsemengevonCK ein = map parseck (lines ein)
```

```
-- Aussagen haben als letztes Eingabesymbol immer ein "|". Ihr
-- Constraint wird folgerichtig als "[]" interpretiert. Der
-- Kernel wird durch "parselitli" geparst.
-- Bedingte Aussagen haben einen nichtleeren Konstraint, der
-- von "parsecon" behandelt wird.

parseck::String->CK
parseck as
   | last as == '|' = (parselitli as,[]) -- Fact, leerer Constraint
   | otherwise      = (parselitli (takeWhile (/= '|') as),
                      parsecon (tail (dropWhile (/= '|') as)))

parsecon::String->[[Lit Pred]]
parsecon ""  = []
parsecon as
   |  head as == '('
        = (parselitli (tail (kopflitli as))):
          (parsecon (restlitli as (length (kopflitli as) +1)))
| (head as == '-') || (isUpper (head as))
         = [parselit (takeUntil (== ')') as)]:
          (parsecon (tail (dropWhile (/= ')') as)))

kopflitli::String->String
kopflitli (a:as)
   | a==')' && head as == ')' = [a]
| otherwise                = a:(kopflitli as)
restlitli::String->Int->String
restlitli as 0 = as
restlitli as n = restlitli (tail as) (n-1)

parselitli::String->[Lit Pred]
parselitli "|" = []
parselitli ""  = []
```

```
parselitli as  =     ((parselit (takeUntil (== ')') as)):
                      (parselitli (tail (dropWhile (/= ')') as))))

parselit::String->Lit Pred
parselit lit
  | head lit == '-' && isUpper (head (tail lit))
       = Neg (parsepred (tail lit))
  | isUpper (head lit)
       = Pos (parsepred lit)

parsepred::String->Pred
parsepred pred  =
(P (takeWhile (/= '(') pred)
       (parseargli (tailh (dropWhile (/= '(') pred))))
parseargli::String->[Arg]
parseargli ""  = []
parseargli argli
   | elem ',' argli
           = (parsearg (takeWhile (/= ',') argli )):
                (parseargli (tailh (dropWhile (/= ',') argli)))
   | otherwise = [parsearg (takeWhile (/= ')') argli)]
parsearg::String->Arg
parsearg arg
  | isUpper (head arg)  && length arg == 2 =
X "x" (ord (last arg) - 48)
  | otherwise = K arg
--  | isLower (head arg) = K arg

tailh::String->String
tailh [] = ""
tailh as = tail as
-- -------------------------------------PARSESYS END
-- -----------------------------------------PARSEDNF
parseDNF::String->[CK]
```

```
parseDNF ""  = []
parseDNF ein
  | head ein == '('
              = (neg (parselitli (tail (take ((klammerend ein)-1) ein))),[]):
                (parseDNF (drop (klammerend ein) ein))
  | otherwise = ([neglit (parselit (takeUntil (== ')') ein))],[]):
                (parseDNF (tail (dropWhile (/= ')') ein)))
--    where ein = lines einumbr
klammerend::String->Int
klammerend ein
  | head ein == ')' && head (tail ein) == ')' = 2
  | otherwise = 1 + (klammerend (tail ein))



-- ----------------------------------------PARSEDNF END
-- -----------------------------------------EINGABETEST
-- Nachdem die Stringeingabe geparst vorliegt, wird sie zunaechst
-- auf Korrektheit bzgl. der Argiablenlistenlaengen gleichnamiger
-- Praedikate gecheckt.
-- Eingabetest

eingabetest::[CK]->[Pred]
eingabetest [] = []
eingabetest as = eingabetesth (allepred as)
allepred::[CK]->[Pred]
allepred [] = []
allepred ((a,b):as) = (allepredh a)++(allepredcon b)++(allepred as)
allepredh::[Lit Pred]->[Pred]
allepredh [] = []
allepredh ((Pos a):as) = a:(allepredh as)
allepredh ((Neg a):as) = a:(allepredh as)
allepredcon::[[Lit Pred]]->[Pred]
allepredcon []     = []
```

```
allepredcon (a:as) = (allepredh a)++(allepredcon as)
eingabetesth::[Pred]->[Pred]
eingabetesth [] = []
eingabetesth (a:[]) = []
eingabetesth (a:as)
  | testevarli a as /= [] = testevarli a as
  | otherwise = eingabetesth as
testevarli::Pred->[Pred]->[Pred]
-- testet ob alle Vorkommen des ersten Praedikates in der Prae-
-- dikatenliste gleichlange Argiablenlisten haben.
testevarli _ [] = []
testevarli (P a b) ((P c d):bs)
  | a==c && length b /= length d = [P a b]
  | otherwise = testevarli (P a b) bs

pname::[Pred]->String
pname [P a b] = a
-- -----------------------------------------EINGABETEST END
-- ---------------------------------------------ALLERESOLV1

-- alleresolv findet alle in einem Schritt resolvierbaren CK

alleresolv::[CK]->[CK]
alleresolv [] = []
alleresolv (a:[]) = [] -- keine Selbstresolution
alleresolv (a:as) = (alleresolvh (aufx a) as)++(alleresolv as)
alleresolvh::CK->[CK]->[CK]
alleresolvh a [] = []
alleresolvh a (b:bs) = (reszweick a (aufy b))++
                       (alleresolvh a bs)
-- -----------------------------------------ALLERESOLV1 END
-- ------------------------------------------------AUFX,AUFY
{-
```

Sollen zwei CCL resolviert werden, ist zunächst eine Variablenumbenennung noetig. Die verschiedenen Variablen der beiden Klauseln werden hierbei auf „X" beziehungsweise „Y" umbenannt und erhalten eine Integer-Nummer. Da „x" und „y" in Gofer nicht explizit als Argumente weitergegeben werden koennen, müssen hierfür zwei identische Prozeduren geschrieben werden. Die folgende Prozedur„aufx" leistet die Variablenumbenennung einer CCL, indem zunächst eine Liste aller Argumente der in der CCL vor- kommenden Literale bestimmt wird (arglivonccl). Die Funktion „umnennerxh" bildet nun für die Variablen dieser Liste geeignete Substitutionen, die anschliessend als Substitutionsliste durch „substccl" auf die umzubenennende Klausel angewendet wird.

```
-}
aufx,aufy::CK->CK
aufx a = substck a (umnennerx (ohnekonst (varlivonck a)))
aufy a = substck a (umnennery (ohnekonst (varlivonck a)))

-- Die Liste aller in der ck enthaltenen Argiablen wird bestimmt.
varlivonck::CK->[Arg]
varlivonck (a,b) = nub ((varlivonlitli a)++
                         (concat (map varlivonlitli b)))

varlivonlitli::[Lit Pred]->[Arg]
varlivonlitli [] = []
varlivonlitli (a:as) = (varlivonlit a)++(varlivonlitli as)

varlivonlit::Lit Pred->[Arg]
varlivonlit (Pos (P a b)) = b
varlivonlit (Neg (P a b)) = b

-- varlivonck enthaelt noch Konstanten. Diese muessen raus.
ohnekonst::[Arg]->[Arg]
ohnekonst [] = []
ohnekonst ((K a):as) = ohnekonst as
ohnekonst ((X a b):as) = (X a b):(ohnekonst as)
```

```
-- Die konstantenfreie Argiablenliste wird nun von umnennerxy
-- behandelt.
umnennerx,umnennery::[Arg]->[(Arg,Arg)]
umnennerx [] = []
umnennerx as = umnennerxh as 1

umnennerxh,umnenneryh::[Arg]->Int->[(Arg,Arg)]
umnennerxh [] _ = []
umnennerxh (a:as) i  = (a,X "x" i):(umnennerxh as (i+1))

umnennery [] = []
umnennery as = umnenneryh as 1

umnenneryh [] _ = []
umnenneryh (a:as) i = (a,X "y" i):(umnenneryh as (i+1))

-- -----------------------------------------------------AUFX,AUFY END
-- --------------------------------------------------------RESZWEICK
-- reszweick Alle Resolventen zweier CK werden bestimmt.

reszweick::CK->CK->[CK]
reszweick (a,b) (c,d)
     = reszweickh (a,(a,b)) (c,(c,d))
reszweickh::([Lit Pred],CK)->([Lit Pred],CK)->[CK]
reszweickh ([],ck1) (_,ck2) = []
reszweickh ((a:as),ck1) ([],(c,d))
     = reszweickh (as,ck1) (c,(c,d))
reszweickh ((a:as),(b,c)) ((d:ds),(e,f))
  | a == neglit d
     = [((ohnelit a b)++(ohnelit d e), nub c++f)] -- langt bereits
  | litname a == litname d && vorz a /= vorz d
    && last u /= (K "Clash",K "")
     = substck ((ohnelit a b)++(ohnelit d e),nub c++f) u
    : reszweickh ((a:as),(b,c)) (ds,(e,f))
```

```
  | otherwise
     = reszweickh ((a:as),(b,c)) (ds,(e,f))
    where u = findunifvarli (varlivonlit a) (varlivonlit d)




kernel::CK->[Lit Pred]
kernel (a,b) = a

vorz::Lit Pred->String
vorz (Pos (P a b)) = "pos"
vorz (Neg (P a b)) = "neg"

-- findesubsth::Lit Pred->[Arg]->[(Arg,Arg)]
-- findesubsth (Pos (P a b)) c = findesubstvarli b c
-- findesubsth (Neg (P a b)) c = findesubstvarli b c
-- gleichnamige Praedikate haben gleichlange Argiablenlisten
-- ---------------------------------------------------RESZWEICK END
-- ----------------------------------------------------FINDMGUVARLI
findunifvarli::[Arg]->[Arg]->[(Arg,Arg)]
findunifvarli [] [] = []
findunifvarli ((K a):as) ((K b):bs)
  | a==b = findunifvarli as bs
  | otherwise = [(K "Clash",K [])]
findunifvarli ((X a b):as) ((K c):bs) =
      (X a b,K c):(findunifvarli (ersetze (X a b) (K c) as)
                                 (ersetze (X a b) (K c) bs))
findunifvarli ((K a):as) ((X b c):bs) =
      (X b c,K a):(findunifvarli (ersetze (X b c) (K a) as)
                                 (ersetze (X b c) (K a) bs))
findunifvarli ((X a b):as) ((X c d):bs)
  | (X a b)/=(X c d) = (X a b,X c d):
                       (findunifvarli (ersetze (X a b) (X c d) as)
```

```
                                       (ersetze (X a b) (X c d) bs))
  | otherwise = findunifvarli as bs

ersetze::Arg->Arg->[Arg]->[Arg]
ersetze a b [] = []
ersetze a b (c:cs)
  | a==c = b:(ersetze a b cs)
  | otherwise = c:(ersetze a b cs)
-- --------------------------------------------------FINDMGUVARLI END
-- ----------------------------------------------------------SUBSTLIT

-- Bottom up von substlit: substvar->substvarli->substlit
substlit::Lit Pred->[(Arg,Arg)]->Lit Pred
substlit (Pos (P a b)) c =
         (Pos (P a (substvarli b c)))
substlit (Neg (P a b)) c =
         (Neg (P a (substvarli b c)))
substvarli::[Arg]->[(Arg,Arg)]->[Arg]
substvarli a [] = a -- doppelt gemoppelt ??
substvarli [] b = []
substvarli (a:as) b =
           (substvar a b):(substvarli as b)

substvar::Arg->[(Arg,Arg)]->Arg
substvar a [] = a
substvar a ((b,c):bs)
  | a==b = c
  | otherwise = substvar a bs
-- ------------------------------------------------------SUBSTLIT END
-- ----------------------------------------------------------SUBSTCK
-- Bottom up von substck:
-- substvar->substvarli->substlit->substlitli->substck
substck::CK->[(Arg,Arg)]->CK
substck (a,b) c = ((substlitli a c),
```

```
                    (substcon b c))

substcon::[[Lit Pred]]->[(Arg,Arg)]->[[Lit Pred]]
substcon [] _ = []
substcon (a:as) b = (substlitli a b):(substcon as b)

substlitli::[Lit Pred]->[(Arg,Arg)]->[Lit Pred]
substlitli a [] = a
substlitli [] c = []
substlitli (a:as) c = (substlit a c):(substlitli as c)

-- ------------------------------------------------------SUBSTCK END
-- ----------------------------------------------------------------KK
-- kk::CK->CK->CK
-- kk (a,b) (c,d) = (nub(a++c),nub(b++d))
ohnelit::Lit Pred->[Lit Pred]->[Lit Pred]
ohnelit a [] = []
ohnelit a (b:bs)
  | a==b = bs
  | otherwise = b:(ohnelit a bs)
neglit::Lit Pred->Lit Pred
neglit (Pos a) = Neg a
neglit (Neg a) = Pos a
nubcon::[[Lit Pred]]->[[Lit Pred]]
nubcon [] = []
nubcon (a:as)
    = (a:(ohnelitli as a))++(nubcon as)
ohnelitli::[[Lit Pred]]->[Lit Pred]->[[Lit Pred]]
ohnelitli [] [] = []

-- ANWENDUNG beispielsweise a==nubcon a b &&
--                          a==nubcon b a
-- ------------------------------------------------------------KK END
-- --------------------------------------------------ALLERESOLVVONMIT
```

```
-- Die obige Funktion alleresolv findet alle Resolventen einer
-- gegebenen Klauselmenge, die in einem Resolutionsschritt erreich-
-- bar sind. Fr die Breitensuche ist noch eine Funktion erforder-
-- lich, die alle Resolventen zwischen zwei disjunkten Klauselmengen
-- bestimmt.
-- alleresolvvonmit

alleresolvvonmit::[CK]->[CK]->[CK]
alleresolvvonmit [] [] = []
alleresolvvonmit [] a = []
alleresolvvonmit a [] = []
alleresolvvonmit (a:as) b =
        (alleresolvvonmith (aufx a) b)++(alleresolvvonmit as b)
alleresolvvonmith::CK->[CK]->[CK]
alleresolvvonmith a [] = []
alleresolvvonmith a (b:bs) =
  (reszweick a (aufy b))++(alleresolvvonmith a bs)
-- ----------------------------------------------ALLERESOLVVONMIT END
-- ----------------------------------------------------------------NEUE
-- Als weiteres wird eine Funktion benoetigt, mit der man aus einer
-- Menge von Resolventen die neu gefundenen Resolventen bestimmen
-- kann.
-- neue
neue,neueh::[CK]->[CK]->[CK]
neue as bs = neueh as (nub bs)
neueh [] [] = []
neueh as [] = []
neueh as (b:bs)
  | elem b as = neue as bs
  | otherwise = b:(neue as bs)
-- ------------------------------------------------------------NEUE END
-- ----------------------------------------------------------------TAUT
taut::[CK]->[CK]
taut [] = []
```

```
taut as
     = neue (filter tautbck as) as -- incl doppelte entfernen
tautbck::CK->Bool
tautbck ([],bs)
     = False
tautbck ((a:as),bs)
   | elem (neglit a) as
  = True
   | otherwise
  = tautbck (as,bs)
-- ----------------------------------------------------------TAUT END
-- ------------------------------------------------------------SUBSUM
subsum::[CK]->[CK]
subsum [] = []
subsum (a:[]) = [a]
subsum (a:as) = a:(subsum (subsumh a as))
subsumh::CK->[CK]->[CK]
subsumh a []     = []
subsumh a (b:bs)
  | subsumbool a b = subsumh a bs
  | otherwise      = b:(subsumh a bs)

subsumbool::CK->CK->Bool
subsumbool (a,b) (c,d)
     = subsumboolh (a,(a,b)) (c,(c,d))
subsumboolh::([Lit Pred],CK)->([Lit Pred],CK)->Bool
subsumboolh (as,ck1) ([],ck2) = False
subsumboolh ([],_) (_) = True
subsumboolh (a:as,(b,c)) (d:ds,(e,f))
  | litname a == litname d &&
    vorz a == vorz d &&
 lasth (sunif a d) /= (K "Clash",K "") &&
 conok (substcon f u) (substcon c u) &&
 length c == length f
```

```
   = subsumboolh (substlitli as u,substck (b,c) u)
               (substlitli e u,substck (e,f) u)
  | otherwise
      = subsumboolh (a:as,(b,c)) (ds,(e,f))
    where u = sunif a d
lasth::[(Arg,Arg)]->(Arg,Arg)
lasth [] = (K "OK",K "")
lasth (a:as) = lasth as

suniflitli::[Lit Pred]->[Lit Pred]->[Lit Pred]->[(Arg,Arg)]
suniflitli as [] ms = [(K "Clash",K "")]
suniflitli [] bs ms = []
suniflitli (a:as) (b:bs) ms
  | a == b    = suniflitli as bs ms
  | litname a == litname b &&
    vorz a    == vorz b &&
 last u    == (K "Clash",K "")
           = suniflitli (a:as) bs ms
  | litname a == litname b &&
    vorz a    == vorz b &&
 last u    /= (K "Clash",K "")
           = u ++ suniflitli (substlitli as u)
                   (substlitli ms u)
 (substlitli ms u)
    where u   = sunif a b
-- ---------------------------------------------------------------SMGU
sunif::Lit Pred->Lit Pred->[(Arg,Arg)]
sunif a b = sunifh (varlivonlit a) (varlivonlit b)
sunifh::[Arg]->[Arg]->[(Arg,Arg)]
sunifh [] []      = []
sunifh ((K a):as) ((K b):bs)
  | a == b       = sunifh as bs
  | a /= b       = [(K "Clash",K "")]
sunifh ((X a b):as) ((X c d):bs)
```

```
  | a==c && b==d = sunifh as bs
  | otherwise    = (X a b,X c d):
                   (sunifh (ersetze (X a b) (X c d) as)
        (ersetze (X a b) (X c d) bs))
sunifh ((X a b):as) ((K c):bs)
                 = (X a b,K c):
    (sunifh (ersetze (X a b) (K c) as)
        (ersetze (X a b) (K c) bs))
sunifh ((K a):as) ((X b c):bs)
                 = [(K "Clash",K "")]
-- ----------------------------------------------------------SMGU END
connub::[CK]->[CK]
connub [] = []
connub ((a,b):as) = (a,nub b):(connub as)

conok::[[Lit Pred]]->[[Lit Pred]]->Bool
conok [] _ = True
conok (a:as) bs
  | subok a bs = conok as bs
  | otherwise  = False

subok::[Lit Pred]->[[Lit Pred]]->Bool
subok as [] = False
subok as (b:bs) = subokh as b b||subok as bs
subokh::[Lit Pred]->[Lit Pred]->[Lit Pred]->Bool
subokh [] _ _ = True
subokh as [] cs = False
subokh (a:as) (b:bs) cs
  | a == b   = subokh as cs cs||subokh (a:as) bs cs
subokh (a:as) (b:bs) cs
  | litname a == litname b &&
    vorz a == vorz b &&
 last u /= (K "Clash",K "")
  = subokh (substlitli as u) cs cs||subokh (a:as) bs cs
```

```
  | otherwise = subokh (a:as) bs cs
    where u   = sunifh (varlivonlit a) (varlivonlit b)
-- ---------------------------------------------------------SUBSUM END
-- -------------------------------------------------------------FAKTOR
faktor::[CK]->[CK] -- faktorisierbare ck bleiben erhalten.
faktor []
     = []
faktor ((a,b):as)
  | a == []
     = [(a,b)] ++ faktor as
  | otherwise
     = [(a,b)] ++
    nub (faktorck (head a) (tail a) (tail a) (a,b) ++
 faktor as)

faktorck::Lit Pred->[Lit Pred]->[Lit Pred]->CK->[CK]
faktorck a [] [] ck
     = []
faktorck a [] cs ck
     =  faktorck (head cs) (tail cs) (tail cs) ck
faktorck a bs cs (b,c)
  | litname a == litname (head bs) &&
    vorz a == vorz (head bs) &&
    last u /= (K "Clash",K [])
  = (nub (substlitli b u),nub (substcon c u)):
    (faktorck a (tail bs) cs (b,c))
  | otherwise
     = faktorck a (tail bs) cs (b,c)
    where u = findunifvarli (varlivonlit a)
                        (varlivonlit (head bs))

-- ---------------------------------------------------------FAKTOR END
-- --------------------------------------------------------BENUTZER KM
benutzerKM::(KM,[CK])->[Lit Pred]->(KM,[CK])
```

```
benutzerKM ([],bs) initkm = (neg initkm,del (neg initkm,bs))
benutzerKM (as,bs) cs = ((neg cs)++as,del (neg cs++as,bs))
del::(KM,[CK])->[CK]
del (as,[]) = []
del (as,((a,b):bs))
  | kmConok (as,b) = (a,b):(del (as,bs))
  | otherwise  = del (as,bs)

kmConok::(KM,[[Lit Pred]])->Bool
kmConok (cs,[]) = True
kmConok (cs,(a:as))
  | elem ([],[]) (sos [] ((a,[]):(kmalsckli cs)))
             = False
  | otherwise = kmConok (cs,as)

kmalsckli::[Lit Pred]->[CK]
kmalsckli [] = []
kmalsckli (a:as) = ([a],[]):(kmalsckli as)

-- ---------------------------------------------------BENUTZER KM END
-- -----------------------------------------------------------AUSGABE
cklialsstring::[CK]->String
cklialsstring [] = ""
cklialsstring (a:as) = (ckalsstring a)++"\n"++(cklialsstring as)

ckalsstring::CK->String
ckalsstring (a,b)
   | b == []   = (litlialsstring a)++"|"
| otherwise = (litlialsstring a)++"|"++(conalsstring b)

conalsstring::[[Lit Pred]]->String
conalsstring [] = ""
conalsstring (a:as) = (litlialsstringc a)++(conalsstring as)
```

```
litlialsstringc::[Lit Pred]->String
litlialsstringc as
  | length as == 1 = litlialsstring as
  | otherwise     = "("++(litlialsstring as)++")"

litlialsstring::[Lit Pred]->String
litlialsstring [] = ""
litlialsstring (a:as) = (litalsstring a)++(litlialsstring as)

litalsstring::Lit Pred->String
litalsstring (Pos (P a b))
     = a ++"("++(varlialsstring b)++")"
litalsstring (Neg (P a b))
     = "-" ++ a ++"(" ++(varlialsstring b)++")"
varlialsstring::[Arg]->String
varlialsstring [] = " "
varlialsstring (a:[]) = varalsstring a
varlialsstring (a:as)
     = (varalsstring a)++","++(varlialsstring as)
varalsstring::Arg->String
varalsstring (X a b)
   = a++(show b)
varalsstring (K a)
   = a
-- -------------------------------------------------------AUSGABE END
-- ------------------------------------------------------------RESOLV
resolv::String->String
resolv sys = cklialsstring (sos [] (parseCKT sys))

resolv2::String->[CK]
resolv2 sys = (sos [] (parseCKT sys))

-- --------------------------------------------------------RESOLV END
-- ------------------------------------------------------------KORESB
```

```
koresp::String->String->String
koresp theorie anfrage

  | eingabetest (parseCKT theorie ++ parseCKT anfrage) /= []
     = "Das Praedikat '"++
    pname (eingabetest (parseCKT theorie ++ parseCKT anfrage))++
 "' kommt mit unterschiedlich langer Argumentliste vor."
  | elem ([],[]) (resolv2 theorie) = " Die Constraint"++
  " Klauseltheorie ist inkonsistent."
  | (map constraint (findLCK (resolv2 theorie))) /= []
    && findLCK (neue (resolv2 theorie) (resolv2 (theorie++anfrage))) == []
    = "Die Constraint Klauseltheorie ist konsistent mit nichtleerer " ++
     " Kontrollmenge." ++ "\n" ++
     " Als Kontrollmenge wurde ausgewaehlt: " ++
     (litlialsstring (findKM (map constraint (findLCK (resolv2 theorie))))) ++
     "\n" ++
    " Es konnte keine neue Constraint Klausel gefunden werden, " ++ "\n" ++
     " A kann also nicht widerlegt werden."
  | (map constraint (findLCK (resolv2 theorie))) /= []
    && length (findLCK (neue (resolv2 theorie)
                         (resolv2 (theorie++anfrage)))) == 1
    = "Die Constraint Klauseltheorie ist konsistent mit nichtleerer " ++
      " Kontrollmenge." ++ "\n" ++
      " Als Kontrollmenge wurde ausgewaehlt: " ++
      (litlialsstring (findKM (map constraint (findLCK (resolv2 theorie))))) ++
      "\n" ++ " Es konnte genau eine neue LCK bestimmt werden, diese ist: " ++
      "\n" ++
      (cklialsstring (connub (findLCK (neue (resolv2 theorie)
                           (resolv2 (theorie++anfrage))))))
   | (map constraint (findLCK (resolv2 theorie))) /= []
    && findLCK (neue (resolv2 theorie)
                         (resolv2 (theorie++anfrage))) /= []
    = "Die Constraint Klauseltheorie ist konsistent mit nichtleerer " ++
      " Kontrollmenge." ++ "\n" ++
```

```
      " Als Kontrollmenge wurde ausgewaehlt: " ++
      (litlialsstring (findKM (map constraint (findLCK (resolv2 theorie))))) ++
      "\n" ++ " Die gefundenen neuen leeren Constraint Klauseln sind:  " ++
      "\n" ++
      (cklialsstring (findLCK (neue (resolv2 theorie)
                                (resolv2 (theorie++anfrage)))))

-- -------------------------------------------------------KORESB END
-- ----------------------------------------------------------------SOS

sos::[CK]->[CK]->[CK]
sos alt []  = alt
sos alt neu
  | neue alt (alleresolv neu++alleresolvvonmit alt neu) == []
            = alt ++ (neue alt neu)
  | otherwise = sos (alt ++ (neue alt neu))
                  (neue (alt ++ (neue alt neu))
         (alleresolv neu++alleresolvvonmit alt neu))

-- ------------------------------------------------------------SOS END


sostest::[CK]->[CK]->Bool
sostest as [] = True
sostest as (b:bs) = (soslitest as (constraint b))&&(sostest as bs)

soslitest::[CK]->[[Lit Pred]]->Bool
soslitest as [] = True
soslitest as (b:bs) = (soslitesth as b)&&
                      (soslitest as bs)

soslitesth::[CK]->[Lit Pred]->Bool
soslitesth as [] = False
soslitesth as (b:bs)
```

```
  | elem ([],[]) (sos as [([neglit b],[])]) = True
  | otherwise                               = False

wclause::[CK]->[CK]
wclause [] = []
wclause ((a,b):as)
  | b == []    = (a,b):(wclause as)
  | otherwise  = wclause as

dclause::[CK]->[CK]
dclause [] = []
dclause ((a,b):as)
  | b /= []    = (a,b):(dclause as)
  | otherwise  = dclause as

-- Alle gefundenen LCK werden noch auf Vertraeglichkeit mit
-- den facts geprueft. Es gibt drei Argianten:
-- 1. Die LCK kann gestrichen werden.
-- 2. Die LCK kann als EC "enttarnt" werden.
-- 3. Die LCK bleibt als Bedingungsauswahl fuer den KM erhalten.

eccopt::[CK]->[CK]->[CK] -- alle LCK mit inkonsistentem constraint
-- fliegen raus.
-- Argument1 = facts
-- Argument2 = LCKs
eccopt as [] = []
eccopt as (b:bs)
  | eccok as b = b:(eccopt as bs)
  | otherwise  = eccopt as bs
eccok::[CK]->CK->Bool
eccok as ([],bs) = eccokh as bs
eccokh::[CK]->[[Lit Pred]]->Bool
eccokh as [] = True
eccokh as (b:bs) = (eccokhh as b)&&(eccokh as bs)
```

```
eccokhh::[CK]->[Lit Pred]->Bool
eccokhh as [] = False
eccokhh as (b:bs)
  | elem ([],[]) (sos as [([b],[])]) = eccokhh as bs
  | otherwise = True


-- -------------------------------------------------------XXXXXXX END
-- -----------------------------------------------------------FINDBED
-- Ziel: Bestimmung aller Bedingungen, unter denen die Aussage gilt.
-- ------------------------------------------------

findbed::[[Lit Pred ]]->KM->[[[(Arg,Arg)]]]
findbed [] cs     = []              -- [[[]]]
findbed (a:as) cs
  | passend a cs == [] = [[[(K "BedingungslosOK",K "")]]]
  | last (passend a cs) == [(K "TestNeuerConstraint3",K "")]
                       = findbed as cs
  | otherwise          = (passend a cs):(findbed as cs)

passend::[Lit Pred]->KM->[[(Arg,Arg)]]
passend [] cs = []
passend (a:as) cs
  | passendh cs a == [[(K "ConstraintLitOK",K "")]]
              = passend as cs
  | passendh cs a == [[(K "TestNeuerConstraint2",K "")]]
              = [[(K "TestNeuerConstraint3",K "")]]
  | otherwise = (passendh cs a)++(passend as cs)

passendh::KM->Lit Pred->[[(Arg,Arg)]]
passendh cs lit
  | [x|x<-cs,litname x == litname lit,vorz x /= vorz lit] == []
           = [[(K "ConstraintLitOK",K "")]]
  | ohneMarke v == []
```

```
          = [[(K "ConstraintLitOK",K "")]]
  | last (passendhh (varlivonlit lit)
                   (map varlivonlit
       [x|x<-cs,
litname x == litname lit,
        vorz x /= vorz lit])) ==
    [(K "TestNeuerConstraint",K "")]
          = [[(K "TestNeuerConstraint2",K "")]]
  | otherwise = ohneMarke v
 where v = passendhh (varlivonlit lit)
                    (map varlivonlit
     [x|x<-cs,
  litname x == litname lit,
  vorz x /= vorz lit])

ohneMarke::[[(Arg,Arg)]]->[[(Arg,Arg)]]
ohneMarke [] = []
ohneMarke (a:as)
  | last a == (K "KMLitOK",K "") = ohneMarke as
  | otherwise                    = a:(ohneMarke as)

passendhh::[Arg]->[[Arg]]->[[(Arg,Arg)]]
passendhh a [] = []
passendhh a (b:bs)
  | varcheck a b == [] = [[(K "TestNeuerConstraint",K "")]]
  | otherwise          = (varcheck a b):(passendhh a bs)


xx2::[Lit Pred]->KM->[[(Arg,Arg)]]
xx2 [] cs = []
xx2 (a:as) cs
  | xx (varlivonlit a) [] == [] = []
--   (map (varlivonlit (filter ((==(litname a))&&(/=(vorz a)) cs)))) == [] = []
```

```
xx::[Arg]->[[Arg]]->[[(Arg,Arg)]]
xx as [] = []
xx as (b:bs)
  | varcheck as b == [] = xx as bs
  | otherwise           = (varcheck as b):(xx as bs)

varcheck::[Arg]->[Arg]->[(Arg,Arg)]
varcheck [] [] = []
varcheck ((K a):as) ((K b):bs)
  | a/=b     = (K a,K b):(varcheck as bs)
varcheck ((X a b):as) ((K c):bs)
             = (X a b,K c):(varcheck as bs)
varcheck (_:as) (_:bs) = varcheck as bs
-- -------------------------------------------------------FINDBED END
-- ------------------------------------------------------OPTUNDPERMUT
optundpermut,optundpermuth ::[[[(Arg,Arg)]]]->[[(Arg,Arg)]]
optundpermut [] = []
optundpermut as
  | last as == [[(K "BedingungslosOK",K "")]]
                = [[(K "BedingungslosOK2",K "")]]
  | otherwise = optundpermuth as

optundpermuth [] = []
optundpermuth (a:as) =
   (foldl list [] a)++(optundpermuth as)

list::[[(Arg,Arg)]]->[(Arg,Arg)]->[[(Arg,Arg)]]
list [] bs      = spalte bs -- Initialspaltung
list (a:[]) bs  =  g a bs
list (a:as) bs  = (g a bs)++(list as bs)

spalte::[(Arg,Arg)]->[[(Arg,Arg)]]
spalte []       = []
spalte (a:as)   = [a]:(spalte as)
```

```
g::[(Arg,Arg)]->[(Arg,Arg)]->[[(Arg,Arg)]]
g as [] = []
g as bs = ((head bs):as):(g as (tail bs))
-- -------------------------------------------------OPTUNDPERMUT END
-- ----------------------------------------------------STRINGAUSGABE

eckialsbed::[CK]->Int->String
eckialsbed [] i = ""
eckialsbed (a:as) i = "Bedingung "++(show' i)++" verlangt "++
                            "dass:\\\n"++(cTeilalsString a)++"\\\n"++
 (eckialsbed as (i+1))

-- ------------------------------------------------
-- ------------------------------------------------
-- Die restlichen Stringausgabefunktionen.
-- ------------------------------------------------

kmalsstring::KM->String
kmalsstring a = litlialsstring a

tostring::[[(Arg,Arg)]]->String
tostring [] = "Das Negat kann nicht widerlegt werden."
tostring [[(K "BedingungslosOK2",K "")]] =
         "Das Negat ist bedingungslos widerlegt."
tostring (a:[]) = "Das Negat kann widerlegt werden, "++
                  "falls folgende Bedingung erfuellt ist: "++"\\\n"++
(tupelalsstring a)
tostring as = "Das Negat kann widerlegt werden, falls eine der "++
              "folgenden"++(show' (length as))++"Bedingungen\\\n"++
  " erfuellt ist: "++(tostringi 1 as)
tostringi::Int->[[(Arg,Arg)]]->String
tostringi i [] = ""
tostringi i (a:as) = "  Bedingung "++(show' i)++"ist"++
```

```
                      (tupelalsstring a)++(tostringi (i+1) as)
tupelalsstring::[(Arg,Arg)]->String
tupelalsstring [] = ""
tupelalsstring ((a,b):cs) = (varalsstring a)++" darf nicht sein "++
                            (varalsstring b)++"\\\n"++
 (tupelalsstring cs)

cTeilalsString::CK->String
cTeilalsString ([],[]) = ""
cTeilalsString ([],a) =
{- litalsstring (head a)++ -}
-- Zeile 913
                          " muss gelten.\\\n"++
                                cTeilalsString ([],tail a)

-- ------------------------------------------------STRINGAUSGABE END

-- ----------------------------------------------------------BASISF10

findLCK::[CK]->[CK]
findLCK [] = []
findLCK ((a,b):as)
  | a==[]     = (a,b):(findLCK as)
  | otherwise = findLCK as

-- ------------------------------------------------------------FINDKM
findKM::[[[Lit Pred]]]->KM
findKM []     = []
findKM (a:as) = findKMh (initkm a) as

initkm::[[Lit Pred]]->KM
initkm []     = []
initkm (a:as) = neg a
```

```
neg::[Lit Pred]->[Lit Pred]
neg [] = []
neg (a:as) = (neglit a):(neg as)

findKMh::KM->[[[Lit Pred]]]->KM
findKMh cs [] = cs
findKMh cs (a:as)
  | kmConok cs a  = findKMh (enlargekm cs a) as
  | otherwise      = findKMh cs as

kmConok::KM->[[Lit Pred]]->Bool
kmConok cs []     = True
kmConok cs (a:as) = kmConokh cs a &&
                    kmConok cs as
kmConokh::KM->[Lit Pred]->Bool
kmConokh cs []    = False
kmConokh cs (a:as)
  | elem (neglit a) cs = kmConokh cs as
  | otherwise          = True

enlargekm::KM->[[Lit Pred]]->KM
enlargekm cs (a:as) = nub (cs++neg a)
-- -------------------------------------------------------FINDKM END
-- -----------------------------------------------------------KMKONS
kmkons::KM->Bool
kmkons [] = True
kmkons (a:as)
  | elem (neglit a) as
     = False
  | otherwise
    = kmkons as
-- -------------------------------------------------------KMKONS END
-- -------------------------------------------------TEST KM,-DNF LCK
-- Dieser Test gilt nur fuer consistente KM
```

```
test::KM->[CK]->Bool
test cs [] = True
test cs ((a,b):as)
  | testh cs b = test cs as
  | otherwise = False
testh::[Lit Pred]->[[Lit Pred]]->Bool
testh cs [] = True
testh cs (a:as)
  | elem ([],[]) (sos [] ((a,[]):[(cs,[])])) = False
  | otherwise = testh cs as
tests::[Lit Pred]->[CK]->String
tests cs bs
  | test cs bs = " Die DNF ist bewiesen, es konnte eine mit dem "++
                 " KM konsistente LCK hergeleitet werden.\n"
  | otherwise  = " Der KM und die gefundenen LCK sind inkonsistent."
-- ----------------------------------------------TEST KM,-DNF LCK END
-- Vor:Konsistenter KM, Menge von LCK(DNF)
-- bedKM::KM->[[[Lit Pred]]]->[(Lit Pred,Lit Pred)]
-- 1. LCK ist nicht inkonsistent
-- 2. P(a), -P(b) [Lit Pred],[(String,String)]
```

Literaturverzeichnis

[Bibel,1982] W.Bibel: *Automated Theorem Proving*, Vieweg (1982)

[Bürckert,1991] H.-J.Bürckert: *A Resolution Principle for a logic with Restricted Quantifires*, Lecture Notes in Artificial Intelligence 568, Springer (1991)

[Chinlund et al.,1964] T.J.Chinlund, M.Davis, G.Hineman, D.McIlroy: *Theorem Proving by matching*, Bell Labotatories (1964)

[Church,1936] A.Church: *A Note on the Entscheidungsproblem*, J.Symbolic Logic, No.1 (1936) 40-41

[Davis,1954] ?

[Davis,1963] A.Davis: *Eliminating the Irrelevant from Mechanical Proofs*, Proc. Symp. Appl. Math. XV (1963) 15-30

[Frege,1882] G.Frege: *Über den Zweck der Begriffsschrift*, G.Ohms Verlagsbuchhandlung, Hildesheim (1964)

[Gentzen,1935] G.Gentzen: *Untersuchungen über das logische Schließen*, Mathematische Zeitschrift 39 (1933) 176-210

[Gödel,1930] K.Gödel *Die Vollständigkeit der Axiome des logischen Funktionenkalküls*, Monatshefte für Mathematik und Physik 37 (1930) 349-360

[Huet,1972] G.Huet *Constrained Resolution - A complete Method for higher Order Logic*, Ph.D. Thesis, (1972)

[Jones,1991] M.P.Jones, *Gofer, a functional programming language*, 1991

[Kowalski,1975] R.Kowalski *A proof Procedure Using Connection Graphs*, J.ACM Vol.22 No.4 (1975)

[Loveland,1968] D.W.Loveland: *A linear format for resolution*, Proc, France,1968 (Springer Vlg.NY) 1970 147-162

[Minsky,1975] M.Minsky: A framework for representing knowledge, P.Winston (Ed.), The Psychologie of Computer Vision, Mc Graw-Hill,NY (1975)

[Panitz,1993] S.E.Panitz: *Default Reasoning with a Constraint Resolution Principle*, Lecture Notes in Artificial Intelligence 698, Springer (1993) 265-276

[Peano,1889] G.Peano: *Arithmetices principia, nova methodo exposita*, Turin (1889)

[Poole,1988] D.Poole: *A Logical Framework for Default Reasoning*, Artificial Intelligence 36 (1988) 27-47

[Poole et al.,1987] D.Poole, R.Goebel, R.Aleliunas: *Theorist: A Logical Reasoning System for Defaults and Diagnosis* (1987)

[Presburger,1930] M.Presburger *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*, Compute-Rendus du Congres des Mathematiciens ds pays, Warsaw (1930) 92-101

[Raphael,1971] B.Raphäl, *The frame problem in problem-solving systems*, N.V.Findler and B.Meltzer (Eds.), Artificial Intelligenz and Heuristic Programming, Edingburgh University Press (1971)

[Reiter,1978] R.Reiter, *On closed world data bases*, H.Gaillaire and J.Minker (Eds.), Logic and Data Bases, Plenum Press NY (1987) 55-76

[Reiter,1980] R.Reiter *A logic for default reasoning*, Artificial Intelligence 13 (1980) 81-132

[Robinson,1965a] J.A.Robinson: *A Machine oriented Logic Based on the Resolution Principle*, J.ACM 12 (1965)

[Robinson,1965b] J.A.Robinson: *Automated Deduction with Hyper-Resolution*, Intern.J.of Comp.Math. 1 (1965b), 227-234

[Smullyan,1968] R.M.Smullyan: *First Order Logic*, Springer Verlag Berlin (1968)

[Stickel,1985] M.E.Stickel: *Automated Deduction by Theory Resolution*, J.of Automated Reasoning 1(4), (1985) 333-357

[Tarski,1953] A.Tarski: *Der Wahrheitsbegriff in den formalisierten Sprachen*, Studia Philosophia I, (1953)

[Woods,1975] W.A.Woods: *Representation and Understanding: Studies in Cognitive Science*, D.G.Bobrow and A.M.Collins (Eds.), NY Academic Press (1975) 35-82