Diplomarbeit

Semantik und Korrektheit von Prolog-Programmen im Naproche-Projekt

Angefertigt am
Mathematischen Institut

Vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

im November 2009

von

Dörthe Arndt

aus
Osnabrück

Danksagung

An dieser Stelle möchte ich denjenigen danken, die mir bei der Erstellung dieser Diplomarbeit geholfen haben. Ohne sie wäre diese Arbeit nicht möglich gewesen.
Mein besonderer Dank gilt Prof. Dr. Peter Koepke für die Betreuung dieser Arbeit. Er stand stets für jede Art von Fragen zur Verfügung und nahm sich immer Zeit für mich. Nicht zuletzt durch seine freundliche und engagierte Betreuung hat mir diese Diplomarbeit viel Freude bereitet.
Außerdem möchte ich dem gesamten Naproche-Team, insbesondere Daniel Kühlwein, für die angenehme Zusammenarbeit danken.
Weiterhin danke ich Nicolas Schulz herzlich für das Korrekturlesen meiner Diplomarbeit.
Selbstverständlich gilt mein Dank auch meiner Familie und meinen Freunden, die mich während des gesamten Studiums und vor allem beim Schreiben der Diplomarbeit in jeder Hinsicht unterstützt haben.

Inhaltsverzeichnis

1 Einleitung

Einer der wichtigsten Bestandteile der Mathematik ist der mathematische Beweis. Doch was ist ein Beweis? Wie muss er formuliert sein, um als „korrekt“ zu gelten?
Beweise legen in bestimmter Form Argumente dar, die den Leser von der Aussage eines Satzes überzeugen sollen. Ob sie dieses Ziel tatsächlich erreichen, hängt von verschiedenen Faktoren ab. Neben den verwendeten Argumenten spielt auch deren Darstellung eine Rolle.
In den meisten mathematischen Texten, wie man sie zum Beispiel in Fachzeitschriften findet, sind Beweise in einer Mischung natürlichsprachlicher Ausdrücke und mathematischer Zeichen formuliert. Hierdurch ergibt sich zumindest theoretisch die Möglichkeit, die Sprache für gleiche Argumentationen beliebig zu variieren. Zwar gibt es in der Mathematik gewisse Formulierungen und Argumentationen, deren Gebrauch in Beweisen üblich ist, ein fester Standard existiert hierfür jedoch nicht. So ist es schwierig, den Begriff des natürlichsprachlichen Beweises genau zu erfassen.
Anders verhält es sich für eine weitere Form des Beweises, den formalen Beweis. Dieser ist in einer klar festgelegten mathematischen Formelsprache geschrieben. In einem Kalkül sind alle auf Formeln anwendbaren Regeln festgelegt. Ein Beweis besteht aus der systematischen Anwendung dieser Regeln auf die Voraussetzungen eines Satzes, so dass dessen Aussage folgt. Auf diese Weise ist der Begriff des Beweises klar definiert, seine Korrektheit lässt sich leicht überprüfen.
Mit der Entwicklung des formalen Beweises kam Anfang des zwanzigsten Jahrhunderts die Frage auf, ob und wie sich die Mathematik formalisieren lässt. In ihrem Werk *Principia Mathematica* [WR63] führen *Bertrand Russell* und *Alfred North Whitehead* formale Beweise für einige grundlegende Sätze der Mathematik. Ausgehend von dieser Arbeit nimmt man an, dass sich für jeden bewiesenen Satz auch ein formaler Beweis finden lässt. Leider wird an diesem Text aber auch der dazu nötige Aufwand deutlich. Die genaue Dokumentation jedes einzelnen Beweisschrittes erweist sich als mühselig. Die Sprache formaler Beweise ist für den Menschen schwer verständlich.

Gegründet von *Peter Koepke* (Mathematik, Universität Bonn) und *Bernhard Schröder* (Linguistik, Universität Duisburg-Essen), beschäftigt sich das Projekt NAPROCHE – NAtural language PROof CHEcking – mit der Sprache der Mathematik. Ein interessanter Aspekt dieser Thematik ist der Zusammenhang zwischen den üblichen vorwiegend natürlichsprachlichen und rein formalen Beweisen. Gelingt es, natürlichsprachliche Beweise eindeutig in rein formale zu übersetzen, ist der Begriff der Korrektheit auch für die übliche Form des Beweises greifbar. Um dieses Ziel umzusetzen, wurde das Naproche-System entwickelt. Hierbei handelt es sich um ein Computerprogramm, das in einer

kontrollierten Eingabesprache verfasste Beweise in formale überführt und diese dann auf Korrektheit überprüft. Das Eingabeformat des Systems ist dabei an der tatsächlichen mathematischen Sprache orientiert, so dass die von dem System akzeptierten Beweise auch für den menschlichen Leser nachvollziehbar sind. So kann dieses Programm zum Beispiel eine Hilfe beim Verfassen mathematischer Texte sein.
Möchte man jedoch als Autor oder Leser einem Beweisprüfungssystem wirklich vertrauen, muss die Korrektheit der ihm zu Grunde liegenden Theorie und ihrer konkreten Umsetzung garantiert sein. Findet das System einen Fehler, so soll dieser im Beweis, nicht aber im System selbst liegen. Umgekehrt sollen nur korrekte Beweise auch als solche erkannt werden.

Einen Beitrag zur Behandlung der Theorie leisten die Arbeiten von *Kolev* [Kol08], *Kühlwein* [Küh09] und *Cramer* [Cra09]. Dort wird die Arbeitsweise wichtiger Teile des Naproche-Systems erklärt und überprüft.
In diesem Diplomprojekt geht es um die Umsetzung der Theorie in Form des Programmcodes. Wie lässt sich garantieren, dass dieser genau das Vorgesehene tut? Welche Fehler können auftreten? Wie kann man diese vermeiden?
Ein praktischer Teil des Projekts bestand aus der Entwicklung und Implementierung des Logikmoduls[1] der Version Naproche 0.2 des Naproche-Systems[2]. Dieses Modul ist, wie ein Großteil des Naproche-Systems, in der Programmiersprache Prolog geschrieben.
Die vorliegende Diplomarbeit zeigt Methoden auf, Korrektheit und Semantik von Prologprogrammen zu untersuchen und behandelt die Fragestellung, ob und wie sich die gefundenen Techniken auf den Quellcode des Naproche-Systems übertragen lassen.
In Kapitel 2 werden hierzu die Grundlagen und Definitionen der Logik erster Stufe behandelt, die in dieser Arbeit verwendet werden. Kapitel 3 bildet dann eine Einführung in die logische Programmierung. Diese ist die theoretische Grundlage der Programmiersprache Prolog. Anschließend wird in Kapitel 4 eine Teilmenge von Prolog, reines Prolog, eingeführt. Wir behandeln, wie sich Prolog von logischer Programmierung unterscheidet und welche Konsequenzen dies für die Korrektheit von Programmen haben kann. Kapitel 5 zeigt dann Möglichkeiten auf, die angesprochenen Probleme auszuschließen und die Bedeutung von Programmen genau zu erfassen. In Kapitel 6 wenden wir die eingeführten Techniken exemplarisch auf die vereinfachte Version eines Teilprogrammes aus dem Naproche-System, der Formelgrammatik, an. Kapitel 7 gibt einen Überblick über die wichtigsten im Naproche-System verwendeten Prolog-Elemente, die über reines Prolog hinausgehen, und erörtert die Möglichkeit, die in Kapitel 5 vorgestellten Techniken auch auf Programmteile, die diese Elemente enthalten, zu übertragen. In Kapitel 8 fassen wir die Ergebnisse dieser Diplomarbeit nochmals zusammen.

[1] Das Logikmodul wurde zwischen April und Juli 2008 von *Daniel Kühlwein* und *Dörthe Arndt* mit Unterstützung der Praktikantinnen *Shruti Gupta* und *Bhoomija Ranjan* erstellt. Es regelt die Beweisprüfung für Beweise in einem logischen Zwischenformat, der Beweisrepräsentationsstruktur. Informationen hierzu findet man bei *Kühlwein* [Küh09].
[2] Auf diese Version des Systems beziehen wir uns im Folgenden.

2 Grundlagen

Dieses Kapitel bildet eine kurze Einführung in die Logik erster Stufe. Hierbei beschränken wir uns auf die Aspekte, die für die weiteren Kapitel dieser Arbeit von Bedeutung sind. Detailliertere Darstellungen findet man in den meisten Einführungswerken in die Logik (z.B. [Rau08]). Das vorliegende Kapitel ist im Wesentlichen an den Büchern von *Ebbinghaus et al.* [EFT07] und *Nilsson und Maluszynski* [NM95] orientiert.

2.1 Logische Formeln

Wir beginnen mit der Syntax einer Sprache erster Stufe. Hierzu definieren wir zunächst die verwendeten Zeichen, das Alphabet. Allgemein gilt:

Definition 2.1 (Alphabet und Wort über einem Alphabet) *Unter einem* Alphabet $\mathcal{A}$ *versteht man eine nicht-leere Menge von Zeichen. Eine endliche Folge von Elementen des Alphabets* $\mathcal{A}$ *nennt man* Wort *über* $\mathcal{A}$*. Mit* $\mathcal{A}^*$ *bezeichnet man die Menge aller Wörter über* $\mathcal{A}$*.*

Für eine Sprache erster Stufe sieht das Alphabet folgendermaßen aus:

Definition 2.2 (Alphabet einer Sprache erster Stufe) *Das* Alphabet $\mathcal{A}$ *einer Sprache erster Stufe besteht aus den folgenden disjunkten Mengen von Symbolen:*

(i) *Variablen:* $x_0, x_1, x_2, ...$

(ii) *Konnektionssymbole: Negation* $\neg$*, Konjunktion* $\wedge$*, Disjunktion* $\vee$*, Implikation* $\rightarrow$*, Äquivalenz* $\leftrightarrow$

(iii) *Quantorensymbole: Allquantor* $\forall$*, Existenzquantor* $\exists$

(iv) *Hilfssymbole: Klammern),(, Komma ,*

(v) *Relationssymbole: für jedes* $n \geq 0$ *eine (eventuell leere) Menge von n-stelligen Relationssymbolen*

(vi) *Funktionssymbole: für jedes* $n \geq 1$ *eine (eventuell leere) Menge von n-stelligen Funktionssymbolen*

(vii) *Konstantensymbole: eine (eventuell leere) Menge von Konstanten*

Die Punkte (i)-(iv) sind für alle Sprachen erster Stufe gleich. Die verschiedenen Alphabete von Sprachen erster Stufe unterscheiden sich durch die in (v)-(vii) genannten Symbolmengen. Die folgenden Definitionen sollen für ein beliebiges, aber fest gewähltes Alphabet $\mathcal{A}$ gelten.

Definition 2.3 (Terme) *Die Menge $\mathcal{T}$ der* Terme *besteht aus der kleinsten Teilmenge von $\mathcal{A}^*$, für die das Folgende gilt:*

(T1) Jede Variable aus $\mathcal{A}$ ist in $\mathcal{T}$.

(T2) Jede Konstante aus $\mathcal{A}$ ist in $\mathcal{T}$.

(T3) Ist f ein n-stelliges Funktionssymbol aus $\mathcal{A}$ und $t_1, ..., t_n \in \mathcal{T}$, so ist $f(t_1, ..., t_n) \in \mathcal{T}$.

Definition 2.4 (Formeln) *Sei $\mathcal{T}$ die Menge der Terme über dem Alphabet $\mathcal{A}$. Die Menge $\mathcal{F}$ der* Formeln *ist die kleinste Teilmenge von $\mathcal{A}^*$, so dass gilt:*

(i) Wenn $r \in \mathcal{A}$ ein n-stelliges Relationssymbol ist und $t_1, ..., t_n \in \mathcal{T}$, so gilt $r(t_1, ..., t_n) \in \mathcal{F}$.

(ii) Wenn ϕ und $\psi \in \mathcal{F}$, so gilt auch $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi) \in \mathcal{F}$

(iii) Wenn $\phi \in \mathcal{F}$ und x eine Variable in $\mathcal{A}$ ist, so gilt: $\forall x\phi$ und $\exists x\phi \in \mathcal{F}$.

Eine Formel der Form $r(t_1, \ldots, t_n)$, bei der r ein n-stelliges Prädikatensymbol aus $\mathcal{A}$ ist und $t_1, \ldots, t_n \in \mathcal{T}$ gilt, nennt man atomare Formel *oder* Atom.

Die Menge aller Formeln, die mit Hilfe des Alphabets $\mathcal{A}$ gebildet werden können, nennen wir die *zu dem Alphabet $\mathcal{A}$ gehörige Sprache erster Stufe.*

Die innerhalb einer Formel vorkommenden Variablen werden unterschieden in freie und gebundene Variablen. Hierzu soll $\mathrm{var}(t)$ die Menge der in dem Term t vorkommenden Variablen bezeichnen:

Definition 2.5 *Für $t \in \mathcal{T}$ definiere* $\mathrm{var}(t) \subseteq \{x_n | n \in \mathbb{N}\}$ *rekursiv durch:*

- $\mathrm{var}(x) := \{x\}$
- $\mathrm{var}(c) := \emptyset$
- $\mathrm{var}(f(t_1, ..., t_n)) := \mathrm{var}(t_1) \cup ... \cup \mathrm{var}(t_n)$

Damit kann man nun Folgendes definieren:

Definition 2.6 (Freie Variablen) *Sei ϕ eine Formel. Die Menge* $\mathrm{frei}(\phi)$ *der freien Variablen aus ϕ ist definiert durch:*

- $\text{frei}(r(t_1, ..., t_n)) = \text{var}(t_1) \cup ... \cup \text{var}(t_n)$
- $\text{frei}(\neg\phi) = \text{frei}(\phi)$
- $\text{frei}(\phi \vee \psi) = \text{frei}(\phi) \cup \text{frei}(\psi)$
- $\text{frei}(\phi \wedge \psi) = \text{frei}(\phi) \cup \text{frei}(\psi)$
- $\text{frei}(\phi \rightarrow \psi) = \text{frei}(\phi) \cup \text{frei}(\psi)$
- $\text{frei}(\phi \leftrightarrow \psi) = \text{frei}(\phi) \cup \text{frei}(\psi)$
- $\text{frei}(\exists x\phi) = \text{frei}(\phi) \setminus \{x\}$
- $\text{frei}(\forall x\phi) = \text{frei}(\phi) \setminus \{x\}$

Enthält eine Formel keine freien Variablen, so nennt man diese abgeschlossen oder *Grundformel*. Entsprechend nennt man einen Term, der keine freien Variablen enthält, *Grundterm*. Variablen, die nicht frei in einer Formel vorkommen, heißen *gebunden*.

Eine nicht abgeschlossene Formel kann man durch entsprechenden Gebrauch eines Quantors abschließen:

Definition 2.7 (Universeller und Existentieller Abschluss) *Seien $x_1, ..., x_n$ alle Variablen, die frei in einer Formel ϕ vorkommen. Die abgeschlossene Formel der Form $\forall x_1 \forall x_2 \forall x_n \phi$ heißt* universeller Abschluss *von ϕ, geschrieben $\forall\phi$. Die abgeschlossene Formel $\exists x_1 \exists x_2 \exists x_n \phi$ heißt* existentieller Abschluss *von ϕ, geschrieben $\exists\phi$.*

2.2 Semantik von Formeln

Dieser Abschnitt widmet sich der Bedeutung einer Formel erster Stufe. Um diese klar fassen zu können, wird hier zunächst der Begriff der Struktur eingeführt:

Definition 2.8 (Struktur) *Eine Struktur über einem Alphabet $\mathcal{A}$ ist ein Paar $\mathfrak{A} = (\mathcal{D}, \mathfrak{a})$ mit den folgenden Eigenschaften:*

(a) $\mathcal{D}$ ist eine nicht leere Menge, die sogenannte Trägermenge *von $\mathfrak{A}$.*

(b) $\mathfrak{a}$ ist eine auf $\mathcal{A}$ definierte Abbildung, für die gilt:

(1) Für jede Konstante $c \in \mathcal{A}$ ist $\mathfrak{a}(c) \in \mathcal{D}$.

(2) Für jedes n-stellige Funktionssymbol $f \in \mathcal{A}$ ist $\mathfrak{a}(f) : \mathcal{D}^n \rightarrow \mathcal{D}$ eine Funktion.

(3) Für jedes n-stellige Relationssymbol $r \in \mathcal{A}$ ist $\mathfrak{a}(r) \subseteq \mathcal{D}^n$ eine Relation.

Wir schreiben $\mathfrak{a}(c)$ auch als $c_{\mathfrak{A}}$, $\mathfrak{a}(f)$ als $f_{\mathfrak{A}}$ und $\mathfrak{a}(r)$ als $r_{\mathfrak{A}}$.

Um auch den Variablen einer Formel einen Wert zuweisen zu können, benötigt man außerdem den Begriff der Belegung.

Definition 2.9 (Belegung) *Eine Belegung β in einer Struktur $\mathfrak{A}$ ist eine Abbildung von der Menge der Variablen in den Definitionsbereich $\mathcal{D}$ der Struktur. Mit der Notation $\beta\{x/d\}$ bezeichnen wir die Belegung, die an allen Stellen, außer $x \mapsto d$, identisch zu β ist.*

Belegungen werden im Folgenden auch als Menge von Paaren der Form x/d dargestellt. Hierdurch soll ausgedrückt werden, dass die Belegung die Variable x auf den Wert $d \in \mathcal{D}$ abbildet. Wir schreiben: $\beta = \{x_1/d_1, ..., x_n/d_n\}$.

Zusammen bilden Belegung und Struktur eine Interpretation:

Definition 2.10 (Interpretation) *Eine Interpretation $\mathfrak{I}$ ist ein Paar $(\mathfrak{A}, \beta)$, bestehend aus einer Struktur $\mathfrak{A}$ und einer Belegung β.*

An dieser Stelle können wir nun die Bedeutung von Termen und Formeln festlegen:

Definition 2.11 (Bedeutung von Termen) *Sei $\mathfrak{I} = (\mathfrak{A}, \beta)$ eine Interpretation und t ein Term. Dann ist die* Bedeutung $\mathfrak{I}(t)$ *von t folgendermaßen definiert:*

- *wenn t eine Konstante c ist, so gilt $\mathfrak{I}(t) := c_{\mathfrak{A}}$*
- *wenn t eine Variable x ist, so gilt $\mathfrak{I}(t) := \beta(x)$*
- *wenn t von der Form $f(t_1, ..., t_n)$ ist, so gilt $\mathfrak{I}(t) := f_{\mathfrak{A}}(\mathfrak{I}(t_1), ..., \mathfrak{I}(t_n))$*

Definition 2.12 (Bedeutung von Formeln) *Seien $\mathfrak{I} = (\mathfrak{A}, \beta)$ eine Interpretation und ϕ und ψ Formeln. Dann gilt:*

- $\mathfrak{I} \models r(t_1, ..., t_n)$ *gdw.* $(\beta(t_1), ..., \beta(t_n)) \in r_{\mathfrak{A}}$
- $\mathfrak{I} \models \neg\phi$ *gdw.* $\mathfrak{I} \not\models \phi$
- $\mathfrak{I} \models (\phi \wedge \psi)$ *gdw.* $\mathfrak{I} \models \phi$ *und* $\mathfrak{I} \models \psi$
- $\mathfrak{I} \models (\phi \vee \psi)$ *gdw.* $\mathfrak{I} \models \phi$ *oder* $\mathfrak{I} \models \psi$ *oder beides*
- $\mathfrak{I} \models (\phi \rightarrow \psi)$ *gdw.* $\mathfrak{I} \models \psi$ *wenn* $\mathfrak{I} \models \phi$
- $\mathfrak{I} \models (\phi \leftrightarrow \psi)$ *gdw.* $\mathfrak{I} \models \phi \rightarrow \psi$ *und* $\mathfrak{I} \models \psi \rightarrow \phi$
- $\mathfrak{I} \models \forall x\phi$ *gdw.* $(\mathfrak{A}, \beta\{x/t\}) \models \phi$ *für alle* $t \in \mathcal{D}$
- $\mathfrak{I} \models \exists x\phi$ *gdw.* $(\mathfrak{A}, \beta\{x/t\}) \models \phi$ *für ein* $t \in \mathcal{D}$

Gilt $\mathfrak{I} \models \phi$ für eine Formel ϕ, so sagen wir ϕ ist wahr *in $\mathfrak{I}$.*

Damit können wir den Begriff des Modells einführen:

Definition 2.13 (Modell) *Sei Φ eine Menge von Formeln. Eine Interpretation $\mathfrak{I}$ heißt Modell von Φ gdw. jede Formel von Φ ist wahr in $\mathfrak{I}$.*

Eng an die Modellbeziehung geknüpft ist die Folgerungsbeziehung:

Definition 2.14 (Logische Konsequenz) *Sei Φ eine Menge von abgeschlossenen Formeln. Eine abgeschlossene Formel ϕ heißt logische Konsequenz von Φ (geschrieben: $\Phi \models \phi$) gdw. ϕ ist wahr in jedem Modell von Φ.*

Diese Definition ermöglicht uns, die Begriffe *Allgemeingültigkeit*, *Erfüllbarkeit* und *logische Äquivalenz* einzuführen.

Definition 2.15 (Allgemeingültigkeit) *Eine logische Formel ϕ heißt* allgemeingültig *(kurz: $\models \phi$) gdw. $\emptyset \models \phi$.*

Definition 2.16 (Erfüllbarkeit) *Eine Formel ϕ heißt genau dann* erfüllbar*, wenn es eine Interpretation gibt, die Modell von ϕ ist. Eine Formelmenge Φ heißt genau dann erfüllbar, wenn es eine Interpretation gibt, die Modell für alle Formeln aus Φ ist.*

Diesbezüglich gilt folgendes Lemma:

Lemma 2.17 *Sei Φ eine Menge von abgeschlossenen Formeln und ϕ eine abgeschlossene Formel. Dann gilt $\Phi \models \phi$ gdw. $\Phi \cup \{\neg\phi\}$ ist unerfüllbar.*

Beweis $\Phi \models \phi$
gdw. jede Interpretation, die Modell von Φ ist, Modell von ϕ ist
gdw. es gibt keine Interpretation, die Modell von Φ ist und nicht Modell von ϕ
gdw. es gibt keine Interpretation, die Modell von $\Phi \cup \{\neg\phi\}$ ist
gdw. $\Phi \cup \{\neg\phi\}$ ist unerfüllbar □

Definition 2.18 (Logische Äquivalenz) *Zwei Formeln ϕ und ψ heißen logisch äquivalent ($\phi \equiv \psi$) gdw. $\phi \models \psi$ und $\psi \models \phi$ gilt.*

2.3 Substitutionen

Besonders wichtig für die folgenden Kapitel ist der Begriff der Substitution. Hierzu sei weiterhin ein Alphabet erster Stufe gegeben. Formal handelt es sich bei der Substitution um eine Abbildung von den Variablen einer Sprache auf deren Terme. Wir schreiben dies als Menge von Termpaaren $\{x_1/t_1, ..., x_n/t_n\}$, um deutlich zu machen, dass die Variable x_i auf den Term t_i abgebildet wird, und definieren nun folgendermaßen:

Definition 2.19 (Substitution) *Eine* Substitution *ist eine endliche Menge von Termpaaren* $\{x_1/t_1, ..., x_n/t_n\}$*, in der jedes* t_i *ein Term und jedes* x_i *eine Variable ist, so dass* $x_i \neq t_i$ *und* $x_i \neq x_j$ *für* $i \neq j$*. Die leere Substitution bezeichnen wir mit* ϵ*.*

Die Substitution kann man nun auf Terme oder Formeln anwenden:

Definition 2.20 (Anwendung) *Sei* θ *eine Substitution* $\{x_1/t_1, ..., x_n/t_n\}$ *und* E *ein Term oder eine quantorenfreie Formel. Die* Anwendung $E\theta$ *von* θ *auf* E *erhält man, indem man simultan jede in* E *vorkommende Variable* x_i *durch* t_i $(1 \leq i \leq n)$ *ersetzt. Man nennt* $E\theta$ *eine* Instanz *von* E*.*

Substitutionen können auch für quantifizierte Formeln definiert werden. Hierbei sind gebundene Variablen gesondert zu betrachten. Da dieser Fall in den folgenden Kapiteln nicht auftritt, verzichten wir an dieser Stelle auf seine Definition und verweisen auf *Ebbinghaus et al.* [EFT07, S.55f].
Substitutionen kann man miteinander kombinieren:

Definition 2.21 (Komposition) *Seien* θ *und* σ *Substitutionen:*

$$\theta := \{x_1/s_1, ..., x_m/s_m\}$$
$$\sigma := \{y_1/t_1, ..., y_n/t_n\}$$

Die Komposition $\theta\sigma$ *von* θ *und* σ *erhält man, indem man zunächst aus der Menge*

$$\{x_1/s_1\sigma, ..., x_m/s_m\sigma\}$$

alle $x_i/s_i\sigma$ *mit* $x_i = s_i\sigma$ $(1 \leq i \leq n)$ *und aus der Menge*

$$\{y_1/t_1, ..., y_n/t_n\}$$

alle y_j/t_j *mit* $y_j \in \{x_1, ..., x_n\}$ $(1 \leq j \leq n)$ *entfernt und die sich ergebenden Mengen vereinigt.*

Aus dieser Definition ergeben sich verschiedene Eigenschaften für die Kombination von Substitutionen:

Satz 2.22 (Eigenschaften von Substitutionen) *Seien* θ*,* σ *und* γ *Substitutionen und* E *ein Term oder eine quantorenfreie Formel, dann gilt:*

(a) $E(\theta\sigma) = (E\theta)\sigma$
(b) $\theta(\sigma\gamma) = (\theta\sigma)\gamma$

(c) $\epsilon\theta = \theta\epsilon = \theta$

Einen Beweis dieses Satzes gibt z.B. *Lloyd* in [Llo87, S.21].

Damit ist die Kombination von Substitutionen zwar assoziativ aber nicht kommutativ, wie das folgende Beispiel für $x \neq y$ zeigt:

$$\{x/f(y)\}\{y/c\} = \{x/f(c), y/c\} \neq \{y/c\}\{x/f(y)\} = \{y/c, x/f(y)\}$$

Wir führen zwei besondere Arten der Substitution ein:

Definition 2.23 (Idempotente Substitution) *Eine Substitution heißt* idempotent *gdw.* $\theta\theta = \theta$.

Jede Substitution, bei der die Menge der Variablen, die in ihrem Wertebereich vorkommen, disjunkt zu ihrem Definitionsbereich ist, ist idempotent. Die umgekehrte Richtung folgt aus Definition 2.21. Gilt für eine Substitution θ $\theta\theta = \theta$, so ist für alle $x_i/t_i \in \theta$ $x_i/t_i\theta = x_i/t_i$. Es gilt also:

Lemma 2.24 *Eine Substitution* θ *ist genau dann idempotent, wenn gilt:*

$$\mathrm{Dom}(\theta) \cap \mathrm{var}(\mathrm{Ran}(\theta)) = \emptyset$$

Dieses Lemma erleichtert den Nachweis von Idempotenz.

Definition 2.25 (Umbenennung) *Sei E ein Term oder eine quantorenfreie Formel. Eine* Umbenennung *für E ist eine Substitution zwischen Variablen* $\{x_1/y_1, \ldots, x_n/y_n\}$*, so dass gilt:*

(i) $\{x_1, \ldots, x_n\} \subset \mathrm{var}(E)$

(ii) $y_i \neq y_j$ *für* $i, j \in [1, n]$ *und* $i \neq j$

(iii) $(\mathrm{var}(E) \setminus \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \emptyset$

Ist E eine quantorenfreie Formel oder ein Term und ω eine Umbenennung, so sind E und $E\omega$ bis auf Variablen gleich. In diesem Fall nennt man $E\omega$ eine *Variante* von E. Da die Umkehrabbildung jeder Umbennenung ebenfalls eine Umbenennung ist, ist dann wegen $E\omega\omega^{-1} = E$ auch E eine Variante von $E\omega$.

3 Logische Programmierung

Logische Programmierung bildet die Grundlage der von Naproche verwendeten Programmiersprache Prolog. Um Aussagen über die Korrektheit des Programms Naproche treffen zu können, ist es deshalb notwendig, sich mit den Grundprinzipien der logischen Programmierung genauer zu beschäftigen.
Aus der Theorie des automatischen Beweisens mittels des 1965 von *J. A. Robinson* in [Rob65] eingeführten Resolutionspinzips entstand die Idee, Logik nicht nur für Beweise, sondern auch zur Programmierung zu benutzen. Dies geht unter anderem auf *R. Kowalski* (eingeführt in [Kow74]) zurück. Dieser betrachtet Algorithmen aus zwei Perspektiven, der *deklarativen*, die die Bedeutung eines Problems behandelt, und der *prozeduralen*, die den Lösungsweg für das behandelte Problem aufzeigt (vgl. [Kow79]). Im erst genannten Artikel (d.h. in [Kow74]) zeigt Kowalski besonders die Möglichkeit auf, Logik erster Stufe prozedural zu betrachten und eröffnet so einen Weg maschineller Verarbeitung von logischen Formeln. Das Ideal der logischen Programmierung ist es, diese prozedurale Seite der Programmierung dem Computer zu überlassen, so dass die logische Formulierung eines Problems schon einen Programmcode bildet und damit auch seine Lösung liefert. Besonders aus Effizienzgründen wird dies in der Praxis, der Programmiersprache Prolog, nicht immer erreicht.
Dieses Kapitel bildet eine kurze Einführung in den Bereich der logischen Programmierung. Besondere Aufmerksamkeit wird hierbei der Korrektheit und Vollständigkeit der zu Grunde liegenden Inferenzregel, die damit auch die Korrektheit des Programms Naproche beeinflussen, gewidmet. Wir richten uns in diesem Kapitel vor allem nach den Büchern von *Lloyd* [Llo87, Llo84] und *Nilsson und Maluszynski* [NM95].

3.1 Definite Programme

Logische Programmierung beschäftigt sich mit einer Teilmenge der Formeln erster Stufe, den sogenannten *Klauseln*. Hierbei handelt es sich um den universellen Abschluss von Disjunktionen atomarer oder negierter atomarer Formeln. Dass diese Formelmenge sehr ausdrucksstark ist, zeigt beispielsweise *Schöning* in [Sch00, S.66f]. Dort wird ein Verfahren vorgestellt, zu jeder Formel erster Stufe eine Formel der oben beschriebenen Form zu finden, die genau dann erfüllbar ist, wenn dies für die Ausgangsformel gilt. Als vertiefende Einführung in diesen Bereich sei hier außerdem [Doe94] empfohlen.
Da im Programmcode von Naproche, bis auf wenige Ausnahmen, nur eine bestimmte Art der Klausel, die *definite Klausel*, vorkommt, beschränken wir unsere Betrachtungen auf

die Behandlung dieser. Einen allgemeineren Einblick in die Verarbeitung von Klauseln durch logische Programmierung gibt *J. W. Lloyd* in [Llo87].

Um den Begriff der Klausel zu präzisieren, beginnen wir mit der Definition von Literalen. Diese bilden die Grundbausteine für Klauseln.

Definition 3.1 (Literale) *Atomare Formeln und Negationen von atomaren Formeln nennt man (*positive *oder entsprechend* negative*)* Literale*.*

Damit kann man nun definieren:

Definition 3.2 (Klausel) *Eine* Klausel *ist eine Formel der Form*

$$\forall x_1 \ldots \forall x_s (L_1 \vee \cdots \vee L_n)$$

wobei jedes L_i ein Literal ist und $x_1, \ldots, x_s$ alle in $L_1, \ldots, L_n$ vorkommenden Variablen sind.

Wir notieren Klauseln im Folgenden in ihrer in der logischen Programmierung üblichen Schreibweise. Eine Klausel der Form

$$\forall x_1 \ldots \forall x_s (A_1 \vee \cdots \vee A_k \vee (\neg B_1) \vee \cdots \vee (\neg B_n))$$

in der $A_1, \ldots, A_k, B_1, \ldots, B_n$ Atome und $x_1, \ldots, x_s$ die in den Atomen vorkommenden Variablen sind, schreiben wir als

$$A_1, \ldots, A_k \leftarrow B_1, \ldots, B_n$$

Hierbei entspricht $\leftarrow$ dem umgekehrten Implikationspfeil der Logik erster Stufe, d.h. für beliebige Formeln ϕ und ψ ist $\phi \leftarrow \psi$ gleichbedeutend mit $\psi \rightarrow \phi$. Die Kommata auf der linken Seite des Pfeils können als die logische Verknüpfung $\vee$, die rechts des Pfeils als $\wedge$ verstanden werden. Alle Formeln werden als universell abgeschlossen angenommen. Diese Schreibweise liegt in der logischen Äquivalenz der Formeln

$$\forall x_1 \ldots x_s (A_1 \vee \cdots \vee A_k \vee (\neg B_1) \vee \cdots \vee (\neg B_n))$$

und

$$\forall x_1 \ldots x_s ((A_1 \vee \cdots \vee A_k) \leftarrow (B_1 \wedge \cdots \wedge B_n))$$

begründet.

Definite Klauseln sind Klauseln, die genau ein positives Literal enthalten:

Definition 3.3 (Definite Klausel) *Eine* definite Klausel *ist eine Klausel der Form*

$$A_0 \leftarrow A_1 \ldots A_n$$

wobei $n \in \mathbb{N}$ gilt und A_i für alle $0 \leq i \leq n$ ein Atom ist. A_0 nennt man den Kopf*, $A_1, \ldots, A_n$ den* Körper *der Klausel.*

Im Fall $n > 0$ nennt man die definite Klausel auch *Regel*. Gilt $n = 0$, spricht man von einem *Fakt* oder einer *Einheitsklausel* und lässt das Zeichen $\leftarrow$ meist weg.

Fakten und Regeln bilden eine Wissensgrundlage, aus der man logische Schlüsse ziehen kann. Hierzu definieren wir:

Definition 3.4 (Definites Programm) *Ein* definites Programm *ist eine endliche Menge von definiten Klauseln.*

Definition 3.5 (Definition eines Relationssymbols) *In einem definiten Programm nennt man die Menge aller Klauseln, die mit demselben Relationssymbol p beginnen, die* Definition von p.

In der logischen Programmierung testet man logische Programme auf ihre Konsequenzen. Statt für ein Programm die unkontrollierbare Menge aller logischen Konsequenzen ausgeben zu lassen, formuliert man eine Art Frage an das logische Programm. Um zu wissen, ob es für die Atome $B_1, \ldots, B_n$ und ein definites Programm eine Belegung gibt, so dass alle Klauseln und die Formeln $B_1, \ldots, B_n$ wahr sind, prüft man, ob die gegenteilige Behauptung zum Widerspruch führt. Hierzu benutzt man definite Ziele:

Definition 3.6 (Definites Ziel) *Eine Klausel der Form*

$$\leftarrow B_1, \ldots, B_n$$

nennt man definites Ziel *(engl. definite goal). Jedes B_i, $1 \leq i \leq n$ heißt* Teilziel.

Definite Ziele heißen auch *Suchen*. Wie oben beschrieben, steht, wenn $y_1, \ldots, y_s$ die in $B_1, \ldots, B_n$ vorkommenden Variablen sind, die Klauselnotation

$$\leftarrow B_1, \ldots, B_n$$

für die Formel

$$\forall y_1 \ldots \forall y_s((\neg B_1) \vee \ldots \vee (\neg B_n))$$

bzw. die dazu äquivalente Formel

$$\neg(\exists y_1 \ldots \exists y_s(B_1 \wedge \ldots \wedge B_n))$$

Dies entspricht der gewünschten Formulierung. Den Widerspruch notieren wir durch die leere Klausel:

Definition 3.7 (Leere Klausel) *Die* leere Klausel $\square$ *ist die Klausel, für die sowohl Kopf als auch Körper leer sind. Diese Klausel ist unwahr in jeder Interpretation.*

Da wir uns in dieser Arbeit ausschließlich mit definiten Zielen und definiten Programmen beschäftigen werden, lassen wir das Adjektiv definit häufig weg.

3.2 Herbrand-Modelle

Dieser Abschnitt beschäftigt sich mit der deklarativen Interpretation definiter Programme. Da man allein durch Regeln und Fakten keine Widersprüche erzeugen kann, dürfen wir davon ausgehen, dass jedes definite Programm ein Modell besitzt. Wir werden zeigen, dass daraus auch die Existenz eines sogenannten *Herbrand-Modells* folgt. Diese Art des Modells ist für die logische Programmierung besonders interessant, da das kleinste Herbrand-Modell eines definiten Programmes genau aus der Menge aller Grundinstanzen seiner atomaren logischen Konsequenzen besteht. Wegen dieser Eigenschaft sind Herbrand-Modelle ein wichtiges Hilfsmittel, die Bedeutung definiter Klauseln zu erfassen.

Wir beginnen mit der Definition der Trägermenge der späteren Herbrand-Modelle:

Definition 3.8 (Herbrand-Universum und Herbrand-Basis) *Sei $\mathcal{A}$ ein Alphabet, das mindestens ein Konstantensymbol enthält. Die Menge $U_\mathcal{A}$ aller Grundterme, die man aus Konstantensymbolen und Funktionssymbolen aus $\mathcal{A}$ konstruieren kann, heißt* Herbrand-Universum *von $\mathcal{A}$. Die Menge $B_\mathcal{A}$ aller atomaren Grundformeln über $\mathcal{A}$ nennt man* Herbrand-Basis *von $\mathcal{A}$.*

Im Folgenden wollen wir Herbrand-Basis und Herbrand-Universum gegebener definiter Programme betrachten. Wir gehen davon aus, dass diese über einem beliebigen, aber fest gewählten Alphabet $\mathcal{A}$ definiert sind, das mindestens ein Konstantensymbol enthält.
Für ein definites Programm P sei sein Herbrand-Universum U_P das Herbrand-Universum $U_\mathcal{A}$ seines zu Grunde liegenden Alphabets. Die Herbrand-Basis B_P von P sei die Menge aller atomaren Grundformeln, die aus den Funktions- und Konstantensymbolen aus $\mathcal{A}$ und den in P vorkommenden Relationssymbolen gebildet werden können.

Definition 3.9 (Herbrand-Interpretation) *Sei P ein definites Programm. Unter einer* Herbrand-Interpretation $\mathfrak{I}$ *von P versteht man eine Interpretation, so dass:*

- *U_P ist die Trägermenge von $\mathfrak{I}$*
- *für jede Konstante c gilt: $c_\mathfrak{I} := c$*
- *für jedes n-stellige Funktionssymbol f aus $\mathcal{A}$ ist die Funktion $f_\mathfrak{I}$ definiert durch:*

$$f_\mathfrak{I}(x_1, ..., x_n) := f(x_1, ..., x_n)$$

- *für jedes n-stellige Relationssymbol r aus P ist die Relation $r_\mathfrak{I}$ eine Teilmenge von U_P^n (der Menge aller n-Tupel von Grundtermen).*

Durch diese Definition sind die Bedeutungen von Funktionen und Konstanten klar festgelegt. Verschiedene Herbrand-Interpretationen desselben Programmes unterscheiden sich

somit nur in den Bedeutungen der Relationssymbole. Deswegen genügt es, um die jeweilige Interpretation zu kennzeichnen, die Bedeutungen aller Relationssymbole aufzuzählen. Für ein n-stelliges Prädikatensymbol p und eine Herbrand-Interpretation $\mathfrak{I}$ besteht die Bedeutung $p_{\mathfrak{I}}$ von p aus der folgenden Menge von n-Tupeln: $\{\langle t_1, ..., t_n\rangle \in U_P^n | \mathfrak{I} \models p(t_1, ..., t_n)\}$.
Weil die Trägermenge der Herbrand-Interpretation das Herbrand-Universum ist, ist die Herbrand-Interpretation eine Teilmenge der Herbrand-Basis, nämlich: $\{A \in B_P | \mathfrak{I} \models A\}$.
Mit Hilfe der Herbrand-Interpretation lassen sich Herbrand-Modelle definieren:

Definition 3.10 (Herbrand-Modell) *Ein Herbrand-Modell einer Menge von (abgeschlossenen) Formeln ist eine Herbrand-Interpretation, die Modell jeder Formel der Menge ist.*

Für definite Programme und definite Ziele ist die Existenz eines Modells äquivalent zur Existenz eines Herbrand-Modells:

Satz 3.11 *Sei P ein definites Programm und G ein definites Ziel, dann gilt:*

$$P \cup \{G\} \textit{ besitzt ein Modell} \Leftrightarrow P \cup \{G\} \textit{ besitzt ein Herbrand-Modell}$$

Beweis „$\Leftarrow$“ klar.

„$\Rightarrow$“ Sei $\mathfrak{I}'$ ein Modell von $P \cup \{G\}$. Wir definieren eine Herbrand-Interpretation $\mathfrak{I}$ von $P \cup \{G\}$ durch:

$$\mathfrak{I} := \{A \in B_P | \mathfrak{I}' \models A\}$$

Dieses $\mathfrak{I}$ ist ein Modell von $P \cup \{G\}$. Angenommen nicht. Dann existiert eine Grundinstanz einer Klausel oder eines Ziels in $P \cup \{G\}$

$$A_0 \leftarrow A_1, \ldots, A_m \quad (m \geq 0),$$

die nicht wahr in $\mathfrak{I}$ ist (für ein Ziel gilt $A_0 = \square$).
Daher muss in $\mathfrak{I}$ gelten, dass $A_1, \ldots, A_m$ wahr und A_0 falsch ist. Nach Definition von $\mathfrak{I}$ gilt dann aber auch für $\mathfrak{I}'$, dass $A_1, \ldots, A_m$ wahr und A_0 falsch ist. Dies widerspricht aber der Annahme, dass $\mathfrak{I}'$ Modell von $P \cup \{G\}$ ist.

□

Dieser Satz gilt nicht nur für definite Programme, sondern kann auch für beliebige Klauselmengen (sog. allgemeine logische Programme) gezeigt werden. Einen Beweis hierzu findet man bei *Lloyd* [Llo87, S.17].
Ein definites Programm kann mehrere Herbrand-Modelle besitzen, so ist zum Beispiel jede Herbrand-Basis auch Herbrand-Modell. Wir betrachten das kleinste dieser Modelle:

Satz 3.12 (Durchschnittseigenschaft von Herbrand-Modellen) *Sei M eine nicht leere Familie von Herbrand-Modellen eines definiten Programmes P. Dann ist der Schnitt $\mathfrak{I} := \bigcap M$ ein Herbrand-Modell von P.*

Beweis Man sieht leicht, dass $\mathfrak{I}$ eine Herbrand-Interpretation von P ist. Es bleibt zu zeigen, dass $\mathfrak{I}$ ein Modell von P ist:
Angenommen nicht. Dann existiert eine Grundinstanz einer Klausel von P:

$$A_0 \leftarrow A_1, \ldots, A_m \quad (m \geq 0),$$

die nicht wahr in $\mathfrak{I}$ ist. Daraus folgt, dass $\mathfrak{I}$ zwar $A_1, \ldots, A_m$ aber nicht A_0 enthält. Damit muss es aber wiederum ein Herbrand-Modell von P geben, dass $A_1, \ldots, A_m$ aber nicht A_0 enthält. Dieses Modell wäre damit aber auch kein Modell von P.
Widerspruch. □

Weil die Herbrand-Basis auch ein Herbrand-Modell ist, wissen wir, dass der Durchschnitt aller Herbrand-Modelle eines definiten Programmes P nicht leer ist. Diesen Durchschnitt nennen wir das *kleinste Herbrandmodell von P* und bezeichnen ihn mit $\mathcal{M}(P)$. Wie bereits erwähnt ist das kleinste Herbrand-Modell wegen der im folgenden Satz beschriebenen Eigenschaft von besonderer Bedeutung für die logische Programmierung. Der Satz geht auf *van Emden* und *Kowalski* [vEK76, S.737f] zurück.

Satz 3.13 *Das kleinste Herbrand-Modell $\mathcal{M}(P)$ eines definiten Programmes P ist die Menge aller atomaren Grundformeln, die logische Konsequenz des Programmes sind, also $\mathcal{M}(P) = \{A \in B_P | P \models A\}$.*

Beweis Es gilt:
$P \models A$
gdw. $P \cup \{\neg A\}$ ist unerfüllbar, nach Lemma 2.17
gdw. $P \cup \{\neg A\}$ besitzt kein Herbrand-Modell, nach Satz 3.11
gdw. $\neg A$ ist falsch in allen Herbrand-Modellen von P
gdw. A ist wahr in allen Herbrand-Modellen von P
gdw. $A \in \mathcal{M}(P)$ □

Damit ist die Frage nach den logischen Konsequenzen eines definiten Programmes äquivalent zu der Frage nach seinem kleinsten Herbrand-Modell. Um eine genauere Charakterisierung von $\mathcal{M}(P)$ zu erhalten, kann man Fixpunktkonzepte benutzen. Hierzu definieren wir zunächst die folgende Abbildung:

Definition 3.14 *Sei* grund(P) *die Menge aller Grundinstanzen von Klauseln in P. T_P ist eine Funktion auf allen Herbrand-Interpretationen von P, die folgendermaßen definiert ist:*

$$T_P(\mathfrak{I}) := \{A_0 \in B_P | A_0 \leftarrow A_1, ..., A_m \in \text{grund}(P) \wedge \{A_1, ..., A_m\} \subseteq \mathfrak{I}\}$$

Die Abbildung T_P ist monoton und besitzt das Herbrand-Modell $\mathcal{M}(P)$ als kleinsten Fixpunkt. Um dies festzuhalten, führen wir die folgende Notation ein:

$$\begin{aligned} T_P \uparrow 0 &:= \emptyset \\ T_P \uparrow (i+1) &:= T_P(T_P \uparrow i) \\ T_P \uparrow \omega &:= \bigcup_{i=0}^{\infty} T_P \uparrow i \end{aligned}$$

Damit gilt nun:

Satz 3.15 *Sei P ein definites Programm und $\mathcal{M}(P)$ dessen kleinstes Herbrand-Modell. Dann gilt:*

- *$\mathcal{M}(P)$ ist die kleinste Herbrand-Interpretation, so dass $T_P(\mathcal{M}(P)) = \mathcal{M}(P)$ (d.h. $\mathcal{M}(P)$ ist der kleinste Fixpunkt von T_P)*
- *$\mathcal{M}(P) = T_P \uparrow \omega$*

Einen Beweis dieses Satzes findet man z.B. bei *Lloyd* [Llo87] oder bei *van Emden* und *Kowalski* [vEK76].

Wir schließen dieses Kapitel mit einer weiteren Eigenschaft des kleinsten Herbrand-Modells. Diese werden wir in den Beweisen späterer Abschnitte benutzen. Wir setzen die Grundbegriffe mathematischer Bäume voraus.[1]

Definition 3.16 (Implikationsbaum) *Ein* Implikationsbaum *für ein Atom B und ein Programm P ist ein endlicher Baum T von Atomen mit Wurzel B, so dass für jedes $A \in T$ eine Instanz $A \leftarrow A_1, \ldots, A_n$ einer Klausel aus P existiert und die Töchter von A in T genau die Atome $A_1, \ldots, A_n$ sind. Die leere Klausel besitzt keine Töchter. Ein Grund-Implikationsbaum ist ein Baum, dessen Knoten Grundatome sind.*

Lemma 3.17 *Für ein Programm P und ein Atom A ist äquivalent:*

(i) $A \in \mathcal{M}(P)$,

(ii) es existiert ein Grund-Implikationsbaum für A.

Der Beweis dieses Lemmas kann bei *K. Doets* [Doe94, S.74] nachgelesen werden.

3.3 Unifikation

Als nächstes soll gezeigt werden, wie man definite Ziele und definite Programme prozedural verarbeiten kann. Bevor wir im nächsten Abschnitt mit der SLD-Resolution ein

[1] Eine Einführung gibt *K. Doets* in [Doe94, Abschnitt 1.3].

Verfahren hierzu kennen lernen werden, benötigen wir ein wichtiges Hilfsmittel, einen Algorithmus zur *Unifikation* von Termen.
Unter Unifikation zweier Terme versteht man die Anwendung einer Substitution auf diese, so dass sie identisch sind. Ein Unifikationsalgorithmus findet, wenn vorhanden, eine solche Substitution. Die Idee zu dieser Art des Algorithmus stammt von *J. Herbrand* [Her67]. Umgesetzt wurde sie erstmals von *J. A. Robinson* [Rob65] im Kontext des automatischen Beweisens. *R. Kowalski* [Kow74] hat dieses Prinzip auf die Programmierung übertragen. Der in diesem Kapitel eingeführte Algorithmus geht auf *Martelli und Montanari* [MM82] zurück.

Wir beginnen mit der genaueren Charakterisierung der zu findenden Substitution, des *allgemeinsten Unifikators*:

Definition 3.18 (Unifikator) *Seien s und t Terme. Eine Substitution θ, so dass $s\theta$ und $t\theta$ identisch sind (geschrieben $s\theta = t\theta$), heißt* Unifikator *von s und t. Existiert für zwei Terme ein Unifikator, so nennen wir diese* unifizierbar.

Wir werden die Suche nach dem allgemeinsten Unifikator zweier Terme s und t als den Prozess zur Lösung der Gleichung $s = t$ betrachten. Hierzu sei ein Unifikator der Gleichungsmenge $\{s_0 = t_0, \ldots, s_n = t_n\}$ eine Substitution θ, so dass $s_i\theta = t_i\theta$ für $0 \leq i \leq n$ gilt.

Definition 3.19 (Allgemeinheit von Substitutionen) *Seien θ und τ Substitutionen. Wir nennen θ* allgemeiner als τ *(engl. more general), gdw. für eine Substitution η gilt: $\tau = \theta\eta$. Wir schreiben dies als $\tau \preceq \theta$.*

Definition 3.20 (Allgemeinster Unifikator) *Seien s und t Terme. Ein Unifikator θ heißt* allgemeinster Unifikator *(engl.* most general unifier*,* mgu*) von s und t, genau dann wenn er allgemeiner als alle anderen Unifikatoren von s und t ist.*

Die Relation $\preceq$ ist reflexiv und transitiv, aber, anders als vielleicht erwartet, nicht antisymmetrisch. Beispielsweise gilt $\{x/y\} \preceq \{y/x\}$, wegen $\{x/y\} = \{y/x\}\{x/y\}$, aber auch $\{y/x\} \preceq \{x/y\}$, wegen $\{y/x\} = \{x/y\}\{y/x\}$. Damit kann es mehrere allgemeinste Unifikatoren für ein Termpaar geben. Die folgenden Sätze zeigen einen Zusammenhang zwischen diesen auf:

Satz 3.21 *Sei θ ein mgu von s und t und ω eine Umbenennung. Dann ist $\theta\omega$ ein mgu von s und t.*

Beweis Weil mit $t\theta = s\theta$ auch $t\theta\omega = s\theta\omega$ gilt, ist $\theta\omega$ ein Unifikator. Sei ω^{-1} nun die zu ω inverse Umbenennung und τ ein Unifikator von s und t. Dann existiert eine Substitution η, so dass $\tau = \theta\eta$ gilt. Damit gilt aber auch $\tau = \theta\omega(\omega^{-1}\eta)$. Daraus folgt $\tau \preceq \theta\omega$. □

Umgekehrt lässt sich zeigen, dass alle allgemeinsten Unifikatoren bis auf Umbenennung gleich sind.

Satz 3.22 *Seien α und β Substitutionen. Gilt $\alpha \preceq \beta$ und $\beta \preceq \alpha$, dann existiert eine Umbenennung ω, so dass $\alpha = \beta\omega$ (und $\beta = \alpha\omega^{-1}$) gilt.*

Beweis Wegen $\alpha \preceq \beta$ und $\beta \preceq \alpha$, existieren Substitutionen η' und μ', so dass $\alpha = \beta\eta'$ und $\beta = \alpha\mu'$ gilt. Schränkt man den Definitionsbereich von η' auf die Variablen aus dem Bild von β und den von μ' auf die Variablen im Bild von α ein, erhält man die Substitutionen η und μ mit den gleichen Eigenschaften. Aus den Formeln folgt $\beta = \alpha\mu = \beta\eta\mu$. Wegen der Einschränkung der Definitionsbereiche muss damit auch $\eta\mu = \epsilon$ gelten. Daraus folgt, dass η und μ Umbenennungen sind. □

Damit unterscheiden sich die allgemeinsten Unifikatoren einer Menge von Termen nur durch verschiedene Variablen. Für die Definition des gesuchten Algorithmus lernen wir zunächst eine Art von Gleichungen kennen, die eng mit dem allgemeinsten Unifikator verbunden ist:

Definition 3.23 (Gelöste Form) *Eine Menge von Gleichungen $\{x_1 = t_1, ..., x_n = t_n\}$ steht in* gelöster Form *(engl. solved form) genau dann, wenn $x_1, ..., x_n$ verschiedene Variablen sind, von denen keine in $t_1, ..., t_n$ vorkommt.*

Satz 3.24 *Sei $\{x_1 = t_1, ..., x_n = t_n\}$ eine Menge von Gleichungen in gelöster Form. Dann ist $\{x_1/t_1, ..., x_n/t_n\}$ ein (idempotenter) mgu dieser Gleichungsmenge.*

Beweis Definiere:

$$\varepsilon := \{x_1 = t_1, \ldots, x_n = t_n\}$$
$$\theta := \{x_1/t_1, \ldots, x_n/t_n\}$$

Dann ist θ bereits ein idempotenter Unifikator von ε. Es bleibt zu zeigen, dass θ allgemeiner ist als jeder andere Unifikator von ε.
Sei dazu σ ein Unifikator von ε. Dann gilt für $1 \leq i \leq n$: $x_i\sigma = t_i\sigma$. Daraus folgt $x_i/t_i\sigma \in \sigma$ für $1 \leq i \leq n$. σ kann außerdem weitere Paare $y_1/s_1, \ldots, y_m/s_m$ enthalten, für die $\{x_1, \ldots, x_n\} \cap \{y_1, \ldots, y_m\} = \emptyset$ gilt. Damit ist σ von der Form:

$$\{x_1/t_1\sigma, \ldots, x_n/t_n\sigma, y_1/s_1, \ldots, y_m/s_m\}$$

Nun gilt: $\theta\sigma = \sigma$. Daher existiert eine Substitution ω $(=\sigma)$, so dass $\sigma = \theta\omega$. Damit ist θ allgemeiner als σ. Also ist θ ein idempotenter mgu. □

Definition 3.25 (Äquivalenz von Gleichungsmengen) *Zwei Mengen von Gleichungen ε_1 und ε_2 heißen* äquivalent*, wenn sie die gleiche Menge Unifikatoren besitzen.*

Damit besitzen zwei äquivalente Gleichungsmengen auch immer die gleiche Menge an Lösungen in jeder Herbrand-Interpretation.

Definition 3.26 (Unifikationsalgorithmus von Martelli und Montanari) *Sei ε eine endliche Menge von Gleichungen. Unter dem Unifikationsalgorithmus von* Martelli und Montanari *verstehen wir den folgenden Algorithmus:*
Wähle eine beliebige Gleichung $s = t \in \varepsilon$. Ist $s = t$ von der Form

1. *$f(s_1, ..., s_n) = f(t_1, ..., t_n)$ und $n \geq 0$, dann ersetze die Gleichung durch $s_1 = t_1, \ldots, s_n = t_n$*
2. *$f(s_1, ..., s_n) = g(t_1, ..., t_n)$ mit $f \neq g$, dann beende den Algorithmus und gib* failure *aus*
3. *$x = x$, dann lösche die Gleichung*
4. *$t = x$ und t ist keine Variable, dann ersetze die Gleichung durch $x = t$*
5. *$x = t$, $x \notin \mathrm{var}(t)$ und x kommt an einer anderen Stelle in ε vor, dann wende die Substitution $\{x/t\}$ in allen anderen Gleichungen an*
6. *$x = t$, $x \in \mathrm{var}(t)$ und $x \neq t$, dann beende den Algorithmus mit* failure

Wiederhole bis keine Operation mehr möglich ist.

Weil wir in dieser Arbeit keine weiteren Unifikationsalgorithmen kennen lernen werden, bezeichnen wir den hier eingeführten Algorithmus im Folgenden als Unifikationsalgorithmus und verzichten auf die Nennung seiner Erfinder. Der folgende Satz, der ebenfalls auf Martelli und Montanari zurück geht, bestätigt, dass der Unifikationsalgorithmus genau das von uns gewünschte Ziel erfüllt.

Satz 3.27 *Der Unifikationsalgorithmus endet und gibt eine zur Gleichungsmenge ε äquivalente Menge in gelöster Form oder, wenn diese nicht existiert,* failure *aus.*

Beweis 1. Der Algorithmus endet: Wenn der Algorithmus nicht endet, treten insbesondere die Fälle 2 und 6 nicht ein. Für eine gegebene Variable x kann Operation 5 nur einmal ausgeführt werden (durch Ausführung wird x aus jeder anderen Gleichung entfernt und kann durch die Operationen 1, 4 und 5 nicht wieder eingeführt werden). Damit kann Operation 5 für das gesamte Programm nur endlich oft ausgeführt werden und es werden nur endlich viele neue Symbole eingeführt. Die Operationen 1 und 3 reduzieren die Anzahl der in der Gleichungsmenge vorkommenden Symbole. Weil auch diese endlich ist, können diese Operationen ebenfalls nur endlich oft ausgeführt werden. Weil durch Operation 4 eine Gleichung gelöscht wird und die Gleichungsmenge endlich ist, folgt also, dass der Algorithmus endet.

2. Durch jede in 1, 3, 4 und 5 beschriebene Operation wird die Gleichungsmenge in eine äquivalente Gleichungsmenge überführt: Für die Operationen 1, 3 und 4 ist

dies offensichtlich.
Für Operation 5 beachte zunächst, dass mit $x\theta = t\theta$ auch $\theta = \{x/t\}\theta$ gilt. Angenommen eine Anwendung von Operation 5 überführt $\varepsilon_1 := \{x = t\} \cup \varepsilon_2$ in $\varepsilon_1' := \{x = t\} \cup \varepsilon_2\{x/t\}$.
Dann gilt:

$$\begin{aligned} \theta \text{ unifiziert } \varepsilon_1 \quad &\text{gdw.} \quad x\theta = t\theta \text{ und } \theta \text{ unifiziert } \varepsilon_2 \\ &\text{gdw.} \quad x\theta = t\theta \text{ und } \{x/t\}\theta \text{ unifiziert } \varepsilon_2 \\ &\text{gdw.} \quad x\theta = t\theta \text{ und } \theta \text{ unifiziert } \varepsilon_2\{x/t\} \\ &\text{gdw.} \quad \theta \text{ unifiziert } \varepsilon_1' \end{aligned}$$

3. Endet der Algorithmus erfolgreich, so steht die Ausgabemenge in gelöster Form. Weil der Algorithmus Äquivalenzen bewahrt, haben die Eingabemenge und die Lösungsmenge dieselben Unifikatoren. Damit ist der mit der Ausgabemenge in gelöster Form verbundene mgu auch mgu der Eingabemenge.
4. Der Algorithmus endet nur für eine Menge, für die kein Unifikator existiert, mit *failure*. In diesem Fall besitzt auch die Eingabemenge keinen Unifikator.

□

Der Unifikationsalgorithmus lässt sich leicht auf atomare Formeln übertragen, indem man Relationssymbole wie Funktionssymbole behandelt. Somit können durch ihn sowohl Terme als auch Atome unifiziert werden.
Unter bestimmten Umständen kann der hier beschriebene Unifikationsalgorithmus ineffizient werden. Diese Ineffizienz liegt unter anderem an den Punkten 5 und 6 im Algorithmus. Hier wird geprüft, ob $x \in \text{var}(t)$ gilt. Diesen Test nennt man *Occur-Check*. Der Unifikationsalgorithmus von den meisten Prolog-Versionen verzichtet aus Effizienzgründen auf diesen Occur-Check und nimmt damit mögliche Fehler in der Unifikation in Kauf. Darauf werden wir in Abschnitt 4.2 genauer eingehen.

3.4 SLD-Resolution

Dieser Abschnitt behandelt die prozedurale Semantik definiter Programme. Als wichtigstes Hilfsmittel Programme prozedural zu interpretieren lernen wir die SLD-Resolution[2] kennen. Dieses Prinzip, das auf *R. Kowalski* (in [Kow74]) zurück geht, ist eine Verfeinerung des 1965 von *J. A. Robinson* eingeführten Resolutionsprinzips (vgl. [Rob65]). Es liefert eine Möglichkeit, durch Anwendung einer einfachen Regel zu berechnen, ob die Atome eines definiten Ziels aus den Klauseln eines Programms folgen. An dieser Stelle geben wir eine Einführung in dieses Prinzip. Der Beweis, dass das Resolutionsprinzip tatsächlich das Gewünschte leistet, folgt im nächsten Abschnitt.

[2]Dieser Ausdruck steht für: ***S****elective* ***L****inear* ***D****efinite clause resolution*. Es handelt sich hierbei um eine lineare Resolution für definite Klauseln, die nach einer *Auswahlregel* (engl. selection rule) vorgeht.

Wir beginnen mit der Definition der zentralen Bausteine der SLD-Ableitung:

Definition 3.28 (Auswahlregel) *Eine* Auswahlregel $\mathfrak{R}$ *(engl. selection rule) ist eine Abbildung von einer Menge von definiten Zielen in eine Menge von Atomen, so dass der Funktionswert für ein Ziel ein Atom aus dem Ziel, genannt* ausgewähltes Atom*, ist.*

Definition 3.29 (Resolutions-Schritt) *Seien $G_i = \leftarrow A_1, \ldots, A_k, \ldots, A_m$ ein definites Ziel und $C_{i+1} = B_0 \leftarrow B_1, \ldots, B_n$ eine definite Klausel, die keine Variablen gemeinsam haben, und $\mathfrak{R}$ eine Auswahlregel. Wir nennen G_{i+1}* abgeleitet *aus G_i und C_{i+1} mittels des allgemeinsten Unifikators θ_{i+1} und der Auswahlregel $\mathfrak{R}$, wenn gilt:*

a) A_k ist das ausgewählte Atom von $\mathfrak{R}$ angewendet auf $A_1, \ldots, A_m$.

b) θ_{i+1} ist der allgemeinste Unifikator von A_k und B_0.

c) G_{i+1} ist das Ziel $\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_n, A_{k+1}, \ldots, A_m)\theta_{i+1}$.

In diesem Falle nennen wir G_{i+1} den Resolventen *von G_i und C_{i+1}. Die Ableitung mittels dieser Regel nennen wir* Resolutions-Schritt*.*

Die wiederholte Anwendung eines Resolutions-Schrittes auf ein definites Programm, beginnend mit einem definiten Ziel G_0, nennen wir SLD-Ableitung:

Definition 3.30 (SLD-Ableitung) *Sei G_0 ein definites Ziel, P ein definites Programm und $\mathfrak{R}$ eine Auswahlregel. Eine* SLD-Ableitung *(auch SLD-Derivation) von $P \cup \{G_0\}$ mittels $\mathfrak{R}$ besteht aus einer (endlichen oder unendlichen) Folge von Zielen $G_0, G_1, \ldots$, einer Folge $C_1, C_2, \ldots$ von Varianten von Klauseln aus P und einer Folge $\theta_1, \theta_2, \ldots$ von mgus, so dass jedes G_{i+1} aus G_i und C_i mittels θ_i und $\mathfrak{R}$ abgeleitet ist. Dies stellen wir folgendermaßen dar:*

$$G_0 \xrightarrow{C_1,\theta_1} G_1 \xrightarrow{C_2,\theta_2} G_2 \cdots G_{n-1} \xrightarrow{C_n,\theta_n} G_n \cdots$$

wobei wir, wenn die verwendete Klausel C_i im jeweiligen Zusammenhang nicht wichtig ist, diese weg lassen.

Um den Voraussetzungen des Resolutions-Schrittes zu genügen, müssen die Varianten C_i der Programmklauseln so gewählt sein, dass die Mengen der Variablen in C_i und in G_{i-1} disjunkt sind. In der Praxis erhält man dies, indem man die Variablen der jeweils verwendeten Klauseln aus P mit Indices versieht. Die in C_1 vorkommenden Variablen erhalten Index 1, die in C_2 vorkommenden Variablen den Index 2 usw. Diese Art der Benennung nennt man *standardisation apart*. Jede Variante C_i einer Programmklausel nennt man *Eingabeklausel* der SLD-Ableitung.

Eine SLD-Ableitung kann auf zwei Weisen enden: Entweder ist der erreichte Resolvent G_i leer, so dass kein Atom ausgewählt werden kann, oder es existiert keine Klausel in P, deren Kopf mit G_i unifizierbar ist. Formal definieren wir diese Fälle folgendermaßen:

Definition 3.31 (SLD-Zurückweisung) *Eine endliche SLD-Ableitung von $P \cup \{G\}$, die $\square$ als letztes Ziel in der Ableitung hat, nennt man* SLD-Zurückweisung *(engl. SLD-refutation) von $P \cup \{G\}$. Gilt $G_n = \square$ so sagt man, die SLD-Zurückweisung habe die* Länge *n.*

Definition 3.32 (Fehlgeschlagene Ableitung) *Eine Ableitung einer Zielklausel G_0 und eines Programmes P, deren letztes Element nicht leer ist und bei der kein weiterer Resolutionsschritt anwendbar ist, nennt man* fehlgeschlagene Ableitung *(engl. failed derivation).*

Die SLD-Zurückweisung kann man als einen Beweis durch Widerspruch verstehen. Man wendet den Resolutions-Schritt so lange an, bis man einen Widerspruch, die leere Klausel, erhält. Aus der Formulierung eines definiten Zieles $\leftarrow A_1, \ldots, A_n$ bzw. in Notation von Logik erster Stufe $\neg(\exists(A_1 \wedge \cdots \wedge A_n))$ geht dann hervor, dass es für die behauptete Aussage ein Gegenbeispiel gibt. Die SLD-Zurückweisung liefert dieses Gegenbeispiel in Form einer sogenannten *Antwort*. Hierbei handelt es sich um die Substitution, die man auf das definite Ziel anwenden muss, um das gesuchte Gegenbeispiel zu erhalten:

Definition 3.33 (Antwort) *Sei P ein definites Programm und G ein definites Ziel. Eine* Antwort *auf $P \cup \{G\}$ ist eine Substitution für die Variablen von G.*

Definition 3.34 (Berechnete Antwort) *Sei P ein definites Programm und G ein definites Ziel. Die* berechnete Antwort *θ einer SLD-Zurückweisung von $P \cup \{G\}$ ist die Substitution, die man aus den in der Zurückweisung benutzten Substitutionen $\theta_1, \ldots, \theta_n$ erhält, wenn man ihre Komposition $\theta_1 \ldots \theta_n$ auf die Variablen von G einschränkt.*

Bevor wir einen Beweis für Korrektheit und Vollständigkeit der SLD-Resolution geben, erläutern wir diese anhand eines Beispiels. Gegeben seien die Klauseln:

(1) $g(x, z) \leftarrow f(x, y), p(y, z)$

(2) $p(x, y) \leftarrow f(x, y)$

(3) $p(x, y) \leftarrow m(x, y)$

(4) $f(a, b)$

(5) $m(b, c)$

und das definite Ziel $\leftarrow g(a, x)$, wobei g, f, p, m Relationen seien, x, y, z Variablen und a, b, c Konstanten. Wir benutzen die Auswahlregel $\mathfrak{R}$, die immer die atomare Formel auswählt, die an erster Stelle im definiten Ziel liegt. Für den ersten Resolutionsschritt wählen wir die Klausel (1) und erhalten mittels standardisation apart und des Unifikationsalgorithmus:

$$\begin{aligned} G_0 &= \leftarrow g(a,x) \end{aligned}$$

$$\begin{aligned} C_1 &= g(x_1,z_1) \leftarrow f(x_1,y_1), p(y_1,z_1) \\ \theta_1 &= \{x_1/a, x/z_1\} \\ G_1 &= \leftarrow f(a,y_1), p(y_1,z_1) \end{aligned}$$

Wegen der Auswahlregel müssen wir nun zunächst eine Eingabeklausel für $f(a,y_1)$ finden, hier wählen wir Klausel (4):

$$\begin{aligned} C_2 &= f(a,b) \\ \theta_2 &= \{y_1/b\} \\ G_2 &= \leftarrow p(b,z_1) \end{aligned}$$

Für den nächsten Schritt wählen wir Klausel (3):

$$\begin{aligned} C_3 &= p(x_3,y_3) \leftarrow m(x_3,y_3) \\ \theta_3 &= \{x_3/b, z_1/y_3\} \\ G_3 &= \leftarrow m(b,y_3) \end{aligned}$$

Nun benutzen wir Klausel (5):

$$\begin{aligned} C_4 &= m(b,c) \\ \theta_4 &= \{y_3/c\} \\ G_4 &= \square \end{aligned}$$

Wir erhalten eine SLD-Zurückweisung. Es ergibt sich:

$$\begin{aligned} \theta_1\theta_2\theta_3\theta_4 &= \{x_1/a, x/z_1\}\{y_1/b\}\{x_3/b, z_1/y_3\}\{y_3/c\} \\ &= \{x_1/a, x/c, y_1/b, x_3/b, z_1/c, y_3/c\} \end{aligned}$$

Die berechnete Antwort ist die Substitution $\theta = \{x/c\}$.

Im nächsten Abschnitt werden wir sehen, dass nun folgt, dass $g(a,c)$ logische Konsequenz der Klauselmenge ist.
Man hätte bei Schritt 3 auch eine andere Wahl für die Eingabeklausel treffen können. Die Resolution wäre so fortgesetzt worden:

$$\begin{aligned} C_3' &= p(x_3,y_3) \leftarrow f(x_3,y_3) \\ \theta_3' &= \{x_3/b, z_1/y_3\} \\ G_3' &= \leftarrow f(b,y_3) \end{aligned}$$

$$\begin{aligned} C_4' &= f(a,b) \end{aligned}$$

Weil $f(b,y_3)$ und $f(a,b)$ nicht unifizierbar sind, handelt es sich hierbei um eine fehlgeschlagene Ableitung.
Eine nicht endende Ableitung erhält man beispielsweise durch Hinzufügen und Verwendung der Klausel $g(x,y) \leftarrow g(x,y)$.

3.5 Vollständigkeit und Korrektheit

In diesem Abschnitt prüfen wir, wie prozedurale und deklarative Semantik definiter Programme zusammenhängen. So zeigen wir, dass die SLD-Resolution richtige Ergebnisse liefert und gewinnen einen Eindruck ihrer Stärke. Beim Beweis der Korrektheit folgen wir dem Buch von *J. W. Lloyd* [Llo87], der Beweis zur Vollständigkeit ist dem Beweis von *K. Apt*, den er in [Apt97] gibt, nachempfunden.

Zur Vereinfachung unserer Ausführungen definieren wir das deklarative Gegenstück der berechneten Antwort:

Definition 3.35 (Korrekte Antwort) *Sei P ein definites Programm und G ein definites Ziel $\leftarrow A_1, \ldots, A_k$ und θ eine Antwort auf $P \cup \{G\}$. Wir nennen θ eine* korrekte Antwort *auf $P \cup \{G\}$, wenn $\forall((A_1 \wedge \cdots \wedge A_k)\theta)$ eine logische Konsequenz von P ist.*

Um den Zusammenhang zwischen dem kleinsten Herbrand-Modell und der korrekten Antwort zu verdeutlichen, halten wir fest:

Satz 3.36 *Sei P ein definites Programm und G ein definites Ziel $\leftarrow A_1, \ldots, A_k$. Sei θ eine Antwort auf $P \cup \{G\}$, so dass $(A_1 \wedge \cdots \wedge A_k)\theta$ eine Grundformel ist. Dann sind äquivalent:*

(i) θ *ist korrekt*

(ii) $(A_1 \wedge \cdots \wedge A_k)\theta$ *ist in jedem Herbrand-Modell von P wahr*

(iii) $(A_1 \wedge \cdots \wedge A_k)\theta$ *ist im kleinsten Herbrand-Modell von P wahr*

Beweis Es genügt, $(iii) \Rightarrow (i)$ zu zeigen. Es gilt:
$(A_1 \wedge \cdots \wedge A_k)\theta$ ist wahr in $\mathcal{M}(P)$

$\Rightarrow$ $(A_1 \wedge \cdots \wedge A_k)\theta$ ist in allen Herbrand-Modellen von P wahr

$\Rightarrow$ $\neg(A_1 \wedge \cdots \wedge A_k)\theta$ ist falsch in allen Herbrand-Modellen von P

$\Rightarrow$ $P \cup \{\neg(A_1 \wedge \cdots \wedge A_k)\}$ besitzt kein Herbrand-Modell

$\Rightarrow$ $P \cup \{\neg(A_1 \wedge \cdots \wedge A_k)\}$ besitzt kein Modell

□

Zum Beweis der Korrektheit der SLD-Resolution zeigen wir, dass jede berechnete Antwort korrekt ist.

Satz 3.37 (Korrektheit der SLD-Resolution) *Sei P ein definites Programm, G ein definites Ziel. Dann ist jede berechnete Antwort auf $P \cup \{G\}$ auch eine korrekte Antwort auf $P \cup \{G\}$.*

Beweis Sei G von der Form $\leftarrow A_1, \ldots, A_k$, sei $\theta_1, \ldots, \theta_n$ die Folge der bei einer SLD-Zurückweisung von $P \cup \{G\}$ benutzten mgus und θ die zugehörige korrekte Antwort. Wir zeigen per Induktion über die Länge n der Zurückweisung die Gültigkeit von

$$P \models \forall((A_1 \wedge \cdots \wedge A_k)\theta_1 \ldots \theta_n).$$

Weil θ die Einschränkung von $\theta_1 \ldots \theta_n$ auf die Variablen von G ist, folgt damit die Behauptung.

Induktionsanfang: Für $n = 1$ ist G ein Ziel der Form $\leftarrow A_1$ und es existiert eine Programmklausel (genauer ein Fakt) $B \leftarrow$, so dass $A_1\theta_1 = B\theta_1$ gilt. Weil $A\theta_1$ eine Instanz eines Faktes von P ist, folgt $P \models \forall(A_1\theta_1)$.

Induktionsschritt: Das zu Zeigende gelte für berechnete Antworten aus SLD-Zurückweisungen der Länge $n-1$. Sei nun $\theta_1, \ldots, \theta_n$ die Folge der mgus, die in einer SLD-Zurückweisung der Länge n benutzt wurden. Sei $B_0 \leftarrow B_1, \ldots, B_q$, $(q > 0)$ die erste Eingabeklausel dieser Zurückweisung und A_m das erste ausgewählte Atom von G. Dann gilt $B_0\theta_1 = A_m\theta_1$ und der SLD-Resolvent ist von der Form:

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_1$$

Nach Induktionsvoraussetzung gilt:

$$P \models \forall(((A_1 \wedge \cdots \wedge A_{m-1} \wedge B_1 \wedge \cdots \wedge B_q \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta_1)\theta_2 \ldots \theta_n).$$

Weil außerdem $(B_0 \leftarrow B_1, \ldots, B_q)\theta_1 \ldots \theta_n$ die Instanz einer Klausel von P ist, folgt:

$$P \models \forall((A_1 \wedge \cdots \wedge A_{m-1} \wedge B_0 \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta_1 \ldots \theta_n).$$

Wegen $A_m\theta = B_0\theta$ gilt damit:

$$P \models \forall((A_1 \wedge \cdots \wedge A_{m-1} \wedge A_m \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta_1 \ldots \theta_n).$$

□

Aus diesem Satz folgt nun direkt:

Korollar 3.38 *Sei P ein definites Programm und G ein definites Ziel und es existiere eine SLD-Zurückweisung von $P \cup \{G\}$. Dann ist $P \cup \{G\}$ unerfüllbar.*

Beweis Sei G von der Form $\leftarrow A_1, \ldots, A_k$. Nach Satz 3.37 ist die mittels der Zurückweisung berechnete Antwort θ korrekt. Also ist $\forall((A_1 \wedge \cdots \wedge A_k)\theta)$ eine logische Konsequenz von $P \cup \{G\}$. Damit ist $P \cup \{G\}$ unerfüllbar. □

Die genaue Umkehrung des obigen Satzes, dass jede korrekte auch eine berechnete Antwort ist, gilt so nicht. Eine berechnete Antwort ist die Komposition allgemeinster Unifikatoren. Angewendet auf die Atome eines definiten Ziels kann es vorkommen, dass man nur Instanzen der Atome erhält, die in gewisser Weise allgemeiner sind als die atomaren Formeln, die sich durch Anwendung einer korrekten Antwort ergeben. Es lässt sich jedoch zeigen, dass für jede korrekte Antwort θ eine berechnete Antwort η existiert, so dass für ein definites Ziel G das Ziel $G\theta$ eine Instanz von $G\eta$ ist. Dies gilt sogar unabhängig von der verwendeten Auswahlregel. Um dies zu beweisen, benutzen wir folgendes Lemma:

Definition 3.39 (Tiefe eines Ziels) *Sei P ein definites Programm und G ein definites Ziel. Wir sagen G* hat die Tiefe *n, wenn jedes Atom in G einen Implikationsbaum bzgl. P besitzt und die Summe aller Knoten dieser Bäume n ist.*

Lemma 3.40 *Sei P ein definites Programm und $G\theta$ ein definites Ziel der Tiefe n, $n \geq 0$. Dann existiert für jede Auswahlregel $\mathfrak{R}$ eine SLD-Zurückweisung von $P \cup \{G\}$ mittels der berechneten Antwort η und $\mathfrak{R}$, so dass $G\theta$ eine Instanz von $G\eta$ ist.*

Beweis Wir konstruieren für $i \in [0, n]$ induktiv eine SLD-Ableitung

$$G_0 = G \overset{\theta_1}{\longrightarrow} G_1 \cdots \overset{\theta_i}{\longrightarrow} G_i$$

von $P \cup \{G\}$ mittels $\mathfrak{R}$ und eine Folge von Substitutionen $\gamma_0, \ldots, \gamma_i$, so dass gilt:

- $G\theta = G\theta_1 \ldots \theta_i\gamma_i$,
- $G_i\gamma_i$ besitzt die Tiefe $(n - i)$.

Dann besitzt $G_n\gamma_n$ Tiefe 0, es gilt also $G_n = \square$. Damit ist

$$G_0 = G \overset{\theta_1}{\longrightarrow} G_1 \cdots \overset{\theta_n}{\longrightarrow} G_n$$

die gewünschte SLD-Zurückweisung.

Induktionsanfang: Für $i = 0$ definiere $\gamma_0 := \theta$.

Induktionsschritt: Das zu Zeigende sei für i bewiesen. Sei B_k das mittels $\mathfrak{R}$ ausgewählte Atom aus $G_i = \leftarrow B_1, \ldots, B_q$. Aus der Induktionsvoraussetzung folgt, dass für $B_k\gamma_i$ ein Implikationsbaum mit r Knoten, $1 \leq r \leq (n - i)$, existiert. Deswegen gibt es eine Klausel $C := A_0 \leftarrow A_1, \ldots, A_p$ in P und eine Substitution τ, so dass $B_k\gamma_i = A_0\tau$ gilt und

$$\leftarrow (A_1, \ldots, A_p)\tau \text{ die Tiefe } (r - 1) \tag{3.1}$$

besitzt.
Sei π nun eine Umbenennung, so dass die Variablen von $C\pi$ disjunkt zu denen in G,

denen in G_i, denen aus den bisher verwendeten Substitutionen und denen der bisher verwendeten Eingabeklauseln sind. Die Substitution α sei die Vereinigung der Substitution γ_i, eingeschränkt auf $\text{var}(G\theta_1 \dots \theta_i) \cup \text{var}(G_i)$ und der Substitution $\pi^{-1}\tau$ eingeschränkt auf $\text{var}(C\pi)$. Aus der Wahl von π folgt, dass α wohldefiniert ist. Außerdem gilt:

$$B_k\alpha = B_k\gamma_i = A_0\tau = A_0\pi(pi^{-1}\tau) = (A_0\pi)\alpha$$

damit sind B_k und $H\pi$ unifizierbar. Sei θ_{i+1} der mgu von B_k und $A_0\pi$. Dann existiert ein γ_{i+1}, so dass

$$\alpha = \theta_{i+1}\gamma_{i+1} \tag{3.2}$$

Sei $G_{i+1} :=\leftarrow (B_1, \dots, B_{k-1}, A_1\pi, \dots, A_p\pi, B_{k+1}, \dots, B_q)\theta_{i+1}$ der nächste Resolvent in der konstruierten SLD-Ableitung. Dann gilt

$$G\theta \overset{\text{I.V.}}{=} G\theta_1 \dots \theta_i\gamma_i \overset{\text{Def. } \alpha}{=} G\theta_1 \dots \theta_i\alpha \overset{3.2}{=} G\theta_1 \dots \theta_i\theta_{i+1}\gamma_{i+1}$$

und

$$\begin{aligned} G_{i+1}\gamma_{i+1} &= \leftarrow (B_1, \dots, B_{k-1}, A_1\pi, \dots, A_p\pi, B_{k+1}, \dots, B_q)\theta_{i+1}\gamma_{i+1} \\ &\overset{3.2}{=} \leftarrow (B_1, \dots, B_{k-1}, A_1\pi, \dots, A_p\pi, B_{k+1}, \dots, B_q)\alpha \\ &\overset{\text{Def. } \alpha}{=} \leftarrow B_1\gamma_i, \dots, B_{k-1}\gamma_i, A_1\tau, \dots, A_p\tau, B_{k+1}\gamma_i, \dots, B_q\gamma_i. \end{aligned}$$

Aus der Induktionsvoraussetzung und 3.1 folgt damit, dass $G_{i+1}\gamma_{i+1}$ die Tiefe $(n-(i+1))$ besitzt. □

Damit folgt:

Satz 3.41 (Starke Vollständigkeit der SLD-Resolution) *Sei P ein definites Programm, G ein definites Ziel und θ eine korrekte Antwort auf $P \cup \{G\}$. Dann existiert für jede Auswahlregel $\mathfrak{R}$ eine SLD-Zurückweisung von $P \cup \{G\}$ mittels $\mathfrak{R}$ und einer berechneten Antwort η, so dass $G\theta$ eine Instanz von $G\eta$ ist.*

Beweis Sei G von der Form $\leftarrow A_1, \dots, A_n$. Wir beweisen das zu Zeigende zunächst für den Fall, dass θ eine Grund-Substitution ist:
Nach Definition 3.35 ist die Grundformel $(A_1 \wedge \dots \wedge A_n)\theta$ logische Konsequenz von P, damit ist auch jedes Grundatom $A_i\theta$ logische Konsequenz von P. Wegen Satz 3.13 gilt damit $A_i \in \mathcal{M}(P)$. Wegen Lemma 3.17 existiert für jedes $A_i\theta$ ein Implikationsbaum, damit besitzt $G\theta$ eine Tiefe $n \geq 0$. Durch Anwendung von Lemma 3.40 erhalten wir das Gewünschte.
Sei θ nun keine Grundsubstitution. Um die Existenz endlicher Implikationsbäume für $A_1, \dots, A_n$ zu beweisen, erweitern wir das Alphabet $\mathcal{A}$, in dem das Programm geschrieben ist. Für jede Variable x aus $G\theta$ sei c_x ein neues Konstantensymbol. Sei $\mathcal{A}' = \mathcal{A} \cup \{c_x | x \in \text{var}(G\theta)\}$ das neue zu Grunde liegende Alphabet von P und σ eine Substitution, die jeder Variable $x \in \text{var}(G\theta)$ das jeweilige Konstantensymbol c_x zuordnet. Dann existiert für jedes $A_i\theta\sigma$ ein Grundimplikationsbaum. Da aber die Menge der Konstanten aus σ disjunkt zur Menge der Konstanten aus $P \cup \{G\}$ ist, besitzt auch jedes $A_i\theta$ einen Implikationsbaum. Mit Lemma 3.40 folgt das zu Zeigende. □

Das Attribut „stark“ erhält die Vollständigkeit in diesem Satz durch die Tatsache, dass sie unabhängig von der Auswahlregel gilt. Die Entsprechung des obigen Satzes ohne Bezug auf die Auswahlregel bezeichnet man allgemein mit Vollständigkeit. Diese folgt direkt aus dem Bewiesenen.

Mit diesem Satz schließen wir das Kapitel. Es wurde gezeigt, dass die SLD-Resolution vollständig im Sinne des letzten Satzes ist, dass sie korrekt ist und dass der verwendete Unifikationsalgorithmus zur Ermittlung allgemeinster Unifikatoren ebenfalls korrekt ist. Dies motiviert die maschinelle Umsetzung der hier vorgestellten Verfahren. Diese behandeln wir im nächsten Kapitel.

4 Prolog

Die Programmiersprache *Prolog*[1] wurde Anfang der 70er Jahre von einer Gruppe um den Informatiker *Alain Colmerauer* in Marseille entwickelt[2]. Sie bildet, soweit technisch möglich, eine praktische Umsetzung der Theorie logischer Programmierung. Aufgrund der Anforderungen an eine moderne Programmiersprache ist Prolog jedoch reicher in seinen Möglichkeiten. Wir beschränken uns in diesem Kapitel auf die Betrachtung der Teilmenge der Sprache, die ungefähr der logischen Programmierung entspricht, *reines Prolog*. Für unsere Ziele sind hierbei vor allem die Unterschiede zwischen logischer Programmierung und reinem Prolog und ihre möglichen Auswirkungen auf Korrektheit und Vollständigkeit von Ableitungen definiter Programme und Ziele von Interesse. Als Grundlage für die spätere Behandlung des Programmcodes von Naproche führen wir ebenfalls einige der in Prolog eingebauten Schreibweisen und Funktionen ein, die wir dort verwenden.
Unsere Ausführungen richten sich hauptsächlich nach dem Buch von *K. R. Apt* [Apt97], der letzte Abschnitt nach *M. A. Covington* [Cov94]. Als eine umfassendere und praktischer orientierte Einführung in Prolog sei hier das Buch von *Blackburn et al.* [BBS06] empfohlen.

4.1 Syntax

Die Syntax von Prolog unterscheidet sich nur in wenigen Punkten von der logischer Programmierung. Genau wie definite Programme bestehen Prolog-Programme aus Fakten und Regeln. Da diese jedoch von einem Computer nach einer bestimmten Reihenfolge als Eingabeklauseln ausgewählt werden müssen, bestehen Prolog-Programme nicht aus Mengen, sondern aus Folgen dieser.
Statt des Zeichens „←“ benutzt man in Prolog-Programmen allgemein das Zeichen „:-“, für definite Ziele „?-“. In Anlehnung an die logische Programmierung werden wir den Gebrauch des Implikationspfeiles beibehalten. Hinter jeder Klausel steht ein Punkt „.“. Variablen schreibt man in Prolog als Ausdrücke, die mit Großbuchstaben beginnen, z.B. `Xs` oder `Y`, oder, im Fall der *anonymen Variablen*, mit dem Unterstrich „_“. Steht dieses Symbol in einer Klausel, so wird intern an der entsprechenden Stelle automatisch ein

[1] Die Abkürzung stammt aus dem Französischen: **Pro**grammation en **Log**ique, dt.: Programmieren in Logik.
[2] Mehr zur Geschichte von Prolog findet man bei *Kowalski* [Kow88].

neues Variablensymbol eingeführt, das dann im Programm und im verwendeten definiten Ziel nur an dieser einen Stelle vorkommt.
Zahlen stehen in Prolog für Konstantensymbole. Ausdrücke, die mit Kleinbuchstaben beginnen, stehen für Relations-, Funktions- oder Konstantensymbole. Hierbei müssen, im Gegensatz zur Logik erster Stufe und damit zur logischen Programmierung, die Mengen der Relationssymbole und Funktionssymbole jeder einzelnen Wertigkeit und die Menge der Konstantensymbole nicht jeweils zueinander disjunkt sein. Der Buchstabe `f` kann im selben Programm beispielsweise sowohl für ein Konstantensymbol, als auch für ein Funktionssymbol oder ein Relationssymbol beliebiger Wertigkeit stehen, so wäre der Ausdruck `f(f(f),f)` eine legitime Formel oder ein legitimer Term. Ob es sich bei einem Symbol um ein Funktions- oder Relationssymbol handelt, wird aus dem Kontext deutlich. Wir bezeichnen diese Freiheit mit dem Begriff *ambivalente Syntax*.
Als Folge der beschriebenen Ambivalenz benötigt man eine Modifikation des in Definition 3.26 eingeführten Unifikationsalgorithmus. Statt Punkt 2 der Definition legen wir fest:

2.' $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ wobei $f \neq g$ oder $n \neq m$, dann beende den Algorithmus und gib *failure* aus.

Beziehen wir uns im Folgenden auf ein Relationssymbol oder ein Funktionssymbol `p` der Wertigkeit `n`, so bezeichnen wir dies mit `p/n`.

4.2 Unifikation

Prolog benutzt zur Unifikation den in Definition 3.26 eingeführten Martelli-Montanari-Algorithmus mit der obigen Modifizierung und dem Unterschied, dass es auf den Occur-Check verzichtet. Damit fällt Punkt 6 der Definition weg und im Fall $x = t$ wird immer Operation 5 angewendet. Dies geschieht vor allem aus Effizienz-Gründen und wird pragmatisch damit begründet, dass für die meisten Ableitungen von Programmen und Zielen der Occur-Check nicht verwendet werden muss. Es gibt jedoch Programme, für die der Verzicht auf den Occur-Check das Ergebnis der Unifikation verändert. Wir betrachten ein Beispiel.

Gegeben sei die folgende Programmklausel:

```
p(X,f(X)).
```

Für das definite Ziel ← `p(X,X).` liefert eine Resolution, die den Occur-Check verwendet, keine Antwort, da die Atome `p(X,X)` und `p(X,f(X))` nicht unifizierbar sind. Lässt man jedoch, wie in den meisten Prolog-Versionen üblich, den Occur-Check weg, ergibt sich die Antwort {`X/f(f(f(....`}, hierbei stehen die Punkte für die unendliche Wiederholung der Zeichen „`f(`“, die Prolog-Ausgabe hierfür ist `X=f(**)`.

Obwohl dieses Beispiel sehr einfach ist, wird das Grundproblem an dieser Stelle deutlich:

Das Weglassen des Occur-Checks ändert das Ergebnis der Unifikation. Durch Weiterverwendung des gefundenen Ergebnisses können leicht Endlosschleifen entstehen oder falsche Antworten ausgegeben werden. Damit wirkt sich der Verzicht auf den Occur-Check auch auf die Korrektheit der Resolution aus.
Das beschriebene Problem nennt man *Occur-Check-Problem*. Wie Programme und Ziele beschaffen sein müssen, damit es nicht auftritt, behandeln wir in Abschnitt 5.3.

4.3 Auswahlregel und Suchstrategie

Die Definitionen und Resultate des letzten Kapitels konnten wir für die SLD-Resolution bezüglich beliebiger Auswahlregeln treffen. In der Umsetzung der Theorie muss jedoch eine Auswahlregel festgelegt werden. Das Beispiel am Ende von Abschnitt 3.4 verdeutlicht außerdem, dass auch die konkrete Wahl der Eingabeklausel eines Resolutionsschrittes für den Erfolg der Resolution entscheidend sein kann. Auch diese muss in einem System festgelegt werden. Im Folgenden geht es um die Entscheidungen, die zur Lösung dieser und anderer Probleme für die Programmiersprache Prolog getroffen wurden. Wir erklären Auswahlregel und Berechnungsprozeß und veranschaulichen diese an einem Beispiel.

Wir beginnen mit der Auswahlregel. Hier wird bei Prolog, ebenso wie im erwähnten Beispiel für die SLD-Resolution, folgende Regel verwendet:

Definition 4.1 (Left Most Selection Rule) *Die Auswahlregel einer SLD-Resolution, die immer das Literal auswählt, das am weitesten links liegt, nennt man* left most selection rule.

Statt des Kürzels SLD benutzen wir bei den entsprechenden Begriffen, wenn die left most selection rule benutzt wird, die Abkürzung LD. So legen wir fest:

Definition 4.2 (LD-Resolution) *Eine SLD-Resolution, die als feste Auswahlregel die* left most selection rule *benutzt, nennt man* LD-Resolution.

Entsprechend sind die Begriffe *LD-Ableitung*, *LD-Zurückweisung* und *LD-Resolvent* definiert.
Um die spätere Erklärung für den Berechnungsprozess von Prolog zu erleichtern, führen wir an dieser Stelle folgenden Begriff ein:

Definition 4.3 (LD-Baum) *Sei P ein definites Programm, G ein definites Ziel. Der LD-Baum T von $P \cup \{G\}$ ist ein Baum, der folgende Eigenschaften erfüllt:*

(i) Jeder Knoten von T ist ein (möglicherweise leeres) definites Ziel

(ii) Die Wurzel von T ist G

(iii) *Sei* $\leftarrow A_1, \ldots, A_k$ $(k > 0)$ *ein Knoten in T. Dann besitzt der Knoten für jede Eingabeklausel* $A \leftarrow B_1, \ldots, B_q$*, für die* A_1 *und* A *unifizierbar mit mgu* θ *sind, einen Tochterknoten*

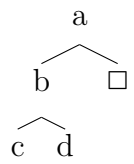
$$\leftarrow (B_1, \ldots, B_q, A_2, \ldots, A_k)\theta$$

(iv) *Ein Knoten, der ein leeres Ziel ist, besitzt keine Töchter*

Gegeben ein definites Programm und ein definites Ziel, zeigt ein LD-Baum alle möglichen Ableitungen mittels der leftmost selection rule auf. Wir veranschaulichen dies an einem Beispiel. Für das definite Ziel $\leftarrow$ `a` und das Programm

```
a  ←  b.
a.
b  ←  c.
b  ←  d.
```

ergibt sich folgender LD-Baum:



Wegen der starken Vollständigkeit der SLD-Resolution (Satz 3.41) wissen wir, dass durch einen LD-Baum, bis auf Umbenennung, alle möglichen berechneten Antworten eines definiten Programmes und definiten Zieles gefunden werden können. Für jeden Zweig, der mit dem leeren Ziel endet, schränken wir die Komposition aller benutzten Substitutionen auf die Variablen des definiten Zieles ein und erhalten die entsprechende berechnete Antwort (in unserem Beispiel: ϵ).
Die Aufgabe eines Prolog-Systems ist es nun, diesen Baum nach berechneten Antworten zu durchsuchen. Hierbei wird nach einer bestimmten Reihenfolge vorgegangen, um dies auszudrücken, betrachten wir sogenannte *geordnete* Bäume, hierunter verstehen wir Bäume, in denen die Tochter-Knoten eines jeden Knotens geordnet sind. Die folgende Definition veranschaulicht nun das Vorgehen:

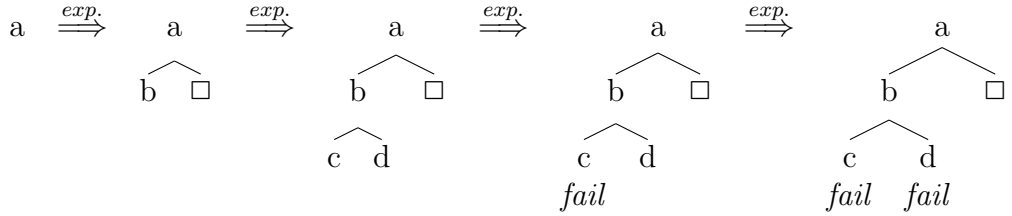
Definition 4.4 (Prolog-Baum) *Für ein gegebenes Programm konstruieren wir einen endlich verzweigten geordneten Baum von eventuell markierten definiten Zielen, indem wir den Operator expand*(T, G) *wiederholt auf das Anfangsziel anwenden. Hierbei sei T der aktuell erzeugte Baum und G das unmarkierte definite Ziel, das am weitesten links steht. expand*(T, G) *sei dabei wie folgt definiert:*

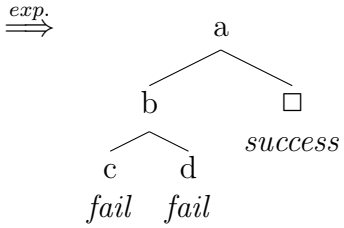
- ***success****: G ist das leere Ziel; markiere G mit* success
- ***fail****: G ist nicht leer und besitzt keine LD-Resolventen; markiere G mit* fail

- ***expand**: G besitzt LD-Resolventen; sei k die Anzahl der Programmklauseln, die auf das ausgewählte Atom anwendbar sind. Füge die k LD-Resolventen von G als Tochterknoten dieses Zieles in T hinzu. Hierbei sei die Reihenfolge der Knoten genau die der Klauseln im Programmcode.*

Als Grenzwert dieses Prozesses erhalten wir einen geordneten Baum von definiten Zielen. Diesen nennen wir Prolog-Baum.

Der Baum aus dem obigen Beispiel wird nun in folgenden Schritten erzeugt und markiert:

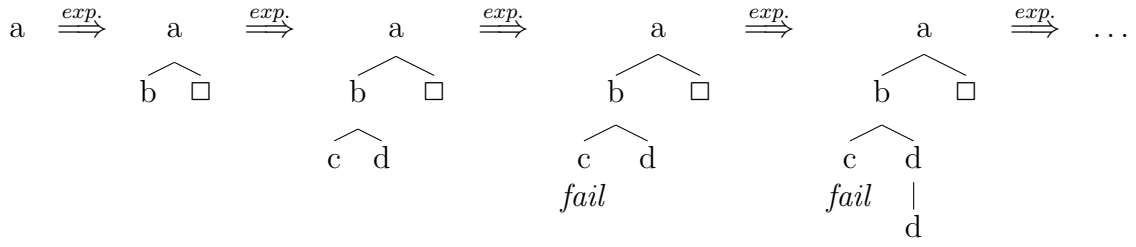




Das Prolog-System verfolgt also zunächst den Pfad zu dem Blatt, das am weitesten links im Prolog-Baum gelegen ist. Dieses Blatt wird mit dem entsprechenden Resultat markiert, die Suche wird bei der letzten Verzweigung des Baumes fortgesetzt. Dieser Vorgang wird wiederholt, bis der gesamte Baum durchsucht ist. Bei erfolgreicher Suche gibt das System eine berechnete Antwort aus, durch einen entsprechenden Befehl kann die Suche fortgesetzt werden.
Das beschriebene Verfahren wurde aus Effizienzgründen gewählt. Für die Korrektheit der Resolution ergeben sich aber auch negative Konsequenzen. Wir erweitern das Beispielprogramm um die Klausel

$$\texttt{d} \leftarrow \texttt{d}.$$

Die Prolog-Resolution beginnt wie im vorherigen Beispiel, bis das definite Ziel $\leftarrow$ d erreicht wird. Hier ist nun ein Resolutionsschritt mittels der ergänzten Klausel möglich. Da der Resolvent das definite Ziel selber ist, kann dieser beliebig oft wiederholt werden, es ergibt sich ein unendlicher Zweig. In diesem Fall findet das Prolog-System die existierende Lösung nicht. Es ergeben sich die Bäume:



Dieses Problem tritt nicht auf, wenn der Prolog-Baum einem LD-Baum entspricht, bei dem jeder Knoten markiert ist. Dies ist unter anderem der Fall, wenn beide Bäume endlich sind. In Abschnitt 5.2 werden wir Eigenschaften definiter Programme und Ziele behandeln, die endliche LD-Bäume garantieren.

4.4 Grammatiken und Listen

Dieser Abschnitt behandelt zwei wichtige Konzepte, die im Programmcode von Naproche verwendet werden, Listen und Grammatiken. Listen bieten in Prolog eine Möglichkeit, Folgen zu speichern und zu verarbeiten. Naproche verwendet außerdem GULP[3], eine Prologerweiterung von *Michael Covington* (vgl. hierzu [Cov94]). In GULP wird das Konzept der Liste erweitert. Es erlaubt die Definition und den Umgang mit sogenannten Merkmalslisten, einer Struktur von Merkmalen und Merkmalswerten.
Grammatiken werden zum Einlesen und Verstehen des Eingabetextes benutzt.

Wir beginnen mit der Einführung von *Listen*. Die Idee hinter diesem Konzept ist es, Folgen mittels einer Konstanten und einer zweistelligen Funktion darzustellen. Ist e diese Konstante und f diese Funktion, so drückt $f(a, f(b, f(c, e)))$ die Folge a, b, c aus. Diese Darstellung macht es sehr einfach, weitere Elemente am Anfang der Folge hinzuzufügen. Ist l eine Folge, dargestellt wie oben, und n ein neues Element, so ist $f(n, l)$ die Folge mit dem neuen Element n an erster Stelle. Jede Datenstruktur, die wie beschrieben nur eine Folgenoperation, das Hinzufügen eines neuen Elements an die erste Stelle, erlaubt, nennen wir *Liste*. Prolog bietet eine Vereinfachung zur Darstellung von Listen. Es benutzt die feste Konstante [ ] und die zweistellige Funktion [.|..]. Damit gilt:

Definition 4.5 (Liste) *Sei* [ ] *eine Konstante. Wir definieren induktiv:*

- [ ] *ist eine Liste*
- *wenn* `xs` *eine Liste ist, so ist auch* [`x`|`xs`] *eine Liste;* `x` *nennt man den* Kopf*,* `xs` *den* Schwanz *der Liste.*

[ ] *nennt man die* leere Liste.

Zur Verbesserung der Lesbarkeit werden Listen vereinfacht dargestellt. Für $n \geq 1$ gilt:

- $[\mathtt{s}_0|[\mathtt{s}_1, \ldots, \mathtt{s}_n|\mathtt{t}]]$ wird abgekürzt durch $[\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_n|\mathtt{t}]$,
- $[\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_n|[\,]]$ wird abgekürzt durch $[\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_n]$

Um Merkmalslisten einzuführen, verdeutlichen wir das Konzept. Es sollen Wert-Merkmals-

[3] Abkürzung für: **G**raph **U**nification **L**ogic **P**rogramming

Paare der folgenden Art dargestellt werden:

$$\begin{bmatrix} merkmal_1 & : & wert_1 \\ merkmal_2 & : & wert_2 \\ merkmal_3 & : & wert_3 \end{bmatrix}$$

GULP verwirklicht diese Darstellung, indem es die Werte in einer Liste speichert, in der jede Stelle einem Merkmal entspricht. Die Reihenfolge der Stellen ist hierbei nach ihrem Vorkommen im Programmcode gewählt. Das erste benannte Merkmal erhält die letzte Position der Liste, das zweite die vorletzte usw. Für jedes neue Merkmal wird ein Listenplatz hinzugefügt. Merkmalslisten, die nur bekannte Merkmale enthalten, werden entsprechend gespeichert. Kommt in einer Merkmalsliste ein schon bekanntes Merkmal nicht vor, wird diesem Merkmal dort intern eine neue Variable als Wert zugewiesen. Hierbei ist die Reihenfolge der einzelnen Wert-Merkmals-Paare nicht wichtig.
Zur Notation von Merkmalslisten werden in GULP Merkmale und ihre Werte durch das Zeichen „∼“ getrennt, verschiedene Wert-Merkmals-Paare durch „..“. Damit ist die obige Beispielliste in GULP-Notation von der Form:

$$merkmal_1 \sim wert_1 \;..\; merkmal_2 \sim wert_2 \;..\; merkmal_3 \sim wert_3$$

Äquivalent dazu ist die Schreibweise

$$merkmal_3 \sim wert_3 \;..\; merkmal_1 \sim wert_1 \;..\; merkmal_2 \sim wert_2$$

und jede andere mögliche Permutation der Wert-Merkmals-Paare.

Merkmale und Werte sind hierbei beliebige Prolog-Terme. Die Unifikation zweier GULP-Listen erfolgt über den Unifikationsalgorithmus von Prolog. Für das folgende Beispiel betrachten wir zunächst die Gleichheitsrelation in Prolog. Diese wird durch das Infixzeichen „=“ ausgedrückt und ist durch folgende Klausel definiert:

$$\mathtt{X} = \mathtt{X}.$$

Die Relation gilt also für zwei Terme, wenn diese unifizierbar sind.
Betrachten wir nun folgendes definite Ziel:

$$\leftarrow \mathtt{X} = \mathtt{m}_1 \sim \mathtt{a} .. \; \mathtt{m}_2 \sim \mathtt{b} .. \; \mathtt{m}_3 \sim \mathtt{c}, \mathtt{X} = \mathtt{m}_1 \sim \mathtt{a} .. \; \mathtt{m}_4 \sim \mathtt{d}.$$

Wenden wir Resolution und Unifikation an, erhalten wir als Lösung:

$$\{\mathtt{X}/\mathtt{m}_1 \sim \mathtt{a} .. \; \mathtt{m}_2 \sim \mathtt{b} .. \; \mathtt{m}_3 \sim \mathtt{c} .. \; \mathtt{m}_4 \sim \mathtt{d}\}$$

Die GULP-Listen sind unifizierbar.

Als weiteres Beispiel betrachten wir das definite Ziel:

$$\leftarrow \mathtt{X} = \mathtt{m}_1 \sim \mathtt{a} .. \; \mathtt{m}_2 \sim \mathtt{b} .. \; \mathtt{m}_3 \sim \mathtt{c}, \mathtt{X} = \mathtt{m}_1 \sim \mathtt{b} .. \; \mathtt{m}_4 \sim \mathtt{d}.$$

Hier existiert keine SLD-Zurückweisung, weil die Wert-Merkmals-Paare $\mathtt{m}_1 \sim \mathtt{a}$ und $\mathtt{m}_1 \sim \mathtt{b}$ nicht unifizierbar sind.

Wir beenden diesen Abschnitt mit der Behandlung eines weiteren in Prolog eingebauten Konzeptes, Grammatiken. Weil die Programmiersprache Prolog gerade auch zur Verarbeitung natürlicher Sprache entwickelt wurde[4], versuchte man die linguistische Theorie der kontextfreien Grammatik in Prolog darstellbar und verwendbar zu machen. Geschaffen wurde hierzu die *Definite Clause Grammar*. Diese erlaubt die in der Linguistik übliche Schreibweise für Grammatiken im Code eines Prolog-Programmes anzuwenden. Wir beschränken uns an dieser Stelle darauf, anhand von Beispielen zu erläutern, wie diese Schreibweisen übersetzt in die Klauselschreibweise logischer Programmierung lauten. Eine genauere Beschreibung der Theorie und ihrer Umsetzung geben *Blackburn et al.* in [BBS06, Kap. 7-8].

Regeln von kontextfreien Grammatiken werden normalerweise mit Hilfe eines Pfeils dargestellt (man denke beispielsweise an die typische Regel $S \rightarrow NP,\ VP$). Auch in Prolog ist eine solche Schreibweise möglich, hier benutzt man das Zeichen „`-->`“. Dieser Pfeil kann auf zwei Weisen angewendet werden. Hierzu seien $\mathtt{a}_0, \ldots, \mathtt{a}_n$ zweistellige Relationssymbole und $\mathtt{t}_1, \ldots, \mathtt{t}_n$ Prologterme. Dann gilt:

1. Ein Ausdruck der Form $\mathtt{a}_0\texttt{-->}\mathtt{a}_1, \ldots, \mathtt{a}_n.$ steht für
$$\mathtt{a}_0(X_0, X_n) \leftarrow \mathtt{a}_1(\mathtt{X}_0, \mathtt{X}_1), \ldots, \mathtt{a}_n(\mathtt{X}_{n-1}, \mathtt{X}_n).$$
2. Ein Ausdruck der Form $\mathtt{a}_0\texttt{-->}[\mathtt{t}_1, \ldots, \mathtt{t}_n].$ steht für
$$\mathtt{a}_0([\mathtt{t}_1, \ldots, \mathtt{t}_n|\mathtt{X}], \mathtt{X}).$$

Die Implementierung der Grammatiknotation macht Gebrauch von sogenannten *Differenzlisten*. Bei dieser Technik versieht man ein Prädikat, das eine Liste verarbeiten soll, mit zwei Argumenten. Das erste Argument bildet gewissermaßen eine Eingabe-, das zweite eine Ausgabeliste. Durch das Prädikat wird der Unterschied dieser beiden Listen behandelt. Es werden einige Zeichen der Eingabeliste „konsumiert“, die weiteren Zeichen, der Rest der Eingabeliste, bildet die Ausgabe. Betrachten wir das Programm:

```
a([a|X],X).
b([b|X],X).
ab(X,Z)←a(X,Y),b(Y,Z).
```

Mittels des Prädikats `ab(X,Y)` können wir prüfen, ob `X` eine Liste ist, die aus den Konstanten `a` und `b` und der Liste `Y` zusammengesetzt ist. Eine Resolution hätte sowohl für das Ziel `←ab([a,b],[]).` als auch beispielsweise für `←ab([a,b,c],[c]).` Erfolg.

Es ist möglich, die oben beschriebenen Schreibweisen zu mischen. So steht der Ausdruck $\mathtt{a}_0\texttt{-->}[\mathtt{t}_1], \mathtt{a}_1, \mathtt{a}_2, [\mathtt{t}_2].$ für
$$\mathtt{a}_0([\mathtt{t}_1|\mathtt{X}_0], \mathtt{X}_2) \leftarrow \mathtt{a}_1(\mathtt{X}_0, \mathtt{X}_1), \mathtt{a}_2(\mathtt{X}_1, [\mathtt{t}_2|\mathtt{X}_2]).$$

[4] Vgl. hierzu *Colmerauer und Roussell* [CR96].

Weiterhin lassen sich Grammatiken auch mit höherstelligen Relationen erzeugen. Hierzu schreibt man alle weiteren Argumente einer Relation in Klammern hinter diese. Die Schreibweise

$$\texttt{a(X, Y, Z)-->b(X, Y), c(Z).}$$

bedeutet damit

$$\texttt{a}(\texttt{X}, \texttt{Y}, \texttt{Z}, \texttt{X}_0, \texttt{X}_2) \leftarrow \texttt{b}(\texttt{X}, \texttt{Y}, \texttt{X}_0, \texttt{X}_1), \texttt{c}(\texttt{Z}, \texttt{X}_1, \texttt{X}_2).$$

Außerdem können an beliebiger Stelle beliebig viele Atome zum Körper der Klausel hinzugefügt werden. Diese schreibt man in geschweifte Klammern.

$$\texttt{a-->\{z(X)\}, b, c(X).}$$

steht für

$$\texttt{a}(\texttt{X}_0, \texttt{X}_1) \leftarrow \texttt{z}(\texttt{X}), \texttt{b}(\texttt{X}_0, \texttt{X}_1), \texttt{c}(\texttt{X}, \texttt{X}_1, \texttt{X}_2).$$

Im Programmcode des Naproche-Systems wird die Grammatikschreibweise benutzt. Beziehen wir uns auf entsprechende Stellen des Programmcodes, benutzen wir stets die alternative Schreibweise. Unter *reinem Prolog* verstehen wir im Folgenden die in diesem Kapitel beschriebene Teilmenge von Prolog.

5 Korrektheit von Prolog-Programmen

Das vorherige Kapitel behandelt reines Prolog als maschinelle Umsetzung logischer Programmierung. Aus technischen Gründen konnte die Theorie der LD-Resolution im Ableitungsmechanismus von Prolog nicht vollständig umgesetzt werden. Dies hat Auswirkungen auf dessen Korrektheit und Vollständigkeit. In diesem Zusammenhang haben wir zwei Probleme kennengelernt, das Occur-Check-Problem und die Möglichkeit, dass der Prolog-Baum eines definiten Programmes und eines definiten Zieles einen unendlichen Zweig besitzt und so existierende Lösungen nicht gefunden werden können.
Dieses Kapitel zeigt Wege auf, die beschriebenen Komplikationen für bestimmte Programme und Ziele auszuschließen. Weiterhin behandeln wir die Bedeutung dieser Programme und die durch die Berechnung gefundenen Lösungen. Auf diese Weise lässt sich überprüfen, ob ein Programm genau den Zweck erfüllt, für den es vorgesehen ist.
Wir gehen hierbei im Wesentlichen nach *K. R. Apt* [Apt97, Apt95] vor.

5.1 Multimengen

Als Grundlage für die weiteren Abschnitte dieses Kapitels geben wir eine Einführung in die Theorie von Multimengen. Hierbei handelt es sich um Mengen, die einzelne Elemente mehrfach enthalten können. Auf Multimengen definieren wir eine Ordnung. Diese bildet ein wichtiges Hilfsmittel, um die Terminierung von Prolog-Programmen zu zeigen.
Das Konzept der Multimenge wird schon von *R. Dedekind* in seinem erstmals 1888 erschienenen Buch „Was sind und was sollen die Zahlen?“ [Ded61, S.47] verwendet. Die Idee eine Ordnung auf Multimengen zu definieren und diese für Terminierungsbeweise zu benutzen stammt von *Derschowitz und Manna* [DM79].[1]
Nach dem letzt genannten Artikel gehen wir in diesem Abschnitt vor und setzen für partielle Ordnungen und ihre Eigenschaften die Definitionen von *Jech* [Jec03, S.17ff] voraus.

Definition 5.1 (Multimenge) *Sei S eine Menge. Eine* Multimenge *über S ist eine Funktion $X : S \to \mathbb{N}$. Die Elemente von $D(X) := \{x \in S | X(x) \neq 0\}$ nennt man auch* Elemente *von X. Für $x \in D(X)$ heißt die Zahl $X(x)$* Vielfachheit *von x in X.*

[1] Mehr Information zur Geschichte der Multimenge gibt *Blizard* in [Bli91].

Man kann jede Teilmenge $X \subseteq S$ als Multimenge auffassen. Hierzu verwendet man die charakteristische Funktion von X, also die Abbildung $\chi_X : S \to \{0, 1\}$, für die gilt:

$$\chi_X(x) = \begin{cases} 1 & \text{wenn } x \in X \\ 0 & \text{sonst} \end{cases}$$

Weil Multimengen Funktionen sind, kann man Teilmultimengen als kleinere Funktionen verstehen.

Definition 5.2 (Teilmultimenge) *Seien X und Y Multimengen über der Menge S. Wir nennen X eine* Teilmultimenge *von Y und schreiben $X \subseteq Y$, wenn die Ungleichung $X(x) \leq Y(x)$ für jedes $x \in S$ gilt.*

Ebenso lassen sich Summe und Differenz von Multimengen definieren:

Definition 5.3 *Seien X und Y Multimengen über der Menge S. Die Summe von X und Y ist die Funktion $X + Y : S \to \mathbb{N}$, für die gilt:*

$$(X + Y)(x) := X(x) + Y(x).$$

Die Differenz von X und Y ist die Funktion $X - Y : S \to \mathbb{Z}$, für die gilt:

$$(X - Y)(x) := X(x) - Y(x).$$

Die Addition und Subtraktion von Multimengen genügt den für Funktionen üblichen Gesetzmäßigkeiten und kann damit genauso verwendet werden. Beispielsweise steht für X, Y, Z Multimengen über S der Ausdruck $X - Y + Z$ für die Funktion $(X - Y + Z)(x) := X(x) - Y(x) + Z(x)$. Man beachte bei den Festlegungen dieser Definition, dass zwar die Summe zweier Multimengen eine Multimenge ist, dies bei ihrer Differenz jedoch nicht immer der Fall ist.

Im Folgenden beschränken wir unsere Ausführungen auf endliche Multimengen:

Definition 5.4 (Endliche Multimenge) *Eine Multimenge X über S heißt* endlich, *wenn $D(X)$ endlich ist. In diesem Fall schreiben wir X auch als* $\text{bag}(x_1, \ldots, x_n)$, *wobei $x_1, \ldots, x_n$ eine ungeordnete Aufzählung der Elemente von X in entsprechender Vielfachheit ist. Die Menge aller endlichen Multimengen über S bezeichnen wir mit* $\text{Mul}(S)$.

Beispiele für endliche Multimengen über $\mathbb{N}$ sind $\text{bag}(9)$, $\text{bag}(1, 2, 3, 3, 3, 4, 7, 7)$ und $\text{bag}(1, 2, 1)$. Auf den endlichen Multimengen über einer partiell geordneten Menge definieren wir folgende Ordnung:

Definition 5.5 (Multimengenordnung) *Sei $(S, \prec)$ eine strikt partiell geordnete Menge. Für $M, N \in \mathrm{Mul}(S)$ gelte $M \prec_m N$, wenn Multimengen $X, Y \in \mathrm{Mul}(S)$, $D(Y) \neq \emptyset$ und $Y \subseteq N$, existieren, so dass gilt:*

(i) Für alle $x \in S$: $M(x) = N(x) - Y(x) + X(x)$

(ii) Für jedes $x \in D(X)$ gibt es ein $y \in D(Y)$ mit $x \prec y$.

Die Ordnung $(\mathrm{Mul}(S), \prec_m)$ nennen wir die Multimengenordnung *über $(S, \prec)$.*

Anschaulich bedeutet die Definition, dass man aus einer Multimenge N über S eine bzgl. $\prec_m$ kleinere Multimenge M erhält, indem man in N eine Teilmultimenge Y durch eine Multimenge X ersetzt, in der jedes Element kleiner bzgl. $\prec$ als ein Element aus Y ist.

Beispiel Sei $(\mathrm{Mul}(\mathbb{N}), \prec_m)$ die Multimengenordnung über $(\mathbb{N}, <)$. Wir betrachten die Multimengen $\mathrm{bag}(1, 2, 3, 3, 7, 9)$ und $\mathrm{bag}(1, 8)$. Es gilt:

$$\mathrm{bag}(1, 8) \prec_m \mathrm{bag}(1, 2, 3, 3, 7, 9)$$

Denn man erhält die Menge $\mathrm{bag}(1, 8)$ aus $\mathrm{bag}(1, 2, 3, 3, 7, 9)$, indem man die Teilmultimenge $Y := \mathrm{bag}(2, 3, 3, 7, 9) \subseteq \mathrm{bag}(1, 2, 3, 3, 7, 9)$ durch $X := \mathrm{bag}(8)$ ersetzt. Für $8 \in D(X)$ gilt $8 < 9$ und $9 \in D(Y)$.

Wir beweisen einige Eigenschaften von Multimengenordnungen:

Lemma 5.6 *Die Multimengenordnung ist eine strikte partielle Ordnung.*

Beweis Sei $(\mathrm{Mul}(S), \prec_m)$ die Multimengenordnung über $(S, \prec)$. Wir zeigen:

1. *Irreflexivität*: Angenommen für $M \in \mathrm{Mul}(S)$ gelte $M \prec_m M$. Dann existiert eine Multimenge $X \subseteq M$, $D(X) \neq \emptyset$, so dass:

   $$\forall x \in D(X) \exists y \in D(X) : x \prec y$$

   Es existiert also für jedes Element von X ein weiteres Element in X, das größer als dieses bezüglich der partiellen Ordnung ist. Weil X eine endliche Multimenge ist, führt dies zu einem Widerspruch.
2. *Transitivität*: Für die Multimengen $L, M, N \in \mathrm{Mul}(S)$ gelte $L \prec_m M$ und $M \prec_m N$. Nach Definition gilt also:
   a) Es existieren $X_M, Y_M \in \mathrm{Mul}(S)$, $D(Y_M) \neq \emptyset$, $Y_M \subseteq M$, so dass gilt:
      (i) Für jedes $x \in S$: $L(x) = M(x) - Y_M(x) + X_M(x)$
      (ii) Für jedes $x \in D(X_M)$ existiert ein $y \in D(Y_M)$ mit $x \prec y$.
   b) Es existieren $X_N, Y_N \in \mathrm{Mul}(S)$, $D(Y_N) \neq \emptyset$, $Y_N \subseteq N$, so dass gilt:
      (i) Für jedes $x \in S$: $M(x) = N(x) - Y_N(x) + X_N(x)$

(ii) Für jedes $x \in D(X_N)$ existiert ein $y \in D(Y_N)$ mit $x \prec y$.

Aus a)(i)und b)(i) ergibt sich die Gleichung:

$$L(x) = N(x) - Y_N(x) + X_N(x) - Y_M(x) + X_M(x)$$

Wir definieren die Multimengen X und Y als die folgenden Funktionen:

$$X(x) := X_M(x) + \max\{0, X_N(x) - Y_M(x)\}$$

$$Y(x) := Y_N(x) + \max\{0, Y_M(x) - X_N(x)\}$$

Wegen

$$\begin{array}{ll} & \max\{0, X_N(x) - Y_M(x)\} - \max\{0, Y_N(x) - X_N(x)\} \\ = & \max\{0, X_N(x) - Y_M(x)\} + \min\{0, X_N(x) - Y_M(x)\} \\ = & X_N(x) - Y_M(x) \end{array}$$

gilt:

$$L(x) = N(x) - Y(x) + X(x)$$

Außerdem gilt $D(Y) \supseteq D(Y_N) \neq \emptyset$.
Um $Y \subseteq N$ zu zeigen, betrachten wir die Vielfachheit von $x \in D(Y)$. Hier können zwei Fälle auftreten:

Fall 1: Ist $\max\{0, Y_M(x) - X_N(x)\} = 0$, so gilt $Y(x) = Y_M(x)$ und wegen $Y_N \subseteq N$ folgt dann $N(x) \geq Y(x)$.

Fall 2: Ist $\max\{0, Y_M(x) - X_N(x)\} \neq 0$, so gilt $Y(x) = Y_M(x) + Y_N(x) - X_N(x)$. Aus $Y_M \subseteq M$ und b)(i) folgt dann:

$$\begin{array}{lll} & N(x) - Y_N(x) + X_N(x) & \geq Y_M(x) \\ \Leftrightarrow & N(x) & \geq Y_M(x) + Y_N(x) - X_N(x) \end{array}$$

Damit gilt $Y \subseteq N$.
Es bleibt Punkt (ii) der Definition zu zeigen. Sei dazu $x \in D(X)$. Wie oben unterscheiden wir zwei Fälle:

Fall 1: $\max\{0, X_N(x) - Y_M(x)\} = 0$. In diesem Fall gilt $x \in D(X_M)$ und nach Voraussetzung existiert ein $y \in D(Y_M)$ mit $x \prec y$. Gilt $y \in D(Y)$, so ist das zu Zeigende erfüllt. Ist $y \notin D(Y)$, so gilt $y \in D(X_N)$. In diesem Fall existiert für y ein $y' \in D(Y_N) \subseteq D(Y)$ mit $y \prec y'$. Wegen der Transitivität von $\prec$ folgt $x \prec y'$.

Fall 2: $\max\{0, X_N(x) - Y_M(x)\} \neq 0$. Dann gilt $x \in D(X_N)$ und nach Voraussetzung existiert ein $y \in D(Y_N) \subseteq D(Y)$ mit $x \prec y$.

Es folgt $L \prec_m N$.

□

Lemma 5.7 *Die Multimengenordnung über einer linearen Ordnung ist linear.*

Beweis Sei $\prec$ eine lineare Ordnung über S und $(\mathrm{Mul}(S), \prec_m)$ die Multimengenordnung über $(S, \prec)$. Seien $N, M \in \mathrm{Mul}(S)$ und $N \neq M$. Wir betrachten den bzgl. $\prec$ maximalen Wert, für den sich die Multimengen unterscheiden:

$$x_0 := \max\{x \in S | M(x) - N(x) \neq 0\}$$

Gilt $M(x_0) - N(x_0) = -n < 0$, so betrachten wir die Multimengen:

$$Y := (N \upharpoonright \{x \in S | x \prec x_0\}) \cup \{(x_0, n)\} \text{ und } X := M \upharpoonright \{x \in S | x \prec x_0\}.$$

Es gilt $x_0 \in D(Y) \neq \emptyset$, $Y \subseteq N$ und $M(x) = N(x) - Y(x) + X(x)$ für alle $x \in S$. Weil außerdem für jedes Element $y \in D(Y)$ nach Definition $y \prec x_0 \in D(X)$ gilt, folgt $M \prec_m N$.
Für den Fall $M(x_0) - N(x_0) = n > 0$ erfolgt der Beweis für $N \prec_m M$ analog. Die Multimengenordnung $\prec_m$ ist linear. □

Der Beweis des vorherigen Lemmas verdeutlicht, dass die Multimengenordnung $(\mathrm{Mul}(S), \prec_m)$ über einer linear geordneten Menge $(S, \prec)$ der folgenden Bedingung genügt:

Korollar 5.8 (Multimengenordnung über einer linearen Ordnung) *Sei $(S, \prec)$ eine linear geordnete Menge. Seien $M, N \in \mathrm{Mul}(S)$. Dann gilt $M \prec_m N$ genau dann, wenn ein $x \in S$ existiert, so dass $M(x) < N(x)$ gilt und für alle $y \in S$ mit $x \prec y$ die Bedingung $M(y) = N(y)$ erfüllt ist.*

Wir zeigen eine weitere Eigenschaft der Multimengenordnung, die für die Beweise des nächsten Abschnitts von besonderer Bedeutung ist. Hierzu benutzen wir folgendes bekannte Lemma[2]:

Lemma 5.9 (König) *Ein unendlicher Baum, der sich endlich verzweigt, besitzt einen unendlichen Zweig.*

Nun zeigen wir:

Lemma 5.10 *Die Multimengenordnung über einer fundierten Ordnung ist fundiert.*

Beweis Sei S eine Menge und $\prec$ eine fundierte partielle Ordnung auf S. Wir erweitern die Menge S auf die Menge $S' = S \cup \{e\}$, $e \notin S$. Für jedes $x \in S$ gelte $e \prec x$. Die partielle

[2]Einen Beweis findet man z.B. in [Doe94, S.5].

Ordnung $(S', \prec)$ ist ebenfalls fundiert. Sei nun $(\text{Mul}(S), \prec_m)$ die Multimengenordnung über $(S, \prec)$. Wir nehmen an, dass eine unendliche absteigende Folge $A_1 \succ_m A_2 \succ_m \ldots$ aus Elementen von $\text{Mul}(S)$ existiert. Aus dieser konstruieren wir nun einen Baum.
Die Elemente von A_1 in entsprechender Vielfachheit bilden die Wurzel des Baumes. Wegen $A_2 \prec_m A_1$ existieren Multimengen $X, Y \in \text{Mul}(S)$, $Y \subseteq A_1$, $D(Y) \neq \emptyset$, so dass für jedes $x \in S$ die Eigenschaft $A_2(x) = A_1(x) - Y(x) + X(x)$ gilt und für alle $x \in D(X)$ ein $y \in D(Y)$ mit $x \prec y$ existiert. Im Baum versehen wir für jedes $y \in D(Y)$, $Y(y) = n$, n viele Knoten y mit dem Tochterknoten e. Weiterhin fügen wir für alle $x \in D(X)$, $X(x) = m$, m Tochterknoten x an einem der zugehörigen $y \in D(Y)$ mit $x \prec y$ hinzu. Alle Blätter des Baumes bilden nun die Multimenge A_2 erweitert um das n-fache Element e. Im nächsten Schritt vergrößern wir den Baum, indem wir mit Hilfe von $A_3 \prec_m A_2$ auf die beschriebene Weise Tochterknoten für die Blätter des Baumes hinzufügen. Es ergibt sich ein endlich verzweigender Baum, der wegen der Unendlichkeit von $A_1 \succ_m A_2 \succ_m \ldots$ ebenfalls unendlich ist. Nach dem Lemma von König existiert dann aber auch ein unendlicher Zweig in diesem Baum, also eine unendliche absteigende Folge $a_1 \succ a_2 \succ \ldots$ von Elementen aus S'. Dies widerspricht der Fundiertheit von $(S', \prec)$. □

Aus dem oben Gezeigten folgt:

Korollar 5.11 *Die Multimengenordnung über einer Wohlordnung ist eine Wohlordnung.*

5.2 Terminierung

In diesem Abschnitt lernen wir Eigenschaften von definiten Programmen und Zielen kennen, die garantieren, dass jede LD-Ableitung dieser und damit auch ihre Prolog-Ableitungen terminieren. Hierzu verwenden wir die Multimengenordnung $(\text{Mul}(\mathbb{N}), \prec_m)$ über $(\mathbb{N}, <)$. Die Idee unseres Vorgehens ist es, mit jedem definiten Ziel G_i, das wir durch einen Resolutionsschritt innerhalb einer LD-Ableitung erhalten, eine Multimenge zu verbinden. Können wir zeigen, dass die erhaltenen Multimengen eine absteigende Folge bilden, so folgt mit Hilfe der Fundiertheit von $(\mathbb{N}, <)$ die Endlichkeit der Ableitung. Wir beziehen uns, neben den erwähnten Werken, auf einen Artikel von *Apt* und *Pedreschi* [AP93], der einen Ansatz von *Bezem* [Bez93] für beliebige Auswahlregeln auf die left most selection rule überträgt und erweitert.

Wir beginnen mit der Definition einer Abbildung, die jedem Grundatom eines Programmes eine natürliche Zahl zuordnet:

Definition 5.12 (Niveau-Abbildung) *Eine* Niveau-Abbildung *für ein Programm P ist eine Funktion $|\ | : B_P \to \mathbb{N}$ von Grundatomen in die natürlichen Zahlen. Für $A \in B_P$ ist $|A|$ das* Niveau *von A.*

Mit dieser Abbildung und der folgenden Definition können wir nun definite Ziele mit Multimengen natürlicher Zahlen verbinden:

Definition 5.13 (Beschränktheit) *Sei P ein Programm, $|\ |$ eine Niveau-Abbildung für P, $\mathfrak{I}$ eine Interpretation von P und k eine natürliche Zahl.*

- *Ein Ziel G heißt* beschränkt durch k *bzgl. $|\ |$ und $\mathfrak{I}$, wenn für jede seiner Grundinstanzen $\leftarrow A_1, \ldots, A_i, \ldots, A_n$ mit $\mathfrak{I} \models A_1, \ldots, A_{i-1}$ gilt:*

$$|A_i| \leq k.$$

  Ein Ziel G heißt beschränkt bzgl. $|\ |$ und $\mathfrak{I}$, wenn es beschränkt durch ein k bzgl. $|\ |$ und $\mathfrak{I}$ ist.

- *Mit jedem Ziel G, bestehend aus n Atomen, verbinden wir n Mengen natürlicher Zahlen, die für $i \in [1, n]$ folgendermaßen definiert sind:*

$$|G|_i^{\mathfrak{I}} := \{|A_i| \mid \leftarrow A_1 \ldots A_n \text{ ist eine Grundinstanz von } G \text{ und } \mathfrak{I} \models A_1, \ldots, A_{i-1}\}.$$

- *Mit einem bzgl. $|\ |$ und $\mathfrak{I}$ beschränkten Ziel G, das aus n Atomen besteht, verbinden wir folgende Multimenge $|G|_{\mathfrak{I}}$ natürlicher Zahlen:*

$$|G|_{\mathfrak{I}} := \mathrm{bag}(\max |G|_1^{\mathfrak{I}}, \ldots, \max |G|_n^{\mathfrak{I}}).$$

Der Begriff der Beschränktheit eines definiten Zieles, wie er in dieser Definition eingeführt wird, meint die Beschränktheit seiner Atome für den Fall, dass diese innerhalb einer LD-Resolution aufgerufen werden. Wegen der Verwendung der left most selection rule wurde, wenn eine Instanz des Atoms A_i eines definiten Zieles $\leftarrow A_1, \ldots, A_n$ in einem LD-Ableitungsprozess ausgewählt wird, bereits eine berechnete Antwort für die Atome $A_1, \ldots, A_{i-1}$ gefunden und auf A_i angewendet. Weil nach Satz 3.37 jede berechnete Antwort auch eine korrekte Antwort ist, sind damit alle Terme, die A_i mit einem der A_j, $j < i$, gemeinsam hat, bereits so substituiert, dass sie der Definition des in A_j verwendeten Relationssymbols genügen. Die daraus resultierenden Eigenschaften dieser Terme können deshalb in A_i vorausgesetzt werden. Dies erklärt die Verwendung der Bedingung $\mathfrak{I} \models A_1, \ldots, A_{i-1}$.
Dieses Prinzip wenden wir auch für die folgende Programmeigenschaft an:

Definition 5.14 (Akzeptabilität) *Sei P ein Programm, $|\ |$ eine Niveau-Abbildung und $\mathfrak{I}$ eine Interpretation von P.*

- *Eine Klausel von P heißt akzeptabel bezüglich $|\ |$ und $\mathfrak{I}$, wenn $\mathfrak{I}$ ein Modell von ihr ist und für jede ihrer Grundinstanzen $A \leftarrow B_1, \ldots, B_i, \ldots, B_n$ aus $\mathfrak{I} \models B_1, \ldots, B_{i-1}$ folgt:*

$$|A| > |B_i|$$

- *Ein Programm P heißt* akzeptabel in Bezug auf $|\ |$ und $\mathfrak{I}$*, wenn jede seiner Klauseln es ist. P heißt* akzeptabel *wenn es akzeptabel bezüglich einer Niveau-Abbildung und einer Interpretation von P ist.*

Wie in der vorherigen Definition werden auch bei der Akzeptabilität einer Klausel ihr Kopf A und ein Atom B_i ihres Körpers in der Situation betrachtet, in der eine Instanz des Atoms B_i innerhalb einer LD-Resolution ausgewählt wird. Ebenso wie oben können wir an der betreffenden Stelle $\mathfrak{I} \models B_1, \ldots, B_{i-1}$ voraussetzen.
Nun lässt sich zeigen, dass die Eigenschaft eines Programmes P, akzeptabel zu sein, für jedes beschränkte definite Ziel G garantiert, dass die Folge der Multimengen, die wir durch die obige Definition mit den Resolventen einer LD-Ableitung von $P \cup \{G\}$ verbinden, absteigend ist:

Lemma 5.15 *Sei P ein bzgl. einer Niveau-Abbildung $|\ |$ und einer Interpretation $\mathfrak{I}$ akzeptables Programm, G_1 ein bzgl. $|\ |$ und $\mathfrak{I}$ beschränktes Ziel und G_2 ein LD-Resolvent von G_1 und einer Klausel von P. Dann gilt:*

(i) G_2 *ist beschränkt bzgl.* $|\ |$ *und* $\mathfrak{I}$*,*

(ii) $|G_2|_{\mathfrak{I}} \prec_m |G_1|_{\mathfrak{I}}$

Beweis Aus einem definiten Ziel und einer Programmklausel erhalten wir einen LD-Resolventen, indem wir zunächst die Variablen der Klausel so umbenennen, dass Klausel und Ziel keine Variablen gemeinsam haben. Danach wird das erste Atom des definiten Zieles mit dem Kopf der umbenannten Klausel unifiziert und durch den Körper der sich ergebenden Instanz der Klausel ersetzt. Deswegen genügt es, bzgl. $|\ |$ und $\mathfrak{I}$ folgende Punkte zu beweisen:

1. Jede Instanz G' eines beschränkten Zieles G ist beschränkt und es gilt $|G'|_{\mathfrak{I}} \preceq_m |G|_{\mathfrak{I}}$:
   Beweis: Für jedes $i \in [1, n]$, n Anzahl der Atome von G, gilt $|G'|_i^{\mathfrak{I}} \subseteq |G|_i^{\mathfrak{I}}$.
2. Jede Instanz einer akzeptablen Klausel ist akzeptabel:
   Beweis: Klar.
3. Für jede akzeptable Klausel $A \leftarrow B_1, \ldots, B_n$ und jede Folge von Atomen $C_1, \ldots, C_m$ ist, wenn $\leftarrow A, C_1, \ldots, C_m$ beschränkt ist, auch $\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m$ beschränkt und es gilt

   $$|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_{\mathfrak{I}} \prec_m |\leftarrow A, C_1, \ldots, C_m|_{\mathfrak{I}}$$

   Beweis: Wir gehen in zwei Schritten vor.

   (i) Für $i \in [1, n]$ ist $|\leftarrow B_1, \ldots, B_n, C_1 \ldots, C_m|_i^{\mathfrak{I}}$ endlich und es gilt

   $$\max|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_i^{\mathfrak{I}} < \max|\leftarrow A, C_1, \ldots, C_m|_1^{\mathfrak{I}}$$

Beweis:

$$
\begin{array}{ll}
 & \max|\leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m|_i^{\mathfrak{I}} \\
\overset{\text{Def.5.13}}{=} & \max\{|B_i'| \mid \leftarrow B_1',\ldots,B_n' \text{ ist eine Grundinstanz von } \leftarrow B_1,\ldots,B_n \\
 & \qquad \text{und } \mathfrak{I} \models B_1',\ldots,B_{i-1}'\} \\
\overset{(*)}{=} & \max\{|B_i'| \mid A' \leftarrow B_1',\ldots,B_n' \text{ ist eine Grundinstanz von} \\
 & \qquad A \leftarrow B_1,\ldots,B_n \text{ und } \mathfrak{I} \models B_1',\ldots,B_{i-1}'\} \\
\overset{\text{Def.5.14}}{<} & \max\{|A'| \mid A' \text{ ist eine Grundinstanz von } A\} \\
\overset{\text{Def.5.13}}{=} & \max|\leftarrow A,C_1,\ldots,C_m|_1^{\mathfrak{I}}
\end{array}
$$

(∗) Diese Gleichheit gilt, weil ein A' existiert, für das $A' \leftarrow B_1',\ldots,B_n'$ eine Grundinstanz von $A \leftarrow B_1,\ldots,B_n$ ist.

(ii) Für $j \in [1,m]$ ist $|\leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m|_{j+n}^{\mathfrak{I}}$ endlich und es gilt

$$\max|\leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m|_{n+j}^{\mathfrak{I}} \leq \max|\leftarrow A,C_1,\ldots,C_m|_{1+j}^{\mathfrak{I}}$$

Beweis:

$$
\begin{array}{ll}
 & \max|\leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m|_{n+j}^{\mathfrak{I}} \\
\overset{\text{Def.5.13}}{=} & \max\{|C_j'| \mid \leftarrow B_1',\ldots,B_n',C_1',\ldots,C_m' \text{ ist eine Grundinstanz von} \\
 & \qquad \leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m \text{ und } \mathfrak{I} \models B_1',\ldots,B_n',C_1',\ldots,C_{j-1}'\} \\
\overset{(**)}{\leq} & \max\{|C_j'| \mid \leftarrow A',C_1',\ldots,C_m' \text{ ist eine Grundinstanz von} \\
 & \qquad \leftarrow A,C_1,\ldots,C_m \text{ und } \mathfrak{I} \models A',C_1',\ldots,C_{j-1}'\} \\
\overset{\text{Def.5.13}}{=} & \max|\leftarrow A,C_1,\ldots,C_m|_{1+j}^{\mathfrak{I}}
\end{array}
$$

(∗∗) Es existiert eine Grundinstanz A' von A, so dass $A' \leftarrow B_1',\ldots,B_n'$ eine Grundinstanz der Programmklausel $A \leftarrow B_1,\ldots,B_n$ ist. Weil $\mathfrak{I}$ ein Modell von P ist, folgt aus $\mathfrak{I} \models B_1',\ldots,B_n',C_1,\ldots,C_{j-1}$ damit $\mathfrak{I} \models A'$. Also gilt die Inklusion:

$$
\begin{array}{ll}
 & \{|C_j'| \mid \leftarrow B_1',\ldots,B_n',C_1',\ldots,C_m' \text{ ist eine Grundinstanz von} \\
 & \qquad \leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m \text{ und } \mathfrak{I} \models B_1',\ldots,B_n',C_1',\ldots,C_{j-1}'\} \\
\subseteq & \{|C_j'| \mid \leftarrow A',C_1',\ldots,C_m' \text{ ist eine Grundinstanz von} \\
 & \qquad \leftarrow A,C_1,\ldots,C_m \text{ und } \mathfrak{I} \models A',C_1',\ldots,C_{j-1}'\}
\end{array}
$$

Damit gilt für $Z_1 :=\leftarrow A,C_1,\ldots,C_m$ und $Z_2 :=\leftarrow B_1,\ldots,B_n,C_1,\ldots,C_m$:

$$\text{bag}(\max|Z_2|_1^{\mathfrak{I}},\ldots\max|Z_2|_{n+m}^{\mathfrak{I}}) \prec_m \text{bag}(\max|Z_1|_1^{\mathfrak{I}},\ldots\max|Z_1|_{m+1}^{\mathfrak{I}})$$

Es folgt die Beschränktheit von Z_2. Und damit das zu Zeigende. □

Damit gilt nun:

Korollar 5.16 *Sei P ein akzeptables Programm und G ein beschränktes definites Ziel. Dann sind alle LD-Ableitungen von $P \cup \{G\}$ endlich.*

Beweis Die Aussage folgt aus Lemma 5.15 und Lemma 5.10. □

Es wurde gezeigt, dass für akzeptable Programme und beschränkte Ziele alle LD-Ableitungen endlich sind. Man kann also durch den Nachweis dieser Eigenschaften garantieren, dass das anfangs beschriebene Problem eines möglichen unendlichen Zweiges im Prolog-Baum für die jeweiligen Ziele und Programme nicht auftritt.
Um den Nachweis von Akzeptabilität zu vereinfachen, betrachten wir eine in vielen Fällen leichter zu beweisende Eigenschaft, Semi-Akzeptabilität. Hierzu sei $\mathrm{rel}(P)$ die Menge der im definiten Programm P vorkommenden Relationssymbole, entsprechend sei $\mathrm{rel}(A)$ das in dem Atom A vorkommende Relationssymbol. Wir betrachten die Beziehungen der Relationssymbole eines Programmes:

Definition 5.17 *Sei P ein Programm und $p, q \in \mathrm{rel}(P)$.*

- *Wir sagen p* verweist auf *q, wenn eine Klausel in P existiert, in deren Kopf p und in deren Körper q vorkommt.*
- *Wir sagen p* hängt in P von q ab *und schreiben dies als $p \sqsupseteq q$, wenn (p, q) im transitiven reflexiven Abschluss der Relation „*verweist auf*“ ist.*
- *Wir nennen p und q* voneinander abhängig *und schreiben $p \simeq q$, wenn $p \sqsupseteq q$ und $q \sqsupseteq p$ gilt. Insbesondere sind p und p gegenseitig rekursiv.*

Gilt $p \sqsupseteq q$ und $q \not\sqsupseteq p$, so schreiben wir $p \sqsupset q$. Kommt p im Kopf einer Klausel und q in ihrem Körper vor, so gilt entweder $p \sqsupset q$ oder $p \simeq q$. Damit definieren wir:

Definition 5.18 (Semi-Akzeptabilität) *Sei P ein Programm, $|\ |$ eine Niveau-Abbildung für P und $\mathfrak{I}$ eine Interpretation von P.*

- *Eine Klausel heißt* semi-akzeptabel bezüglich $|\ |$ und $\mathfrak{I}$, *wenn $\mathfrak{I}$ ein Modell von ihr ist und für jede Grund-Instanz $A \leftarrow B_1, \ldots, B_n$ aus $\mathfrak{I} \models B_1, \ldots, B_{i-1}$ folgt*

  (i) $|A| > |B_i|$ *wenn* $\mathrm{rel}(A) \simeq \mathrm{rel}(B_i)$,
  (ii) $|A| \geq |B_i|$ *wenn* $\mathrm{rel}(A) \sqsupset \mathrm{rel}(B_i)$.
- *Ein Programm heißt* semi-akzeptabel bzgl. $|\ |$ und $\mathfrak{I}$, *wenn jede seiner Klauseln es ist. P heißt* semi-akzeptabel, *wenn es semi-akzeptabel bzgl. einer Niveau-Abbildung und einer Interpretation von P ist.*

Man sieht leicht, dass jedes akzeptable Programm auch semi-akzeptabel ist. Wir zeigen die umgekehrte Richtung:

Lemma 5.19 *Ist ein Programm P semi-akzeptabel bzgl. einer Niveau-Abbildung $|\ |$ und einer Interpretation $\mathfrak{I}$, so ist P akzeptabel bezüglich einer Niveau-Abbildung $\|\ \|$ und $\mathfrak{I}$. Weiterhin ist jedes bzgl. $|\ |$ beschränkte Atom A ebenfalls bzgl. $\|\ \|$ beschränkt.*

Beweis Um eine Niveau-Abbildung $\|\ \|$ zu definieren, führen wir zunächst eine Abbildung von den Relationssymbolen von P in die natürlichen Zahlen ein, so dass für $p, q \in \mathrm{rel}(P)$ gilt:

(i) $p \simeq q \Rightarrow |p| = |q|$

(ii) $p \sqsupset q \Rightarrow |p| > |q|$

Nun sei die Niveau-Abbildung $\|\ \|$ für P folgendermaßen definiert:

$$\|A\| := |A| + |\,\mathrm{rel}(A)|$$

für alle $A \in B_P$.
Wir zeigen, dass $\|\ \|$ akzeptabel ist. Hierzu sei $A \leftarrow B_1, \ldots, B_n$ die Grundinstanz einer Klausel aus P und für ein $i \in [1, n]$ gelte $\mathfrak{I} \models B_1, \ldots, B_{i-1}$. Wir unterscheiden zwei Fälle:

a) Gilt $\mathrm{rel}(A) \simeq \mathrm{rel}(B_i)$, so folgt mittels Definition 5.18:

$$\|A\| = |A| + |\,\mathrm{rel}(A)| \overset{5.18(i)}{>} |B_i| + |\,\mathrm{rel}(A)| \overset{(i)}{=} |B_i| + |\,\mathrm{rel}(B_i)|$$

b) Gilt $\mathrm{rel}(A) \sqsubset \mathrm{rel}(B_i)$, so folgt ebenfalls mittels der Definition:

$$\|A\| = |A| + |\,\mathrm{rel}(A)| \overset{5.18(ii)}{\geq} |B_i| + |\,\mathrm{rel}(A)| \overset{(ii)}{>} |B_i| + |\,\mathrm{rel}(B_i)|$$

In beiden Fällen gilt $\|A\| > \|B_i\|$. Es folgt die Akzeptabilität jeder Klausel aus P, damit die Akzeptabilität von P bzgl. $\|\ \|$ und $\mathfrak{I}$.
Sei das Atom A bzgl. $|\ |$ durch $k \in \mathbb{N}$ beschränkt. Die oben beschriebene Abbildung bildet $\mathrm{rel}(A)$ auf eine natürliche Zahl l ab. Damit ist $\|A\|$ durch $k + l$ beschränkt. □

Als Beweis für die Endlichkeit aller LD-Ableitungen eines Zieles und eines Programmes genügt es also, Beschränktheit und Semi-Akzeptabilität zu zeigen. Dieses Prinzip wenden wir in Abschnitt 6.4 an.

5.3 Occur-Check-Problem

In Abschnitt 4.2 wurde die Unifikation von Prolog erklärt. Prolog verwendet den in Definition 3.26 vorgestellten Unifikationsalgorithmus von *Martelli* und *Montanari* mit dem Unterschied, dass es auf den Occur-Check verzichtet und Operation 6 der Definition

weglässt. Wir haben bereits gesehen, dass dieses Vorgehen Einfluss auf die Korrektheit der Resolution haben kann. An dieser Stelle stellen wir syntaktische Eigenschaften von Programmen und Zielen vor, die garantieren, dass in ihrer Prolog-Resolution das Occur-Check-Problem nicht auftritt. Hierzu untersuchen wir die Beschaffenheit der zu unifizierenden Terme. Sind diese von der Art, dass auch im Martelli-Montanari-Algorithmus Operation 6 nicht angewendet werden kann, bildet das Weglassen des Occur-Checks für das Ergebnis der Unifikation keinen Unterschied.
Im Folgenden bezeichnen wir für die Atome $A := p(s_1, \ldots, s_n)$ und $H := p(t_1, \ldots, t_n)$ mit demselben Relationssymbol durch den Ausdruck $A = H$ die Gleichungsmenge $\{s_1 = t_1, \ldots, s_n = t_n\}$. Damit führen wir folgende Begriffe ein:

Definition 5.20 (NSTO) *Eine Menge von Gleichungen E heißt* not subject to occur check *(NSTO), wenn in keiner Ausführung des Martelli-Montanari-Algorithmus, die mit E startet, Operation (6) angewendet werden kann.*

Definition 5.21 (Occur-Check-Freiheit) *Sei P ein definites Programm.*

- *Sei ξ eine LD-Ableitung für P und A ein ausgewähltes Atom in ξ. Sei H eine Variante des Kopfes einer Klausel von P, so dass A und H das gleiche Relationssymbol enthalten. Dann nennt man die Gleichungsmenge $A = H$* verfügbar *in ξ.*
- *Sei die Menge aller in den LD-Ableitungen von $P \cup \{G\}$ verfügbaren Gleichungen NSTO. Dann nennt man $P \cup \{G\}$* occur-check-frei.

Wir zeigen zunächst ein Kriterium für die Occur-Check-Freiheit einer Gleichungsmenge. Hierzu betrachten wir jede Gleichung in einer bestimmten Orientierung, d.h. wir unterscheiden zwischen rechter und linker Seite des Gleichheitszeichens.

Definition 5.22 (Linke und rechte Seite einer Gleichung) *Die Gleichung e sei von der Form $s = t$ und s und t seien Terme. Dann nennen wir s die* linke Seite *und t die* rechte Seite *der Gleichung e. Die Gleichung e' von der Form $t = s$ nennen wir* Umorientierung von *e.*

Damit können wir nun festlegen:

Definition 5.23 (linear, linkslinear) *Wir nennen eine Familie von Termen* linear*, wenn jede Variable höchstens einmal in ihr vorkommt. Wir nennen eine Menge von Gleichungen* linkslinear*, wenn die Familie von Termen, die aus den linken Seiten der Gleichungen bestehen, linear ist.*

Die Eigenschaft einer Termmenge linear zu sein schließt also aus, dass eine Variable zweimal in einem Term vorkommt (wie etwa in $f(x, x)$) und dass zwei Terme die gleiche Variable verwenden.

Definition 5.24 *Sei E eine Menge von Gleichungen. Mit $\rightarrow_E$ bezeichnen wir die folgende auf den Elementen von E definierte Relation:*
$e_1 \rightarrow_E e_2$ gdw. die linke Seite von e_1 und die rechte Seite von e_2 eine Variable gemeinsam haben.

Es gilt also beispielsweise $x = f(y) \rightarrow_E a = x$, wenn E diese Gleichungen enthält. Kommt in einer Gleichung $e \in E$ eine Variable auf beiden Seiten des Gleichheitszeichens vor, so gilt für diese $e \rightarrow_E e$.

Definition 5.25 *Sei r eine Relation auf der Menge M. Wir nennen r* zyklisch*, wenn Elemente $a_1, \ldots, a_n \in M$ existieren, so dass $a_1 r a_2 r \ldots a_n r a_1$ gilt. Andernfalls nennen wir die Relation* azyklisch*.*

Nun können wir beweisen:

Lemma 5.26 (NSTO) *Sei E eine Menge von Gleichungen, die so umorientiert werden kann, dass die resultierende Menge F linkslinear und die Relation $\rightarrow_F$ azyklisch ist. Dann ist E NSTO.*

Beweis Wir nennen eine Menge von Gleichungen *gut*, wenn sie den Voraussetzungen dieses Satzes genügt. Wir zeigen zwei Punkte:

1. Durch jede Anwendung einer Operation des Martelli-Montanari-Algorithmus auf eine Gleichungsmenge, die gut ist, wird diese Eigenschaft bewahrt.

   Beweis: Wir betrachten jede Operation einzeln. Wir schreiben $E \dot\cup \{e\}$ als Abkürzung für $E \cup \{e\}$ und $e \notin E$.

   Operation (1): Ist $E \dot\cup \{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\}$ linkslinear, so ist die Gleichungsmenge $F := E \cup \{s_1 = t_1, \ldots, s_n = t_n\}$ ebenfalls linkslinear.
   Wir betrachten nun einen Pfad $e_1 \rightarrow_F e_2 \rightarrow_F \ldots \rightarrow_F e_m$ für die Relation $\rightarrow_F$. Indem man jede Gleichung der Form $s_i = t_i$ durch die Gleichung $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ ersetzt, erhält man einen Pfad für die Relation $\rightarrow_{E \cup \{f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)\}}$. Damit folgt, wenn $\rightarrow_{E \cup \{f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)\}}$ azyklisch ist, dass dies auch für $\rightarrow_F$ gilt.

   Operation (3): Für alle Gleichungsmengen E mit $x = x \in E$ gilt $x = x \rightarrow_E x = x$. Damit kann eine Gleichungsmenge, die gut ist, die Gleichung $x = x$ nicht enthalten, Operation (3) kann auf eine solche Gleichungsmenge also nicht angewendet werden.

   Operation (4): Nach Definition ist die Gleichungsmenge $E \dot\cup \{x = t\}$ genau dann gut, wenn auch $E \cup \{t = x\}$ es ist, damit wird durch Operation (4) die Güte bewahrt.

Operation (5): Die Variable x trete nicht im Term t auf. Für die Gleichungsmenge E kann dann kein $e \in E$ existieren mit $x = t \to_{E\{x/t\}\cup\{x=t\}} e\{x/t\}$, weil x nicht in $e\{x/t\}$ vorkommt. $x = t$ besitzt also keinen Nachfolger bzgl. $\to_{E\{x/t\}\cup\{x=t\}}$.
Für eine Gleichungsmenge E bezeichne $\mathrm{LS}(E)$ (bzw. $\mathrm{RS}(E)$) die Menge der Variablen, die in den linken (bzw. rechten) Seiten der Gleichungen aus E vorkommen. Die gleiche Notation verwenden wir auch für einzelne Gleichungen $e \in E$. Wir unterscheiden nun zwei Fälle:

(i) $E\dot\cup\{x = t\}$ ist linkslinear und die Relation $\to_{E\cup\{x=t\}}$ ist azyklisch. Dann gilt $x \notin \mathrm{LS}(E)$, damit ist $E\{x/t\} \cup \{x = t\}$ ebenfalls linkslinear.
Um zu zeigen, dass die Relation $\to_{E\{x/t\}\cup\{x=t\}}$ azyklisch ist, betrachten wir zwei Gleichungen $e_1, e_2 \in E$, für die $e_1\{x/t\} \to_{E\{x/t\}\cup\{x=t\}} e_2\{x/t\}$ gilt. Es gibt also eine Variable y, für die $y \in \mathrm{LS}(e_1\{x/t\})$ und $y \in \mathrm{RS}(e_2\{x/t\})$ gilt. Wegen der Linkslinearität von $E \cup \{x = t\}$ gilt jedoch $x \notin \mathrm{LS}(e_1)$ und damit $y \in \mathrm{LS}(e_1)$. Nun können zwei Fälle auftreten:
   a) $y \in \mathrm{RS}(e_2)$. Dann gilt $e_1 \to_{E\cup\{x=t\}} e_2$.
   b) $y \notin \mathrm{RS}(e_2)$. Dann gilt $x \in \mathrm{RS}(e_2)$ und $y \in \mathrm{var}(t)$. Es folgt $e_1 \to_{E\cup\{x=t\}} x = t \to_{E\cup\{x=t\}} e_2$.

   Wir können aus den Beobachtungen schließen, dass, wenn für die Relation $\to_{E\{x/t\}\cup\{x=t\}}$ ein zyklischer Pfad existiert, ein solcher ebenfalls für die Relation $\to_{E\cup\{x=t\}}$ gefunden werden kann. Damit ist $\to_{E\{x/t\}\cup\{x=t\}}$ azyklisch.

(ii) Sei $E\dot\cup\{t = x\}$ linkslinear und die Relation $\to_{E\cup\{t=x\}}$ azyklisch. Dann kommt x höchstens einmal in $\mathrm{LS}(E)$ vor und es gilt $\mathrm{var}(t) \cap \mathrm{LS}(E) = \emptyset$. Damit ist $E\{x/t\}\cup\{x = t\}$ linkslinear, weil x nicht in t vorkommt. Wie oben betrachten wir, um zu zeigen, dass die Relation $\to_{E\{x/t\}\cup\{x=t\}}$ azyklisch ist, zwei Gleichungen $e_1, e_2 \in E$, für die $e_1\{x/t\} \to_{E\{x/t\}\cup\{x=t\}} e_2\{x/t\}$ gilt. Für eine Variable y gilt dann $y \in \mathrm{LS}(e_1\{x/t\})$ und $y \in \mathrm{RS}(e_2\{x/t\})$. Hier unterscheiden wir vier Fälle:
   a) $y \in \mathrm{LS}(e_1)$ und $y \in \mathrm{RS}(e_2)$. Dann gilt $e_1 \to_{E\cup\{t=x\}} e_2$.
   b) $y \in \mathrm{LS}(e_1)$ und $y \notin \mathrm{RS}(e_2)$. Dann gilt $x \in \mathrm{RS}(e_2)$ und $y \in \mathrm{var}(t)$. Wegen der Linkslinearität von $E\cup\{t = x\}$ gilt jedoch $\mathrm{var}(t)\cap\mathrm{LS}(E) = \emptyset$. Damit kann dieser Fall nicht eintreten.
   c) $y \notin \mathrm{LS}(e_1)$ und $y \in \mathrm{RS}(e_2)$. Dann gilt $x \in \mathrm{LS}(e_1)$ und $y \in \mathrm{var}(t)$. Damit gilt $e_1 \to_{E\cup\{t=x\}} t = x \to_{E\cup\{t=x\}} e_2$.
   d) $y \notin \mathrm{LS}(e_1)$ und $y \notin \mathrm{RS}(e_2)$. Dann gilt $x \in \mathrm{LS}(e_1)$ und $x \in \mathrm{RS}(e_2)$ und deswegen $e_1 \to_{E\cup\{t=x\}} e_2$.

   Wie oben können wir an dieser Stelle schließen, dass die Relation $\to_{E\{x/t\}\cup\{x=t\}}$ nur dann azyklisch sein kann, wenn die Relation $\to_{E\cup\{t=x\}}$ es ist. Damit ist $\to_{E\{x/t\}\cup\{x=t\}}$ azyklisch.

Operationen (2) und (6) verändern die Gleichungsmenge nicht und müssen deswegen an dieser Stelle nicht betrachtet werden.

2. Auf eine gute Gleichungsmenge kann Operation (6) des Martelli-Montanari-Algorithmus nicht angewendet werden.

Beweis: Betrachte eine Gleichung der Form $x = t$, für die x in t vorkommt. Dann gilt für jede Gleichungsmenge E sowohl $x = t \rightarrow_{E \cup \{x=t\}} x = t$ als auch $t = x \rightarrow_{E \cup \{x=t\}} t = x$. Damit ist für eine Gleichungsmenge E, auf die Operation (6) angewendet werden kann, die Relation $\rightarrow_F$ für jede Gleichungsmenge F aus umorientierten Gleichungen aus E zyklisch. Damit ist E nicht gut.

Aus 1. und 2. folgt die Aussage des Lemmas. □

Die umgekehrte Richtung dieses Lemmas ist nicht gültig, wie man sich leicht am Beispiel der Gleichungsmenge $E = \{f(x) = g(x)\}$ klar machen kann.

Wir betrachten nun die zu unifizierenden Atome einer Resolution genauer. Hierzu definieren wir den Begriff *mode*. Hierbei handelt es sich um eine Abbildung, die jeder Positon einer Relation entweder das Zeichen „+“ oder „−“ zuordnet.

Definition 5.27 (Mode) *Sei p ein n-stelliges Relationssymbol. Ein* mode *für p ist eine Funktion m_p von $\{1, \ldots, n\}$ in die Menge $\{+, -\}$. Gilt $m_p(i) =$„+“, so nennen wir i eine* Eingabeposition *von p, im Fall $m_p(i) =$„−“ ist i eine* Ausgabeposition *von p. Mit* moding *meinen wir eine Familie von modes für jeweils verschiedene Relationssymbole.*

Wir schreiben die Funktion m_p auch als $p(m_p(1), \ldots, m_p(n))$, so drückt $p(+, -)$ aus, dass die erste Position der zweistelligen Relation p eine Eingabe-, die zweite eine Ausgabeposition ist. Für alle weiteren Betrachtungen gehen wir von folgender Annahme aus:

Annahme: Jede betrachtete Relation besitzt jeweils einen festen mode.

Diese Annahme erlaubt es, von Eingabe- und Ausgabepositionen eines Atoms zu sprechen.
Wie diese Begriffe vermuten lassen, gab das moding ursprünglich an, wie die Relationen verwendet werden sollen, d.h. an welchen Positionen Werte gegeben sind und an welchen sie gesucht werden. Diese Zuordnung ist recht einfach, wenn man davon ausgeht, dass Variablen nur in Ausgabepositionen von Relationen benutzt werden. In der Praxis kommt es jedoch vor, dass die Terme aller Positionen einer Relation in einem definiten Ziel Variablen enthalten. In solchen Situationen ist die Zuordnung von Eingabe- und Ausgabepositionen schwierig und oft wenig intuitiv. Trotz der Wahl der Bezeichnung, verstehen wir im Folgenden ein moding eher als eine Art Ordnung der Terme in den einzelnen Prädikaten.
Unser nächstes Ziel ist es, Lemma 5.26 bzgl. modes zu formulieren. Hierzu benötigen wir folgende Festlegung:

Definition 5.28 (Eingabe-(Ausgabe-)linear, Eingabe-Ausgabe-disjunkt)

- *Ein Atom heißt* Eingabe- *(oder entsprechend* Ausgabe-*)* linear*, wenn die Familie der Terme, die in seinen Eingabe- (bzw. Ausgabe-) Positionen stehen, linear ist.*

- *Ein Atom heißt* Eingabe-Ausgabe-disjunkt*, wenn die Familie der Terme, die in seinen Eingabe-Positionen vorkommen, keine Variablen gemeinsam hat mit der Familie der Terme, die in seinen Ausgabepositionen vorkommen.*

Damit gilt nun:

Lemma 5.29 (NSTO mittels Modes) *Seien A und H zwei Atome mit demselben Relationssymbol. Außerdem gelte:*

- *A und H haben keine Variable gemeinsam*
- *A oder H ist Eingabe-Ausgabe-disjunkt*
- *eines der beiden Atome ist Eingabe-linear, das andere Ausgabe-linear*

Dann ist A=H NSTO.

Beweis Wir unterscheiden vier Fälle. Sei zunächst A Eingabe-Ausgabe-disjunkt und Eingabe-linear und H Ausgabe-linear. Seien $e_1^A, \ldots, e_m^A$ (entsprechend $e_1^H, \ldots, e_m^H$) die Terme, die die Eingabepositionen von A (bzw. H) ausfüllen und $a_1^A, \ldots, a_n^A$ (bzw. $a_1^H, \ldots, a_n^H$) die, die in Ausgabeposition von A (bzw. H) stehen. Folgende Gleichungsmenge ist zu betrachten:

$$E := \{e_1^A = e_1^H, \ldots, e_m^A = e_m^H, a_1^A = a_1^H, \ldots, a_n^A = a_n^H\}$$

Dies ordnen wir um zu:

$$F := \{e_1^A = e_1^H, \ldots, e_m^A = e_m^H, a_1^H = a_1^A, \ldots, a_n^H = a_n^A\}$$

Nach Voraussetzung haben A und H keine Variablen gemeinsam, deswegen gilt:

- F ist linkslinear, weil A außerdem Eingabe- und H Ausgabe-disjunkt ist
- Weil A außerdem Eingabe-Ausgabe-disjunkt ist, besitzt für jedes j die Gleichung $e_j^A = e_j^H$ keinen Nachfolger und $a_j^H = a_j^A$ keinen Vorgänger bzgl. $\rightarrow_F$. Damit ist die Relation nicht zyklisch.

Wegen Lemma 5.26 ist $A = H$ NSTO. Für die verbleibenden Fälle erfolgt der Beweis analog. □

Dieses Resultat wollen wir nun auf Programme übertragen. Hierzu benötigen wir folgendes Konzept:

Definition 5.30 (Output Drivenness) *Eine LD-Resolution ist* output driven*, wenn jedes in ihr ausgewählte Atom Ausgabe-linear und Eingabe-Ausgabe-disjunkt ist.*

Damit gilt:

Satz 5.31 (Occur-Check) *Sei P ein definites Programm und G ein definites Ziel und es gelte:*

- *Der Kopf jeder Klausel in P ist Eingabe-linear.*
- *Alle LD-Ableitungen von $P \cup \{G\}$ sind output driven.*

Dann ist $P \cup \{G\}$ occur-check-frei.

Beweis Betrachte eine LD-Resolution von $P \cup \{G\}$. Sei A ein in ihr ausgewähltes Atom und H eine durch standardisation apart entstandene Variante des Kopfes einer Programmklausel von P, so dass H und A das gleiche Relationssymbol verwenden. Dann besitzen A und H keine gemeinsame Variable. Nach Voraussetzung ist H außerdem Eingabe-linear und A, wegen der Output-Drivenness, Ausgabe-linear und Eingabe-Ausgabe-disjunkt. Mittels Lemma 5.29 folgt das zu Zeigende. □

Dieser Satz bietet eine Möglichkeit, Occur-Check-Freiheit für Programme und Ziele zu garantieren. Die Bedingungen des Satzes sind jedoch in der obigen Form schwer nachweisbar. Wir formulieren eine Eigenschaft, die dies erleichtert.
Für die Definition vereinfachen wir die Notation der modes. Schreiben wir ein Atom als $p(\mathbf{u}, \mathbf{v})$, so sei $\mathbf{u}$ die Folge aller Terme, die eine Eingabeposition, $\mathbf{v}$ die Folge aller Terme die eine Ausgabeposition ausfüllen. Damit legen wir fest:

Definition 5.32 (Nice-Modedness)

- *Ein definites Ziel $\leftarrow p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$ heißt* nicely moded*, wenn $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n$ eine lineare Familie von Termen ist und für $i \in [1, n]$ gilt:*

$$\mathrm{var}(\boldsymbol{s}_i) \cap (\bigcup_{j=i}^{n} \mathrm{var}(\boldsymbol{t}_j)) = \emptyset \tag{5.1}$$

- *Eine Klausel*

$$p_0(\boldsymbol{s}_0, \boldsymbol{t}_0) \leftarrow p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$$

  heißt nicely moded*, wenn $\leftarrow p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$ nicely moded ist und wenn gilt:*

$$\mathrm{var}(\boldsymbol{s}_0) \cap (\bigcup_{j=1}^{n} \mathrm{var}(\boldsymbol{t}_j)) = \emptyset \tag{5.2}$$

  Insbesondere ist jeder Fakt nicely moded.

- *Ein Programm heißt* nicely moded*, wenn jede seiner Klauseln es ist.*

Anschaulich bedeutet das vorgestellte Konzept, dass in einem definiten Ziel jede Variable, die in der Ausgabeposition eines Atoms vorkommt, neu ist, das heißt, dass sie bis zu dem

Zeitpunkt, zu dem der Berechnungsprozess von Prolog diese erreicht, noch nicht im Programm aufgetaucht ist und deswegen auch noch nicht durch eine Substitution verändert wurde. Diese Eigenschaft garantiert für das Atom Eingabe-Ausgabe-Disjunktheit und Ausgabelinearität.
Wir wollen dies festhalten und verwenden folgendes Lemma:

Lemma 5.33 *Ein LD-Resolvent eines definiten Zieles und einer Klausel, die beide nicely moded sind, ist nicely moded.*

Der Beweis dieses Lemmas ist recht aufwendig und kann bei *Apt* und *Pellegrini* [AP94, S.719–724] nachgelesen werden.

Daraus folgt:

Lemma 5.34 *Seien P und G nicely moded. Dann sind alle LD-Ableitungen von $P \cup \{G\}$ output driven.*

Beweis Bei der LD-Resolution wird in jedem Teilziel das erste Atom ausgewählt. Für ein Ziel, das nicely moded ist, ist dieses Atom Eingabe-Ausgabe-disjunkt und Ausgabelinear. □

Nun erhalten wir aus Satz 5.31 und Lemma 5.34:

Korollar 5.35 *Seien P und G nicely moded und der Kopf jeder Klausel von P sei Eingabe-linear. Dann ist $P \cup \{G\}$ occur-check-frei.*

Das Nicht-Auftreten des Occur-Check-Problems lässt sich also durch einen Nachweis der in diesem Korollar geforderten Eigenschaften beweisen. Dieses Prinzip wenden wir in Abschnitt 6.5 an.
Neben der vorgestellten Nice-Modedness existieren andere Arten des modings, die Occur-Check-Freiheit garantieren. Diese kann man z.B. bei *Apt* und *Pellegrini* in [AP94] nachlesen.
Allgemein ist jedoch die Frage, ob ein Programm und ein Ziel NSTO sind, unentscheidbar, wie *Deransart* und *Maluszynski* in [DM85] aufzeigen. Deswegen bieten die meisten Prolog-Versionen, wie beispielsweise auch das im Naproche-System verwendete SWI-Prolog, die Möglichkeit, Unifikation mit Occur-Check durchzuführen. Dies leistet ein entsprechendes Prädikat. Da sich dies aber auf die Laufzeit des Programms auswirkt, kann davon nur in Ausnahmefällen Gebrauch gemacht werden.

5.4 Semantik

Nachdem in den letzten Abschnitten Methoden eingeführt wurden, die behandelten Probleme für Korrektheit und Vollständigkeit der Prolog-Resolution auszuschließen, betrachten wir an dieser Stelle das Ergebnis dieser, die Instanzen der Atome eines Zieles, die sich aus der erfolgreichen LD-Ableitung als logische Konsequenz eines Programmes ergeben.
Es wird hierbei gezeigt, dass die Eigenschaft eines Programmes, akzeptabel zu sein, die Suche nach seinem kleinsten Herbrand-Modell erleichtert. Wir vereinfachen die verwendeten Methoden weiter, indem wir nur das Modell betrachten, das sich ergibt, wenn jedes definite Ziel entsprechend vorgesehener Spezifikationen formuliert ist.
Als Alphabet der Programme werden alle Relations-, Funktions-, Konstanten- und Variablensymbole, die in Prolog definierbar sind, angenommen. Deshalb formulieren wir die Sätze dieses Abschnittes für unendliche Alphabete.

Für unsere Betrachtungen definieren wir die Instanzen eines definiten Zieles, die sich durch Anwendung von Antworten ergeben:

Definition 5.36 (Berechnete und korrekte Instanz) *Sei P ein Programm und $G =\leftarrow A_1, \ldots, A_n$ ein definites Ziel.*

- *Wir nennen $G' =\leftarrow A'_1, \ldots, A'_n$ eine* korrekte Instanz *von G, wenn eine korrekte Antwort θ auf $P \cup \{G\}$ existiert, so dass $G' = G\theta$ gilt, d.h. G' ist eine Instanz von G und $P \models A'_1, \ldots, A'_n$.*
- *Wir nennen G' eine berechnete Instanz von G, wenn für eine berechnete Antwort θ die Gleichung $G' = G\theta$ erfüllt ist.*

Wir führen folgende Schreibweise ein:

Definition 5.37 *Sei P ein Programm, G ein definites Ziel und $\mathcal{G}$ eine Menge von definiten Zielen. Wir schreiben $\{G\}\ P\ \mathcal{G}$, um auszudrücken, dass $\mathcal{G}$ die Menge der mittels P berechneten Instanzen von G ist.*

Wegen Korrektheit und starker Vollständigkeit der SLD-Resolution sind berechnete und korrekte Instanzen eng miteinander verbunden und es ergibt sich:

Satz 5.38 *Sei das Herbrand-Universum des zu Grunde liegenden Alphabetes $\mathcal{A}$ unendlich. Sei P ein Programm, G ein Ziel und $\mathcal{G}$ die Menge aller korrekten Grundinstanzen von G. $\mathcal{G}$ sei außerdem endlich. Dann gilt $\{G\}\ P\ \mathcal{G}$.*

Beweis Weil das Herbrand-Universum von $\mathcal{A}$ unendlich ist, die Menge $\mathcal{G}$ jedoch endlich, gilt:

$$\text{Jede korrekte Instanz } G' \text{ von } G \text{ ist eine Grundinstanz.} \tag{5.3}$$

Betrachte $G_1 \in \mathcal{G}$. Wegen des Vollständigkeitssatzes 3.41 existiert eine berechnete Instanz G_2 von G, so dass G_1 eine Instanz von G_2 ist. Wegen des Korrektheitssatzes 3.37 ist G_2 eine korrekte Instanz von G und wegen (5.3) ist G_2 eine Grundinstanz. Damit gilt $G_2 = G_1$ und deswegen ist G_1 eine berechnete Instanz von G.
Wähle umgekehrt eine berechnete Instanz G_1 von G. Wegen des Korrektheitssatzes 3.37 ist G_1 eine korrekte Instanz von G, mit (5.3) folgt, dass G_1 eine Grundinstanz ist, es gilt also $G_1 \in \mathcal{G}$. □

Es folgt:

Korollar 5.39 *Das Herbrand-Universum des zu Grunde liegenden Alphabetes $\mathcal{A}$ sei unendlich. Sei P ein Programm, A ein Atom und die Menge* $\text{grund}(A) \cap \mathcal{M}(P)$ *sei endlich. Dann gilt:*

$$\{\leftarrow A\}\ P\ \ \text{grund}(A) \cap \mathcal{M}(P)$$

Mit Hilfe dieses Korollars lässt sich die Menge aller berechneten Instanzen eines Atoms bestimmen. Hierzu wird jedoch das kleinste Herbrand-Modell des Programmes benötigt. Um dies zu erhalten, benutzen wir die Abbildung T_P aus Satz 3.15. Dieser vereinfacht sich für akzeptable Programme:

Satz 5.40 (Eindeutigkeit des Fixpunktes) *Sei das Programm P akzeptabel. Dann ist $\mathcal{M}(P)$ der einzige Fixpunkt der Abbildung T_P.*

Beweis Sei das Programm P akzeptabel bzgl. eines Modells und der Abbildung $|\ |$. Aus der Definition des kleinsten Herbrand-Modells und der Akzeptabilität folgt, dass P ebenfalls akzeptabel bzgl. $|\ |$ und $\mathcal{M}(P)$ ist. Sei $\mathfrak{I}$ ein Fixpunkt von T_P. Wegen Satz 3.15 gilt $\mathcal{M}(P) \subseteq \mathfrak{I}$.
Angenommen $\mathfrak{I} \neq \mathcal{M}(P)$. Sei $A \in \mathfrak{I} - \mathcal{M}(P)$. Dann existiert wegen der Wahl von $\mathfrak{I}$ eine Klausel $A \leftarrow B_1, \ldots, B_n \in \text{grund}(P)$ mit $\mathfrak{I} \models B_1, \ldots, B_n$. Wegen $\mathcal{M}(P) \not\models A$ gilt auch $\mathcal{M}(P) \not\models B_1, \ldots, B_n$. Sei B_i das erste Atom in $B_1, \ldots, B_n$, für das $\mathcal{M}(P) \not\models B_i$ gilt. Dann gilt $B_i \in \mathfrak{I} - \mathcal{M}(P)$ und, wegen der Akzeptabilität von P, $|A| > |B_i|$. Für jedes Element $A \in \mathfrak{I} - \mathcal{M}(P)$ existiert damit ein Element $B \in \mathfrak{I} - \mathcal{M}(P)$, so dass $|A| > |B|$ gilt. Dies widerspricht der Fundiertheit von $(>, \mathbb{N})$. □

Trotz der Vereinfachung kann die Anwendung dieses und des vorherigen Satzes, um die tatsächlichen berechneten Instanzen eines definiten Zieles zu bestimmen, sehr aufwendig sein. Das kleinste Herbrand-Modell eines Programmes erfasst alle atomaren Formeln, die logische Konsequenz von diesem sind. Im praktischen Gebrauch werden Programme jedoch zu einem bestimmten Zweck geschrieben und damit nur auf eine bestimmte Weise benutzt. Meist ist festgelegt, wie die Eingabe eines Programmes aussehen und welche Leistung es erbringen soll. Wir betrachten im Folgenden nur die Teilmenge des kleinsten Herbrand-Modells, die Instanzen von „richtig“, d.h. im Sinne einer vorgegebenen Spezifikation, formulierten Zielen enthält. Hierzu definieren wir:

Definition 5.41 (Assertion) *Sei p ein Relationssymbol.*

- *Eine* Assertion für *p ist eine unter Substitution abgeschlossene Menge von p-Atomen.*
- *Eine* Assertion *ist eine Assertion für ein Relationssymbol p.*
- *Wir sagen, eine Assertion $\mathcal{A}$* gilt *für ein Atom A, wenn $A \in \mathcal{A}$*
- *Eine* Spezifikation *für p ist ein Paar pre_p, post_p von Assertionen für p. Wir nennen pre_p (bzw. post_p) eine Prä-Assertion (bzw. Post-Assertion) für p.*
- *Eine* Spezifikation *ist eine Menge von Spezifikationen für verschiedene Relationssymbole.*

Wir gehen in diesem Abschnitt von Folgendem aus:

Annahme: Jede betrachtete Relation besitzt eine feste Spezifikation.

Im Falle, dass eine Relation auf verschiedene Weisen verwendet werden kann, ändern wir für jede Verwendungsweise den Namen der Relation und führen jede umbenannte Relation mit ihrer Definition einzeln im Programm auf.
Damit wir Spezifikationen verwenden können, müssen wir sicher stellen, dass sie korrekt sind, d.h. dass die Atome jeder berechneten Instanz eines definiten Zieles, das entsprechend der Prä-Assertionen der einzelnen Relationen formuliert ist, Elemente der jeweiligen Post-Assertionen ihrer Relationen sind. Hierzu führen wir folgende Schreibweise ein:

Definition 5.42 *Gegeben seien die Atome $A_1, \ldots, A_n, A_{n+1}$ und die Assertionen $\mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{A}_{n+1}$, $n \geq 0$. Wir schreiben*

$$\models A_1 \in \mathcal{A}_1, \ldots, A_n \in \mathcal{A}_n \Rightarrow A_{n+1} \in \mathcal{A}_{n+1},$$

um auszudrücken, dass für alle Substitutionen θ aus $A_1\theta \in \mathcal{A}_1, \ldots, A_n\theta \in \mathcal{A}_n$ folgt, dass $A_{n+1}\theta \in \mathcal{A}_{n+1}$ gilt.

Wir kürzen $A \in \text{pre}_{\text{rel}(A)}$ durch $\text{pre}(A)$ ab, entsprechendes gilt für post. Damit schreiben wir, gegeben ein Atom $p(s)$ und eine Prä-Assertion pre_p, für $p(s) \in \text{pre}_p$ den Ausdruck $\text{pre}(p(s))$ und definieren:

Definition 5.43 (Well-Assertedness)

- *Ein definites Ziel $\leftarrow A_1, \ldots, A_n$ heißt* well-asserted, *wenn für $j \in [1, n]$ gilt:*

$$\models \text{post}(A_1), \ldots, \text{post}(A_{j-1}) \Rightarrow \text{pre}(A_j)$$

- *Eine Klausel $H \leftarrow B_1, \ldots, B_n$ heißt* well-asserted, *wenn für $j \in [1, n+1]$ gilt:*

$$\models \text{pre}(H), \text{post}(B_1), \ldots, \text{post}(B_{j-1}) \Rightarrow \text{pre}(B_j),$$

wobei $\text{pre}(B_{n+1}) := \text{post}(H)$.

- *Ein Programm heißt* well-asserted*, wenn jede seiner Klauseln es ist.*

Ein atomares Ziel $\leftarrow A$ ist damit well-asserted, wenn $\models \text{pre}(A)$ gilt und ein Fakt $A \leftarrow$ ist well-asserted, wenn $\models \text{pre}(A) \Rightarrow \text{post}(A)$ gilt.
Nehmen wir an, dass für jedes Atom, das in der Prä-Assertion seines Relationssymbols enthalten ist, jede berechnete Instanz Element von dessen Post-Assertion ist. Dann stellt die Well-Assertedness eines definiten Zieles sicher, dass die Variablen jedes seiner Atome zu dem Zeitpunkt, zu dem es innerhalb einer LD-Resolution ausgewählt wird, so substituiert sind, dass das Atom in der Prä-Assertion seines Relationssymbols enthalten ist.
Wir können außerdem zeigen, dass die Well-Assertedness eines solchen Zieles unter Anwendung eines SLD-Resolutionsschrittes erhalten bleibt, wenn das zu Grunde liegende Programm ebenfalls well-asserted ist:

Lemma 5.44 *Ein SLD-Resolvent eines definiten Zieles und eines definiten Programmes, die beide well-asserted sind, ist well-asserted.*

Beweis Wir gehen in zwei Schritten vor:

1. Besitzt eine definite Klausel oder ein definites Ziel die Eigenschaft, well-asserted zu sein, so gilt dies auch für jede Instanz dieser.

   Beweis: Dies folgt aus der Abgeschlossenheit der Assertionen unter Substitution.

2. Das definite Ziel $G = \leftarrow A_1, \ldots, A_k, H, A_{k+l+1}, \ldots, A_{k+l+m}$ und die Klausel $C = H \leftarrow A_{k+1}, \ldots, A_{k+l}$ seien well-asserted, dann ist auch das definite Ziel

   $$\leftarrow A_1, \ldots, A_{k-1}, A_k, \ldots, A_{k+l}, A_{k+l+1}, \ldots, A_n$$

   well-asserted.

   Beweis: Wir zeigen für $i \in [1, k+l+m]$:

   $$\models \text{post}(A_1), \ldots, \text{post}(A_{i-1}) \Rightarrow \text{pre}(A_i)$$

   Hierbei behandeln wir drei Fälle:

   Fall 1: $i \in [1, k]$. Weil G well-asserted ist, ist auch das Teilziel $\leftarrow A_1, \ldots, A_k$ well-asserted, die Aussage folgt.

   Fall 2: $i \in [k+1, k+l]$. Wegen der Well-Assertedness von G und C gilt:

   $$\models \text{post}(A_1), \ldots, \text{post}(A_k) \Rightarrow \text{pre}(H)$$

   und

   $$\models \text{pre}(H), \text{post}(A_{k+1}), \ldots, \text{post}(A_{i-1}) \Rightarrow \text{pre}(A_i)$$

   Es folgt die Behauptung.

Fall 3: $i \in [k+l+1, k+l+m]$. Weil G well-asserted ist, gilt:

$$\models \text{post}(A_1), \ldots, \text{post}(A_k), \text{post}(H), \text{post}(A_{k+l+1}), \ldots, \text{post}(A_{i-1}) \Rightarrow \text{pre}(A_i)$$

und

$$\models \text{post}(A_1), \ldots, \text{post}(A_k) \Rightarrow \text{pre}(H).$$

Weil C well-asserted ist, gilt

$$\models \text{pre}(H), \text{post}(A_{k+1}), \ldots, \text{post}(A_{k+l}) \Rightarrow \text{post}(H).$$

Daraus folgt das zu Zeigende.

□

Aus diesem Lemma können wir verschiedene Schlüsse ziehen. Es folgt direkt:

Korollar 5.45 *Seien P und G well-asserted. Dann sind alle definiten Ziele in allen SLD-Resolutionen von $P \cup \{G\}$ well-asserted.*

Für LD-Ableitungen gilt außerdem:

Korollar 5.46 *Seien P und G well-asserted und sei ξ eine LD-Ableitung von $P \cup \{G\}$. Dann gilt $\models \text{pre}(A)$ für jedes in ξ ausgewählte Atom.*

Beweis Für ein definites Ziel $\leftarrow A_1, \ldots, A_n$, das well-asserted ist, gilt $\models \text{pre}(A_1)$, damit folgt das zu Zeigende aus dem vorherigen Korollar. □

Sehr wichtig für spätere Sätze ist folgende Erkenntnis:

Korollar 5.47 *Seien P und G well-asserted. Dann gilt für jede berechnete Instanz $\leftarrow A_1, \ldots, A_n$ von G und jedes $j \in [1, n]$:*

$$\models \text{post}(A_j).$$

Beweis Seien $G = \leftarrow p_1(\mathbf{s}_1), \ldots, p_k(\mathbf{s}_k)$ und $\mathbf{s}_1, \ldots, \mathbf{s}_k$ Folgen von Termen. Sei p ein neues Funktionssymbol der Stellenzahl, die sich aus der Summierung der Stellenzahlen von $p_1, \ldots, p_k$ ergibt. Wir definieren folgende Prä-Assertion für p:

$$p(\mathbf{s}_1, \ldots, \mathbf{s}_k) \in \text{pre}_p \text{ gdw. } p_1(\mathbf{s}_1) \in \text{post}_{p_1}, \ldots, p_k(\mathbf{s}_k) \in \text{post}_{p_k}.$$

Weil jedes $post_{p_i}$ als Assertion abgeschlossen unter Substitution ist, gilt dies auch für pre_p. Die Post-Assertion für p sei beliebig definiert.

Nun ist das definite Ziel $\leftarrow p_1(\mathbf{s}_1), \ldots, p_k(\mathbf{s}_k), p(\mathbf{s}_1, \ldots, \mathbf{s}_k)$ well-asserted, weil mit der Definition von pre_p gilt:

$$\text{post}(p_1(\mathbf{s}_1)), \ldots, \text{post}(p_k(\mathbf{s}_k)) \Rightarrow \text{pre}(p(\mathbf{s}_1, \ldots, \mathbf{s}_k)).$$

Sei θ eine berechnete Antwort für G. Dann ist $\leftarrow p(\mathbf{s}_1, \ldots, \mathbf{s}_k)\theta$ ein Ziel in einer SLD-Ableitung von $P \cup \{p(\mathbf{s}_1, \ldots, \mathbf{s}_k)\}$. Wegen Korollar 5.45 gilt damit $\models \text{pre}(p(\mathbf{s}_1, \ldots, \mathbf{s}_k)\theta)$ und wegen der Definition von pre_p

$$\models \text{pre}(p(\mathbf{s}_1, \ldots, \mathbf{s}_k)\theta) \Rightarrow \text{post}(p_1(\mathbf{s}_1)\theta), \ldots, \text{post}(p_k(\mathbf{s}_k)\theta).$$

Daraus folgt die Aussage des Korollars. □

Für ein Atom A sei $\text{inst}(A)$ die Menge aller seiner Instanzen. Dann folgt aus dem vorherigen Korollar:

Korollar 5.48 *Seien P ein Programm und $\leftarrow A$ ein atomares Ziel, die beide well-asserted sind und es gelte $p = \text{rel}(A)$. Sei $\mathcal{G}$ die Menge aller mittels P berechneten Instanzen von $\leftarrow A$. Dann gilt:*

$$\mathcal{G} \subseteq \text{inst}(A) \cap \text{post}_p$$

Um Korollar 5.39 und Satz 5.40 für Programme und Ziele zu formulieren, die well-asserted sind, definieren wir:

Definition 5.49 *Sei das Programm P well-asserted. Dann gelte*

$$\text{pre} := \bigcup_{p \text{ ist in } P} \text{grund}(\text{pre}_p),$$

$$\text{post} := \bigcup_{p \text{ ist in } P} \text{grund}(\text{post}_p),$$

$$\mathcal{M}_{(\text{pre,post})}(P) := \mathcal{M}(P) \cap \text{pre},$$

$$\text{pre}(P) := \{H \leftarrow B_1, \ldots, B_n \in \text{grund}(P) | H \in \text{pre}\}.$$

Das Modell $\mathcal{M}_{(\text{pre,post})}(P)$ kann als die „richtig formulierte“ Teilmenge des kleinsten Herbrand-Modells angesehen werden. Wir halten folgende Beobachtung fest:

Bemerkung 5.50 *Sei das Programm P well-asserted. Dann gilt $\mathcal{M}_{(\text{pre,post})}(P) \subseteq \text{post}$.*

Beweis Betrachte ein Grundatom $A \in \mathcal{M}(P) \cap \text{pre}$. Dann ist das definite Ziel $\leftarrow A$ well-asserted. Wegen $A \in \mathcal{M}(P)$ existiert außerdem eine erfolgreiche LD-Ableitung von $P \cup \{\leftarrow A\}$. Wegen Korollar 5.47 folgt $A \in \text{post}$. □

Damit können wir zeigen:

Satz 5.51 ($\mathcal{M}_{(\text{pre,post})}(P)$) *Sei das Programm P well-asserted, dann gilt:*

$$\mathcal{M}_{(\text{pre,post})}(P) = \mathcal{M}(\text{pre}(P))$$

Beweis Wegen Satz 3.15 gilt $\mathcal{M}_{(\text{pre,post})}(P) = T_P \uparrow \omega \cap \text{pre}$ und $\mathcal{M}(\text{pre}(P)) = T_{\text{pre}(P)} \uparrow \omega$. Wir zeigen per Induktion, dass für $n \geq 0$ gilt:

$$T_{\text{pre}(P)} \uparrow n = T_P \uparrow n \cap \text{pre}\,.$$

Induktionsanfang: Für $n = 0$ ist die Aussage klar.

Induktionsschritt: Es gelte:

$$T_{\text{pre}(P)} \uparrow (n-1) = T_P \uparrow (n-1) \cap \text{pre} \tag{5.4}$$

Wir behandeln die Inklusionen einzeln:

1. „$\subseteq$“:

$$\begin{array}{rl} & T_{\text{pre}(P)} \uparrow n \\ = & T_{\text{pre}(P)}(T_{\text{pre}(P)} \uparrow (n-1)) \\ = & T_P(T_{\text{pre}(P)} \uparrow (n-1)) \cap \text{pre} \\ \overset{(5.4)}{=} & T_P(T_P \uparrow (n-1) \cap \text{pre}) \cap \text{pre} \\ \underset{\textit{Monotonie von } T_P}{\subseteq} & T_P(T_P \uparrow (n-1)) \cap \text{pre} \\ = & T_P \uparrow n \cap \text{pre} \end{array}$$

2. „$\supseteq$“:
   Sei $H \in T_P \uparrow n \cap \text{pre}$. Dann existiert $H \leftarrow B_1, \ldots, B_m \in \text{grund}(P)$, so dass

$$\{B_1, \ldots, B_m\} \subseteq T_P \uparrow (n-1). \tag{5.5}$$

   Wir beweisen durch Induktion über m, dass ebenfalls

$$\{B_1, \ldots, B_m\} \subseteq \text{pre} \tag{5.6}$$

   gilt.

   Induktionsanfang: Für $m = 0$ ist die Aussage klar.

   Induktionsschritt: $m > 0$. Es gelte $\{B_1, \ldots, B_{m-1}\} \subseteq \text{pre}$. Wegen (5.5) gilt damit auch $\{B_1, \ldots, B_{m-1}\} \subseteq \mathcal{M}_{(\text{pre,post})}(P)$ und aus Bemerkung 5.50 folgt dann

$\{B_1, \ldots, B_{m-1}\} \subseteq \text{post}$. Nun ist wegen $H \in \text{pre}$ und der Well-Assertedness $B_m \in \text{pre}$. Es gilt also (5.6). Damit ist die Induktion über m abgeschlossen.

Nun gilt wegen (5.4), (5.5) und (5.6):

$$\{B_1, \ldots, B_m\} \subseteq T_{\text{pre}(P)} \uparrow (n-1).$$

Daraus folgt $H \in T_{\text{pre}(P)} \uparrow n$. Dies beendet den eigentlichen Induktionsbeweis.

□

Es folgen die gewünschten Resultate:

Korollar 5.52 *Sei das Herbrand-Universum des zu Grunde liegenden Alphabetes $\mathcal{A}$ unendlich. Sei P ein Programm und $\leftarrow A$ ein atomares Ziel, die beide well-asserted sind und die Menge* $\text{grund}(A) \cap \mathcal{M}_{(\text{pre,post})}$ *sei endlich. Dann gilt:*

$$\{A\}\ P\ \ \text{grund}(A) \cap \mathcal{M}_{(\text{pre,post})}$$

Beweis Für ein atomares Ziel $\leftarrow A$, das well-asserted ist, gilt $\models \text{pre}(A)$ und damit $\text{grund}(A) \subseteq \text{pre}$. Also:

$$\text{grund}(A) \cap \mathcal{M}_{(\text{pre,post})}(P) = \text{grund}(A) \cap \mathcal{M}(P).$$

Die Behauptung folgt mit Korollar 5.39. □

Satz 5.53 (Eindeutigkeit des Fixpunktes II) *Sei das Programm P well-asserted und akzeptabel. Dann ist $\mathcal{M}_{(\text{pre,post})}(P)$ der einzige Fixpunkt von $T_{\text{pre}(P)}$.*

Beweis Wegen der Akzeptabilität von P ist auch das Programm $\text{grund}(P)$ und damit auch $\text{pre}(P)$ akzeptabel. Die Behauptung folgt direkt aus Satz 5.51 und der Anwendung von Satz 5.40 auf $\text{pre}(P)$. □

Mit Hilfe dieser Sätze kann man für entsprechende Ziele und Programme die jeweilige Menge der berechneten Instanzen bestimmen. Wir zeigen dies in Abschnitt 6.6 an einem Beispiel.

6 Korrektheitsbeweise am Beispiel der Formelgrammatik

Die Datenerfassung des Naproche-Systems erfolgt weitgehend über Grammatiken. Exemplarisch wenden wir die Verfahren des letzten Kapitels an einer vereinfachten Form der Grammatik `expr_grammar.pl` an, die im Eingabetext verwendete mathematische Ausdrücke in ein internes Listenformat übersetzt. Die hier gezeigten Prinzipien lassen sich aber auch auf die anderen Grammatiken im Naproche-System übertragen.

6.1 Die Formelsprache von Naproche

Das Eingabeformat für mathematische Formeln des Naproche-Systems orientiert sich an der in Abschnitt 2.1 gegebenen Definition von Formeln erster Stufe. In der Praxis kann diese zu Grunde liegende Definition jedoch je nach Autor des mathematischen Textes variieren. So können z.B. zweistellige Relationssymbole entweder in Infix- oder Präfixnotation aufgeschrieben werden oder Klammern, abhängig von der Eindeutigkeit der Formulierung, hinzugefügt oder weggelassen werden. Das Naproche-System versucht flexibel mit diesen Variationen umzugehen und akzeptiert oft mehrere Formulierungen. Trotzdem bleibt die Formelsprache beschränkt und muss den in diesem Absatz gegebenen Definitionen genügen.

Wir beginnen mit dem Formelalphabet. Im Naproche-System besteht dieses aus einer endlichen Anzahl von Zeichen, die im Wörterbuch `math_lexicon.pl`[1] festgelegt sind. Dieses Wörterbuch kann um beliebig viele Zeichen erweitert werden. Für die hier angestrebten theoretischen Betrachtungen genügt es, folgende Teilmenge zu betrachten:

Definition 6.1 (Formelalphabet) *Das Formelalphabet $\mathcal{A}_N$ besteht aus folgenden Symbolen:*

(i) Variablen: x, y, z

(ii) Konnektionssymbole oder logische Symbole: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$

(iii) Quantorensymbole: $\forall$, $\exists$

(iv) Hilfssymbole: Klammern (,), *Komma* ,

[1] Vgl. Anhang A.1.2.

(v) *Relationssymbole:*
   a) *0-stellig: contradiction*
   b) *1-stellig: Ord, Trans*
   c) *2-stellig: R, =, ∈, <, >*

(vi) *Funktionssymbole:*
   a) *1-stellig: f*
   b) *2-stellig: g*
   c) *3-stellig: h*

(vii) *Konstanten: c, ∅*

Die Terme werden genau wie in Definition 2.3 gebildet. Der Vollständigkeit wegen sei diese hier noch einmal aufgeführt:

Definition 6.2 (Terme in Naproche) *Die Menge der Terme $\mathcal{T}_N$ von Naproche besteht aus der kleinsten Teilmenge von $\mathcal{A}_N^*$, für die folgendes gilt:*

(T1) *Jede Variable aus $\mathcal{A}_N$ ist in $\mathcal{T}_N$.*

(T2) *Jede Konstante aus $\mathcal{A}_N$ ist in $\mathcal{T}_N$.*

(T3) *Ist f ein n-stelliges Funktionssymbol aus $\mathcal{A}_N$ und $t_1, \ldots, t_n \in \mathcal{T}_N$, so gilt $f(t_1, \ldots, t_n) \in \mathcal{T}_N$.*

Im Fall (T3) nennen wir den Ausdruck $f(t_1, \ldots, t_n)$ Funktion.

Das Naproche-System verwendet folgende Regeln, um aus Termen und Alphabet Formeln zu konstruieren:

Definition 6.3 (Formeln in Naproche) *Die Menge $\mathcal{F}_N$ der Formeln von Naproche ist die kleinste Teilmenge von $\mathcal{A}_N^*$, so dass gilt:*

(i) *Ist $r \in \mathcal{A}_N$ ein n-stelliges Relationssymbol und $t_1, \ldots, t_n \in \mathcal{T}_N$, so gilt $r(t_1, \ldots, t_n) \in \mathcal{F}_N$.*

(ii) *Ist $r \in \mathcal{A}_N$ ein 2-stelliges Relationssymbol und sind $t_1, t_2 \in \mathcal{T}_N$, so gilt $t_1 r t_2 \in \mathcal{F}_N$.*

(iii) *Sind ϕ und $\psi \in \mathcal{F}_N$, so gilt auch $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi \in \mathcal{F}_N$.*

(iv) *Ist $\phi \in \mathcal{F}_N$ und sind $x_1, \ldots, x_n$ Variablen aus $\mathcal{A}_N$, so gilt $\forall x_1 \ldots x_n \phi$ und $\exists x_1 \ldots x_n \phi \in \mathcal{F}_N$.*

(v) *Gilt $\phi \in \mathcal{F}_N$, so gilt auch $(\phi) \in \mathcal{F}_N$.*

Diese Definition gewährt dem Nutzer die Freiheit, bei Formeln der Form $(\phi \wedge \psi)$ die Klammern wegzulassen. Dies ist aber auch eine Quelle für mögliche Mehrdeutigkeit. So

ist z.B. auch $\phi \wedge \psi \vee \xi$ eine Formel von $\mathcal{F}_N$. Dies kann aber entweder als $(\phi \wedge (\psi \vee \xi))$ oder als $((\phi \wedge \psi) \vee \xi)$ verstanden werden. Die Formelgrammatik übersetzt den logischen Ausdruck in beide Lesarten, für die Beweisprüfung wählt Naproche 0.2 die erste Interpretation. Um so entstehende Missverständnisse sicher zu vermeiden, kann man mathematische Formeln nach einer modifizierten Form von Definition 6.3 formulieren. Statt Punkt (iii) gelte folgende Regel:

(iii') Sind ϕ und $\psi \in \mathcal{F}_N$, so gilt auch $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi) \in \mathcal{F}_N$.

Die vereinfachte Version von `expr_grammar.pl`, die wir in dieser Arbeit betrachten werden, akzeptiert nur Formeln in der letzten Version als Eingabe. Deswegen sei, wenn nicht anders angegeben, mit $\mathcal{F}_N$ im Folgenden immer die modifizierte Formelmenge gemeint.

6.2 Das interne Listenformat

In mathematischen Texten kommen formelsprachliche Ausdrücke entweder als mathematische Formeln oder als Terme vor. Das Modul `expr_grammar.pl` übersetzt logische Formeln und Terme eines Eingabetextes in ein internes Listenformat. In diesem werden für jeden Term und jede Formel bestimmte Merkmale gespeichert. Um diese Merkmalslisten zu definieren sei *name* eine Funktion, die jedem Element des Formelalphabetes $\mathcal{A}_N$ seinen Namen, eine Prolog-Konstante, eindeutig zuordnet. Damit gilt nun:

Definition 6.4 (Merkmalslisten in Naproche) *Sei $\phi \in \mathcal{T}_N \cup \mathcal{F}_N$. Die Merkmalsliste $M(\phi)$ von ϕ sei folgendermaßen definiert:*

(i) Ist ϕ eine Variable, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(\phi) \\ type & : & variable \\ arity & : & 0 \end{bmatrix}$$

(ii) Ist ϕ eine Konstante, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(\phi) \\ type & : & constant \\ arity & : & 0 \end{bmatrix}$$

(iii) Ist ϕ von der Form $f(t_1, \ldots, t_n)$, f ein n-stelliges Funktionssymbol und $t_1, \ldots, t_n \in \mathcal{T}_N$, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(f) \\ type & : & function \\ arity & : & n \\ arguments & : & [M(t_1), \ldots, M(t_n)] \end{bmatrix}$$

(iv) Ist ϕ von der Form $r(t_1, \ldots, t_n)$, r ein n-stelliges Relationssymbol und $t_1, \ldots, t_n \in \mathcal{T}_N$, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(r) \\ type & : & relation \\ arity & : & n \\ arguments & : & [M(t_1), \ldots, M(t_n)] \end{bmatrix}$$

(v) Ist ϕ von der Form $f_1 r f_2$, r ein 2-stelliges Relationssymbol und $f_1, f_2 \in \mathcal{F}_N$, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(r) \\ type & : & relation \\ arity & : & 2 \\ arguments & : & [M(f_1), M(f_2)] \end{bmatrix}$$

(vi) Ist ϕ von der Form $(\neg\psi)$ mit $\psi \in \mathcal{F}_N$, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(\neg) \\ type & : & logical\_symbol \\ arity & : & 1 \\ arguments & : & [M(\psi)] \end{bmatrix}$$

(vii) Ist ϕ von der Form $(\psi * \xi)$ mit $\psi, \xi \in \mathcal{F}_N$ und $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(*) \\ type & : & logical\_symbol \\ arity & : & 2 \\ arguments & : & [M(\psi), M(\xi)] \end{bmatrix}$$

(viii) Ist ϕ von der Form $Qx_1 \ldots x_n\psi$ mit $Q \in \{\exists, \forall\}$ und $x_1, \ldots, x_n$ Variablen aus $\mathcal{A}_N$, so gilt:

$$M(\phi) := \begin{bmatrix} name & : & name(Q) \\ type & : & quantifier \\ arity & : & 2 \\ arguments & : & [[M(x_1), \ldots, M(x_n)], M(\psi)] \end{bmatrix}$$

(ix) Ist ϕ von der Form (ϕ') und $\phi' \in \mathcal{F}_N$, so gilt $M(\phi) := M(\phi')$

Die Menge aller Merkmalslisten für $\mathcal{T}_N \cup \mathcal{F}_N$ bezeichnen wir mit $\mathcal{M}_N$.

Beispiel Für die Formel $x < c$ ergibt sich folgende Merkmalsliste:

$$\begin{bmatrix} name & : & < \\ type & : & relation \\ arity & : & 2 \\ arguments & : & \left[\begin{bmatrix} name & : & x \\ type & : & variable \\ arity & : & 0 \end{bmatrix}, \begin{bmatrix} name & : & c \\ type & : & constant \\ arity & : & 0 \end{bmatrix} \right] \end{bmatrix}$$

6.3 Programmcode

Wir betrachten eine vereinfachte Version des Programmcodes von `expr_grammar.pl`. Hierzu integrieren wir das mathematische Lexikon direkt in den Programmcode der Grammatik. Klauseln in Grammatikschreibweise führen wir übersetzt in ihre reine Prologschreibweise auf.[2] Die Merkmalslisten werden mit Hilfe der Prolog-Erweiterung GULP[3] dargestellt. Der GULP-Ausdruck:

$$\texttt{name} \sim name(x)..\texttt{type} \sim type(x)..\texttt{arity} \sim arity(x)..\texttt{arguments} \sim arguments(x)$$

sei hierbei gleichbedeutend mit:

$$\begin{bmatrix} \texttt{name} & : & name(x) \\ \texttt{type} & : & type(x) \\ \texttt{arity} & : & arity(x) \\ \texttt{arguments} & : & arguments(x) \end{bmatrix}$$

Die zu übersetzende Formel wird als Liste von Prolog-Konstanten eingegeben. Um spätere Betrachtungen zu erleichtern und um das Zeichen „=“ im Programmcode vom Gleichheitszeichen in seiner üblichen Bedeutung unterscheiden zu können, ersetzen wir dieses im Programmcode durch die Relation `gleich`. Diese sei durch die Klausel

```
gleich(X,X).
```

definiert.

Wir erhalten das lauffähige Programm `EXPRESSION`, das aus den folgenden Klauseln besteht:

Für das *Alphabet*:

(i) Variablensymbole:

```
variable(type∼variable.. arity∼0.. name∼'x', ['x'|X], X).[4]
variable(type∼variable.. arity∼0.. name∼'y', ['y'|X], X).
variable(type∼variable.. arity∼0.. name∼'z', ['z'|X], X).
```

(ii) Konnektionssymbole oder logische Symbole:

```
logsym(type∼logical_symbol.. arity∼1.. name∼'∼', ['¬'|X], X).
logsym(type∼logical_symbol.. arity∼2.. name∼'&', ['∧'|X], X).
logsym(type∼logical_symbol.. arity∼2.. name∼'|', ['∨'|X], X).
```

[2] Den Programmcode der Originalversionen von `expr_grammar.pl` und der vereinfachten Version in Grammatikschreibweise findet man in Anhang A.1.1 bzw. A.1.3.

[3] Vgl. Abschnitt 4.4.

[4] An dieser Stelle machen wir Gebrauch von der in Abschnitt 4.4 beschriebenen Technik der Differenzlisten.

```
logsym(type∼logical_symbol.. arity∼2.. name∼'=>', ['→'|X], X).
logsym(type∼logical_symbol.. arity∼2.. name∼'<=>', ['↔'|X], X).
```

(iii) Quantorensymbole:

```
quantifier(type∼quantifier.. arity∼2.. name∼'!', ['∀'|X], X).
quantifier(type∼quantifier.. arity∼2.. name∼'?', ['∃'|X], X).
```

(iv) Hilfssymbole:

```
lb(['('|X], X).
rb([')'|X], X).
comma([','|X], X).
```

(v) Relationssymbole:

```
relationsymb(type∼relation..arity∼0..name∼'$false',
                                              ['contradiction'|X],X).
relationsymb(type∼relation..arity∼1..name∼'ord', ['Ord'|X], X).
relationsymb(type∼relation..arity∼1..name∼'trans', ['Trans'|X],X).
relationsymb(type∼relation..arity∼2..name∼'r', ['R'|X], X).
relationsymb(type∼relation..arity∼2..name∼'=', ['='|X], X).
relationsymb(type∼relation..arity∼2..name∼'in', ['∈'|X], X).
relationsymb(type∼relation..arity∼2..name∼'less', ['<'|X], X).
relationsymb(type∼relation..arity∼2..name∼'greater', ['>'|X], X).
```

(vi) Funktionssymbole:

```
funcsym(type∼function.. arity∼1.. name∼'f', ['f'|X], X).
funcsym(type∼function.. arity∼2.. name∼'g', ['g'|X], X).
funcsym(type∼function.. arity∼3.. name∼'h', ['h'|X], X).
```

(vii) Konstantensymbole:

```
const(type∼constant.. arity∼0.. name∼'emptyset', ['∅'|X], X).
const(type∼constant.. arity∼0.. name∼'c', ['c'|X], X).
```

Für *Terme*:

(T1)
```
term(X0, Y0, Y1) ←
            variable(X0, Y0, Y1).
```

(T2)
```
term(X0, Y0, Y1) ←
            const(X0, Y0, Y1).
```

(T3)
```
term(X0, Y0, Y1) ←
            function(X0, Y0, Y1).
```

Für *Funktionen*:

(F1)
```
function(X0, Y0, Y4) ←
             gleich(X0,arity∼1.. arguments∼[X1]),
```

```
                funcsym(X0, Y0, Y1),
                lb(Y1, Y2),
                term(X1, Y2, Y3),
                rb(Y3, Y4).
```

(F2)
```
function(X0, Y0, Y6) ←
                gleich(X0,arity∼2.. arguments∼[X1, X2]),
                funcsym(X0, Y0, Y1),
                lb(Y1, Y2),
                term(X1, Y2, Y3),
                comma(Y3, Y4),
                term(X2, Y4, Y5),
                rb(Y5, Y6).
```

(F3)
```
function(X0, Y0, Y8) ←
                gleich(X0,arity∼3.. arguments∼[X1,X2,X3]),
                funcsym(X0, Y0, Y1),
                lb(Y1, Y2),
                term(X1, Y2, Y3),
                comma(Y3, Y4),
                term(X2, Y4, Y5),
                comma(Y5, Y6),
                term(X3, Y6, Y7),
                rb(Y7, Y8).
```

Für *Formeln*:

(E1)
```
expr(X0, Y0, Y1) ←
                gleich(X0,arity∼0),
                relationsymb(X0, Y0, Y1).
```

(E2)
```
expr(X0, Y0, Y4) ←
                gleich(X0,arity∼1.. arguments∼[X1]),
                relationsymb(X0, Y0, Y1),
                lb(Y1, Y2),
                term(X1, Y2, Y3),
                rb(Y3, Y4).
```

(E3)
```
expr(X0, Y0, Y6) ←
                gleich(X0,arity∼2.. arguments∼[X1, X2]),
                relationsymb(X0, Y0, Y1),
                lb(Y1, Y2),
                term(X1, Y2, Y3),
                comma(Y3, Y4),
                term(X2, Y4, Y5),
                rb(Y5, Y6).
```

(E4)
```
expr(X0, Y0, Y3) ←
        gleich(X0, arity∼2.. arguments∼[X1,X2]),
        term(X1, Y0, Y1),
        relationsymb(X0, Y1, Y2),
        term(X2, Y2, Y3).
```

(E5)
```
expr(X0, Y0, Y4) ←
        gleich(X0, arity∼1.. arguments∼[X1]),
        lb(Y0, Y1),
        logsym(X0, Y1, Y2),
        expr(X1, Y2, Y3),
        rb(Y3, Y4).
```

(E6)
```
expr(X0, Y0, Y5) ←
        gleich(X0,arity∼2.. arguments∼[X1,X2]),
        lb(Y0, Y1),
        expr(X1, Y1, Y2),
        logsym(X0, Y2, Y3),
        expr(X2, Y3, Y4),
        rb(Y4,Y5).
```

(E7)
```
expr(X0, Y0, Y3) ←
        gleich(X0,arity∼2.. arguments∼[X1, X2]),
        quantifier(X0, Y0, Y1),
        variable_list(X1, Y1, Y2),
        expr(X2, Y2, Y3).
```

(E8)
```
expr(X0, Y0, Y3) ←
        lb(Y0, Y1),
        expr(X0, Y1, Y2),
        rb(Y2, Y3).
```

Für *Variablenlisten* nach den Quantoren:

(V1)
```
variable_list([X], Y0, Y1) ←
        variable(X, Y0, Y1).
```

(V2)
```
variable_list([X|Xs], Y0, Y2) ←
        variable(X, Y0, Y1),
        variable_list(Xs, Y1, Y2).
```

6.4 Terminierung

In diesem Abschnitt zeigen wir, dass alle LD-Ableitungen des Programms `EXPRESSION` und der in Naproche benutzten definiten Ziele enden. Hierzu verwenden wir die in 5.2 vorgestellten Methoden.

Hier und in den folgenden Abschnitten stehen die Variablen $\mathtt{x}, \mathtt{y}, \mathtt{x}_0, \mathtt{y}_0, \ldots$ für beliebige Prologterme und die Variablen $\mathtt{X}, \mathtt{Y}, \mathtt{X}_0, \mathtt{Y}_0, \ldots$ für Prologvariablen.
Wir beginnen mit folgender Definition:

Definition 6.5 (Länge einer Liste) *Für alle Grundterme definieren wir die Funktion* $|\ |$*, genannt* Listenlänge*, induktiv durch*

$$|[x|xs]| = |xs| + 1$$

$$|f(x_1, \ldots, x_n)| = 0 \text{ wenn } f \neq [\ .\ |\ .\ ].$$

Insbesondere gilt: $|[\ ]| = 0$.

Man sieht leicht, dass diese Funktion jeder Liste die Anzahl ihrer Elemente zuordnet.

Wenn nicht anders angegeben, sei mit $|\ |$ angewendet auf Terme die Funktion Listenlänge gemeint. Wie in Abschnitt 5.2 bezeichne rel(P) die Menge aller Relationssymbole, die das Programm P enthält.

Wir betrachten folgende Interpretation des Programms `EXPRESSION`:

$$\begin{array}{rcl} \mathfrak{I}_{\text{expr}} & := & \{\mathtt{r}(\mathtt{x}_1, \ldots, \mathtt{x}_n) | \mathtt{r} \in \text{rel}(\texttt{EXPRESSION}) \setminus \{\texttt{gleich}\} \text{ und } |\mathtt{x}_{n-1}| > |\mathtt{x}_n|\} \\ & \cup & \text{grund}(\texttt{gleich(X,Y)}) \end{array}$$

Für diese gilt:

Lemma 6.6 $\mathfrak{I}_{\text{expr}}$ *ist ein Modell von* `EXPRESSION`*.*

Beweis Für die Relation `gleich` ist die Behauptung klar. Für die anderen Relationen betrachten wir die letzten beiden Stellen. Die Schreibweise $\mathtt{r}(\ldots, \mathtt{Y}, \mathtt{Z})$ stehe hierbei für $\mathtt{r}(\mathtt{Y}, \mathtt{Z})$ oder $\mathtt{r}(\mathtt{X}, \mathtt{Y}, \mathtt{Z})$ abhängig von der Stellenanzahl der Relation `r`.

- Jede Alphabetsklausel $\mathtt{a}(\ldots, \mathtt{X}, \mathtt{Y})$ ist von der Form $\mathtt{a}(\ldots, [\mathtt{zeichen}|\mathtt{Y}], \mathtt{Y})$. Damit gilt $|\mathtt{x}| = |[\mathtt{zeichen}|\mathtt{y}]| = |\mathtt{y}| + 1 > |\mathtt{y}|$.

- Alle anderen Klauseln aus `EXPRESSION` sind von der Form:

$$\begin{array}{rl} \mathtt{r}(\ldots,\mathtt{Y}_0,\mathtt{Y}_n) \leftarrow & \mathtt{gleich}(\mathtt{X},\mathtt{arity} \sim m..\ \mathtt{arguments} \sim [\mathtt{X}_1,\ldots,\mathtt{X}_m]),\\ & \mathtt{r}_1(\ldots,\mathtt{Y}_0,\mathtt{Y}_1),\\ & \mathtt{r}_2(\ldots,\mathtt{Y}_1,\mathtt{Y}_2),\\ & \vdots\\ & \mathtt{r}_n(\ldots,\mathtt{Y}_{n-1},\mathtt{Y}_n). \end{array}$$

bzw.

$$\begin{array}{rl} \mathtt{r}(\ldots,\mathtt{Y}_0,\mathtt{Y}_n) \leftarrow & \mathtt{r}_1(\ldots,\mathtt{Y}_0,\mathtt{Y}_1),\\ & \mathtt{r}_2(\ldots,\mathtt{Y}_1,\mathtt{Y}_2),\\ & \vdots\\ & \mathtt{r}_n(\ldots,\mathtt{Y}_{n-1},\mathtt{Y}_n). \end{array}$$

mit $n, m \in \mathbb{N}$ und $\mathtt{r}_i \in \mathrm{rel}(\mathtt{EXPRESSION}) \setminus \{\mathtt{gleich}\}$ für $i \in [1, n]$.

Gilt nun $|\mathtt{y}_i| > |\mathtt{y}_{i+1}|$ für $i \in [0, n-1]$, so folgt $|\mathtt{y}_0| > |\mathtt{y}_n|$.

□

Mit diesem Modell können wir nun zeigen:

Satz 6.7 *Das Programm* EXPRESSION *ist semi-akzeptabel.*

Beweis Wir definieren folgende Niveau-Abbildung | |:

$$\begin{array}{rcl} |\mathtt{gleich}(\mathtt{x},\mathtt{y})| & = & 0\\ |\mathtt{expr}(\mathtt{x},\mathtt{y},\mathtt{z})| & = & |\mathtt{y}|+1\\ |\mathtt{term}(\mathtt{x},\mathtt{y},\mathtt{z})| & = & |\mathtt{y}|+1\\ |\mathtt{r}(\ldots,\mathtt{x},\mathtt{y})| & = & |\mathtt{x}| \end{array}$$

für $\mathtt{r} \in \mathrm{rel}(\mathtt{EXPRESSION}) \setminus \{\mathtt{gleich}, \mathtt{expr}, \mathtt{term}\}$.

Wir zeigen, dass EXPRESSION bezüglich dieser Abbildung und des Modells $\mathfrak{I}_{\mathrm{expr}}$ semi-akzeptabel ist. Für alle Fakten ist dies klar.

Für die Regeln verwenden wir zunächst die allgemeine Darstellung aus dem letzten Beweis. Es gilt $|\mathtt{r}(\ldots \mathtt{s}_1, \mathtt{s}_2)| \geq |\mathtt{gleich}(\mathtt{t}_1, \mathtt{t}_2)| = 0$ und $\mathtt{r} \not\sim \mathtt{gleich}$ für alle $\mathtt{r} \in \mathrm{rel}(\mathtt{EXPRESSION}) \setminus \{\mathtt{gleich}\}$ und alle Grundterme $\mathtt{s}_1, \mathtt{s}_2, \mathtt{t}_1, \mathtt{t}_2$. Deswegen lassen wir in unserer weiteren Betrachtung, falls vorhanden, das Atom $\mathtt{gleich}(\mathtt{x}, \mathtt{arity} \sim m..\ \mathtt{arguments} \sim [\mathtt{x}_1, \ldots, \mathtt{x}_m])$ im Körper der Klausel weg. Damit ist jede Klausel von der Form:

$$\begin{array}{rl} \mathtt{r}(\ldots,\mathtt{Y}_0,\mathtt{Y}_n) \leftarrow & \mathtt{r}_1(\ldots,\mathtt{Y}_0,\mathtt{Y}_1),\\ & \mathtt{r}_2(\ldots,\mathtt{Y}_1,\mathtt{Y}_2),\\ & \vdots\\ & \mathtt{r}_n(\ldots,\mathtt{Y}_{n-1},\mathtt{Y}_n). \end{array}$$

mit $n \in \mathbb{N}$ und $\mathtt{r}_i \in \text{rel}(\mathtt{EXPRESSION}) \setminus \{\mathtt{gleich}\}$ für $i \in [1, n]$.

Betrachten wir für die Atome nun die Werte unter $|\ |$, stellen wir zunächst fest:

- Für $i \in [3, n]$ folgt aus $\mathfrak{I}_{\text{expr}} \models \mathtt{r}_j(\ldots, \mathtt{y}_{j-1}, \mathtt{y}_j)$ für $0 < j < i$:
$$\begin{aligned} |\mathtt{r}(\ldots, \mathtt{y}_0, \mathtt{y}_n)| &\geq |\mathtt{y}_0| \\ &> |\mathtt{y}_1| \\ &\geq |\mathtt{y}_2| + 1 \\ &\vdots \\ &\geq |\mathtt{y}_{i-1}| + 1 \\ &\geq |\mathtt{r}_i(\ldots, \mathtt{y}_{i-1}, \mathtt{y}_i)| \end{aligned}$$
- Für i=2 folgt aus $\mathfrak{I}_{\text{expr}} \models \mathtt{r}_1(\ldots, \mathtt{y}_0, \mathtt{y}_1)$:
$$|\mathtt{r}(\ldots, \mathtt{y}_0, \mathtt{y}_n)| \geq |\mathtt{y}_0| \geq |\mathtt{r}_2(\ldots, \mathtt{y}_1, \mathtt{y}_2)|$$

Es bleibt zu zeigen, dass aus

$$\mathfrak{I}_{\text{expr}} \models \mathtt{r}_j(\ldots, \mathtt{y}_{j-1}, \mathtt{y}_j)$$

für $0 < j < i$ folgt:

(i) Wenn $i = 1, 2$ und $\mathtt{r} \simeq \mathtt{r}_i$: $|\mathtt{r}(\ldots, \mathtt{y}_0, \mathtt{y}_n)| > |\mathtt{r}_i(\ldots, \mathtt{y}_{i-1}, \mathtt{y}_i)|$.

(ii) Wenn $\mathtt{r} \sqsupset \mathtt{r}_1$: $|\mathtt{r}(\ldots, \mathtt{y}_0, \mathtt{y}_n)| \geq |\mathtt{r}_1(\ldots, \mathtt{y}_0, \mathtt{y}_1)|$.

Hierzu betrachten wir die ersten beiden Körperatome aller Regeln einzeln:

a) *Terme*: Die Klauseln sind von der Form:

$$\mathtt{term(X_0,Y_0,Y_1)} \leftarrow \mathtt{r(X_0,Y_0,Y_1)}\,.$$

mit $\mathtt{r} \in \text{rel}(\mathtt{EXPRESSION}) \setminus \{\mathtt{gleich}, \mathtt{term}, \mathtt{expr}\}$.
Damit gilt: $|\mathtt{term(x_0,y_0,y_1)}| = |\mathtt{y_0}| + 1 > |\mathtt{y_0}| = |\mathtt{r(x_0,y_0,y_1)}|$

b) *Funktionen*: Die Klauseln sind von der Form:

$$\begin{aligned} \mathtt{function}(\mathtt{X_0}, \mathtt{Y_0}, \mathtt{Y_n}) \leftarrow\ & \mathtt{funcsym}(\mathtt{X_0}, \mathtt{Y_0}, \mathtt{Y_1}), \\ & \mathtt{lb}(\mathtt{Y_1}, \mathtt{Y_2}), \\ & \vdots \end{aligned}$$

Es gilt $\mathtt{function} \not\simeq \mathtt{funcsym}$, $\mathtt{function} \not\simeq \mathtt{lb}$ und $|\mathtt{function(x_0,y_0,y_1)}| = |\mathtt{y_0}| = |\mathtt{funcsym(x_0,y_0,y_1)}|$.

c) *Formeln*: Für Formelklauseln gilt:

$$|\mathtt{expr}(\mathtt{x_0}, \mathtt{y_0}, \mathtt{y_n})| = |\mathtt{y_0}| + 1 \geq |\mathtt{r_1}(\mathtt{x_0}, \mathtt{y_0}, \mathtt{y_1})|.$$

Die einzige Relation r aus EXPRESSION, für die expr $\simeq$ r gilt, ist expr selbst. An erster Stelle des Körpers kommt expr in keiner seiner Klauseln vor, an zweiter Stelle nur bei den Klauseln (E6) und (E8). Diese sind von der Form:

$$\begin{aligned}\texttt{expr}(\texttt{X}_0, \texttt{Y}_0, \texttt{Y}_n) \leftarrow\ & \texttt{lb}(\texttt{Y}_0, \texttt{Y}_1),\\ & \texttt{expr}(\ldots, \texttt{Y}_1, \texttt{Y}_2),\\ & \vdots\end{aligned}$$

Es folgt aus $\mathfrak{I}_{\text{expr}} \models \texttt{lb}(\texttt{y}_0, \texttt{y}_1)$:

$$|\texttt{expr}(\ldots, \texttt{y}_0, \texttt{y}_n)| = |\texttt{y}_0| + 1 > |\texttt{y}_1| + 1 = |\texttt{expr}(\ldots, \texttt{y}_1, \texttt{y}_2)|$$

d) *Variablenlisten*: Wir betrachten die Klauseln (V1) und (V2):

$$\texttt{variable\_list}([\texttt{X}], \texttt{Y}_0, \texttt{Y}_1) \leftarrow \texttt{variable}(\texttt{X}, \texttt{Y}_0, \texttt{Y}_1).$$

$$\begin{aligned}\texttt{variable\_list}([\texttt{X}|\texttt{Xs}], \texttt{Y}_0, \texttt{Y}_2) \leftarrow\ & \texttt{variable}(\texttt{X}, \texttt{Y}_0, \texttt{Y}_1),\\ & \texttt{variable\_list}(\texttt{Xs}, \texttt{Y}_1, \texttt{Y}_2).\end{aligned}$$

Es gilt variable_list $\not\simeq$ variable und

$$|\texttt{variable\_list}(\ldots, \texttt{y}_0, \texttt{y}_n)| = |\texttt{y}_0| = |\texttt{variable}(\ldots, \texttt{y}_0, \texttt{y}_1)|.$$

Außerdem folgt aus $\mathfrak{I}_{\text{expr}} \models \texttt{variable}(\texttt{x}, \texttt{y}_0, \texttt{y}_1)$:

$$|\texttt{variable}(\ldots, \texttt{y}_0, \texttt{y}_1)| = |\texttt{y}_0| > |y_1| = |\texttt{variable\_list}(\ldots, \texttt{y}_1, \texttt{y}_2)|.$$

Damit sind alle Klauseln aus EXPRESSION semi-akzeptabel. Es folgt die Semi-Akzeptabilität des Programms. □

Das Programm EXPRESSION übersetzt Terme und Formeln in die dazugehörigen Merkmalslisten. Definite Ziele, die für EXPRESSION verwendet werden, sind von der Form ←expr(X,y,Z) oder ←term(X,y,Z), wobei X und Y Prologvariablen sind und y ein Grundterm in Prolog. Beide Ziele sind bezüglich der Abbildung | | beschränkt. Es folgt mittels Korollar 5.16:

Korollar 6.8 *Alle LD-Ableitungen von* EXPRESSION $\cup$ {← expr(X,y,Z)} *und* EXPRESSION $\cup$ {← term(X,y,Z)} *sind endlich.*

Dieses Korollar zeigt, dass für die Formelgrammatik und die in Naproche verwendeten definiten Ziele alle LD-Ableitungen endlich sind. Für SLD-Resolutionen im Allgemeinen ist dies nicht der Fall, wie man sich leicht an der Klausel

expr(X0,Y0,Y3) ← lb(Y0,Y1),expr(X0,Y1,Y2),rb(Y2,Y3)

verdeutlichen kann. Geht man hier von einer Auswahlregel aus, die, wenn vorhanden, das zweite Atom aus dem Körper der Klausel wählt, kann sich diese Klausel beliebig oft selbst aufrufen.

6.5 Occur-Check

Wir zeigen in diesem Abschnitt, dass das Occur-Check-Problem im Programm `EXPRESSION` nicht auftritt. Hierzu betrachten wir folgendes moding für $\mathtt{r} \in \text{rel}(\mathtt{EXPRESSION})$:
Wenn `r` dreistellig:

$$\mathtt{r}(+,+,-)$$

Wenn `r` zweistellig:

$$\mathtt{r}(+,-)$$

Damit gilt nun folgender Satz:

Satz 6.9 *Das Programm* `EXPRESSION` *ist nicely moded.*

Beweis Alle Fakten sind laut Definition 5.32 nicely moded. Wir betrachten die Regeln und beginnen mit den Klauseln für Terme, Formeln und Funktionen. Hierzu variieren wir die Schreibweise aus den vorherigen Beweisen. Die Klauseln haben die Form:

$$\begin{aligned} \mathtt{r}_0(\mathtt{X}_0,\mathtt{Y}_0,\mathtt{Y}_1) \quad \leftarrow \quad & \mathtt{gleich}(\mathtt{X}_0,\mathtt{arity} \sim m..\mathtt{arguments} \sim [\mathtt{X}_1,\ldots,\mathtt{X}_m]),\\ & \mathtt{r}_1(f(\mathtt{r}_1),\mathtt{Y}_0,\mathtt{Y}_1),\\ & \vdots\\ & \mathtt{r}_n(f(\mathtt{r}_n),\mathtt{Y}_{n-1},\mathtt{Y}_n). \end{aligned}$$

bzw.

$$\begin{aligned} \mathtt{r}_0(\mathtt{X}_0,\mathtt{Y}_0,\mathtt{Y}_1) \quad \leftarrow \quad & \mathtt{r}_1(f(\mathtt{r}_1),\mathtt{Y}_0,\mathtt{Y}_1),\\ & \vdots\\ & \mathtt{r}_n(f(\mathtt{r}_n),\mathtt{Y}_{n-1},\mathtt{Y}_n). \end{aligned}$$

Es seien $m,n \in \mathbb{N}$ und $f : \text{rel}(\mathtt{EXPRESSION}) \to \{\mathtt{X}_0,\ldots,\mathtt{X}_m\} \cup \{\emptyset\}$ eine Funktion, die jedem Relationssymbol aus `EXPRESSION` abhängig von seiner Stellenanzahl entweder die leere Menge oder den entsprechenden Wert $\mathtt{X}_j$, $j \in [0,m]$ in der Klausel zuweist. Für zweistellige Relationen $\mathtt{r}_i$ verstehen wir die Schreibweise $\mathtt{r}_i(\emptyset,\mathtt{Y}_{i-1},\mathtt{Y}_i)$ als das zweistellige Atom $\mathtt{r}_i(\mathtt{Y}_{i-1},\mathtt{Y}_i)$.
Um die Eingabe- bzw. Ausgabeterme jedes Relationssymbols $p \in \text{rel}(\mathtt{EXPRESSION})$ unterscheiden zu können, benutzen wir die Notation aus Definition 5.32. Steht p in einer Klausel an i-ter Stelle, bezeichne der Ausdruck $\mathbf{s}_i$ die Liste der Eingabe- und der Ausdruck $\mathbf{t}_i$ die Liste der Ausgabeterme für p.

Für Klauseln der ersten Form gilt:

- Die Termfamilie $\mathtt{arity} \sim m..\mathtt{arguments} \sim [\mathtt{X}_1,\ldots,\mathtt{X}_m],\mathtt{Y}_1,\ldots,\mathtt{Y}_n$ ist linear.
- $\text{var}(\mathbf{s}_0) \cap (\bigcup_{j=1}^{n+1} \text{var}(\mathbf{t}_j)) = \{\mathtt{X}_0,\mathtt{Y}_0\} \cap \{\mathtt{X}_1,\ldots,\mathtt{X}_m,\mathtt{Y}_1,\ldots,\mathtt{Y}_n\} = \emptyset$
- $\text{var}(\mathbf{s}_1) \cap (\bigcup_{j=1}^{n+1} \text{var}(\mathbf{t}_j)) = \{\mathtt{X}_0\} \cap \{\mathtt{X}_1,\ldots,\mathtt{X}_m,\mathtt{Y}_1,\ldots,\mathtt{Y}_n\} = \emptyset$

- Für $i \in [2, n+1]$:
  $\text{var}(\mathbf{s}_i) \cap (\bigcup_{j=i}^{n+1} \text{var}(\mathbf{t}_j)) = (\{\mathtt{Y}_{i-2}\} \cup \text{var}(f(\mathbf{r}_{i-1}))) \cap \{\mathtt{Y}_{i-1} \ldots \mathtt{Y}_n\} = \emptyset$

Für Klauseln der zweiten Form gilt entsprechend:

- Die Termfamilie $\mathtt{Y}_1, \ldots, \mathtt{Y}_n$ ist linear.
- $\text{var}(\mathbf{s}_0) \cap (\bigcup_{j=1}^{n+1} \text{var}(\mathbf{t}_j)) = \{\mathtt{X}_0, \mathtt{Y}_0\} \cap \{\mathtt{Y}_1 \ldots \mathtt{Y}_n\} = \emptyset$
- Für $i \in [1, n]$: $\text{var}(\mathbf{s}_i) \cap (\bigcup_{j=i}^{n} \text{var}(\mathbf{t}_j)) = (\{\mathtt{Y}_{i-1}\} \cup \text{var}(f(\mathbf{r}_i))) \cap \{\mathtt{Y}_i \ldots \mathtt{Y}_n\} = \emptyset$

Damit sind die Klauseln für Formeln, Terme und Funktionen nicely moded. Es bleibt zu zeigen, dass die Klauseln für die Variablenlisten nicely moded sind. Hierzu betrachten wir die entsprechenden Klauseln:

(V1) `variable_list([X], Y0, Y1)` ← `variable(X, Y0, Y1).`

(V2) `variable_list([X|Xs], Y0, Y2)` ← `variable(X, Y0, Y1),`
`variable_list(Xs, Y1, Y2).`

Für (V1) gilt:

- $\text{var}(\mathbf{s}_0) \cap \text{var}(\mathbf{t}_1) = \{\mathtt{X}, \mathtt{Y}_0\} \cap \{\mathtt{Y}_1\} = \emptyset$
- $\text{var}(\mathbf{s}_1) \cap \text{var}(\mathbf{t}_1) = \{\mathtt{X}, \mathtt{Y}_0\} \cap \{\mathtt{Y}_1\} = \emptyset$

Für (V2) gilt:

- $\text{var}(\mathbf{s}_0) \cap (\text{var}(\mathbf{t}_1) \cup \text{var}(\mathbf{t}_2)) = \{\mathtt{X}, \mathtt{Xs}, \mathtt{Y}_0\} \cap (\{\mathtt{Y}_1\} \cup \{\mathtt{Y}_2\}) = \emptyset$
- $\text{var}(\mathbf{s}_1) \cap (\text{var}(\mathbf{t}_1) \cup \text{var}(\mathbf{t}_2)) = \{\mathtt{X}, \mathtt{Y}_0\} \cap (\{\mathtt{Y}_1\} \cup \{\mathtt{Y}_2\}) = \emptyset$
- $\text{var}(\mathbf{s}_2) \cap \text{var}(\mathbf{t}_2) = \{\mathtt{Xs}, \mathtt{Y}_1\} \cap \{\mathtt{Y}_2\} = \emptyset$

Damit sind auch die Klauseln für die Variablenlisten nicely moded. Es folgt Nicely-Modedness für das Programm `EXPRESSION`.

□

Daraus folgt nun:

Korollar 6.10 *Für die Terme* `x`, `y`, `z` *gelte* $\text{var}(\mathtt{x}, \mathtt{y}) \cap \text{var}(\mathtt{z}) = \emptyset$. *Dann sind* `EXPRESSION` $\cup \{\leftarrow$ `expr(x,y,z)`$\}$ *und* `EXPRESSION` $\cup \{\leftarrow$ `term(x,y,z)`$\}$ *occur-check-frei.*

Beweis Wir benutzen Korollar 5.35. Das Programm `EXPRESSION` und die definiten Ziele ←`expr(x,y,z)` und ←`term(x,y,z)` sind unter den oben genannten Bedingungen nicely moded. Es bleibt zu zeigen, dass der Kopf jeder Klausel aus `EXPRESSION` Eingabe-linear ist.
In den Alphabetsklauseln und der Klausel für die Relation `gleich` enthalten die Eingabepositionen je nur ein einziges Variablensymbol.

Für die Klauseln für Formeln, Terme und Funktionen sind, nach der Notation aus dem letzten Beweis, die Köpfe von der Form: $\mathtt{r}_0(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_n)$ und es gilt: $\mathtt{X}_0 \neq \mathtt{Y}_0$.
Die Köpfe der Klauseln für die Variablenlisten sind `variable_list([X0], Y0, Y1)` und `variable_list([X0|Xs], Y0, Y2)` und die Termfamilien $[\mathtt{X0}], \mathtt{Y0}$ und $[\mathtt{X0}|\mathtt{Xs}], \mathtt{Y0}$ sind linear. □

In Naproche werden zum Aufruf von `EXPRESSION` definite Ziele der Form ←`expr(X,ausdruck,[ ])` bzw. ←`term(X,ausdruck,[ ])` verwendet, wobei mit `ausdruck` mathematische Terme oder Formeln aus dem Eingabeformat von Naproche gemeint sind. Weil die Ausgabepositionen dieser Klauseln keine Variablen enthalten, tritt bei dieser Benutzung des Programms das Occur-Check-Problem nicht auf.

6.6 Semantik

In diesem Abschnitt bestimmen wir die logischen Konsequenzen des Programms `EXPRESSION`. Wir gehen wie in Abschnitt 5.4 beschrieben vor.
Um die Definition der Post-Assertionen zu erleichtern, erweitern wir den Begriff der Merkmalsliste aus Definition 6.4 auf die Symbole des Alphabets $\mathcal{A}_N$. Hierbei sei *name* weiterhin eine eindeutige Abbildung, die jedem Symbol aus dem Alphabet einen Namen zuordnet. Die Funktionswerte seien hierbei genau die in den Alphabetsklauseln angegebenen Prologkonstanten. Außerdem seien $arity : \mathcal{A}_N \to \mathbb{N}$ und $type : \mathcal{A}_N \to \{\mathtt{variable}, \mathtt{logical\_symbol}, \mathtt{quantifier}, \mathtt{relation}, \mathtt{function}, \mathtt{constant}\}$ Abbildungen, die jedem Symbol aus dem Alphabet seine Wertigkeit bzw. seine Symbolart gemäß Definition 6.1 zuweisen. Den Hilfssymbolen werde keine Merkmalsliste zugeordnet, für alle anderen Symbole s sei die Merkmalsliste von der Form:

$$M(s) = \begin{bmatrix} name & : & name(s) \\ type & : & type(s) \\ arity & : & arity(s) \end{bmatrix}$$

Wir notieren durch „$*$“ die Konkatenation zweier Listen $[s_1, \ldots, s_m]$ und $[t_1, \ldots, t_n]$ mit $m, n \geq 0$. Es gelte:

$$[s_1, \ldots, s_m] * [t_1, \ldots, t_n] = [s_1, \ldots, s_m, t_1, \ldots, t_n]$$

Um eine Spezifikation festlegen zu können, müssen wir außerdem zwischen Termen, Formeln, Variablen usw. aus dem Eingabeformat und diesen Begriffen im Programmcode unterscheiden. Hierzu belegen wir in diesem Abschnitt, wenn das Gemeinte nicht aus dem Zusammenhang hervorgeht, Bezeichnungen dieser Art, die sich auf das Eingabeformat von Naproche beziehen, mit der Vorsilbe Naproche-. Wir sprechen also von Naproche-Termen, Naproche-Formeln, Naproche-Variablen usw.

An dieser Stelle sei daran erinnert, dass das Eingabeformat von `EXPRESSION` Listen von Prolog-Konstanten sind. Der Term $f(x)$ entspricht in dieser Schreibweise der Liste `[f,'(',x,')']`. Im Folgenden behandeln wir diese beiden Ausdrücke gleichwertig. Ist `t` eine Liste, so meinen wir mit $\mathtt{t} \in \mathcal{T}_N$, dass der Ausdruck, den man erhält, indem man die Elemente von `t` hintereinandersetzt, ein Naproche-Term ist. Ebenso bezeichnen wir, wenn `t` eine Liste ist, mit $M(\mathtt{t})$ die Merkmalsliste des zu `t` gehörenden Ausdrucks.

Damit können wir nun eine Spezifikation für `EXPRESSION` festlegen:

(i) Für die Relation `gleich` sei dies:

$$\text{pre}_{\texttt{gleich}} = \{\texttt{gleich(x,y)}|\ \texttt{x} \text{ ist ein Term}\}$$

$$\text{post}_{\texttt{gleich}} = \{\texttt{gleich(x,y)}|\ \texttt{x}, \texttt{y} \text{ sind Terme, so dass } \texttt{x=y}\}$$

(ii) Für alle weiteren zweistelligen Relationen `r(x,y)` aus `EXPRESSION` sei die Prä-Assertion:

$$\text{pre}_{\texttt{r}} = \{\texttt{r(x,y)}|\ \texttt{x} \text{ ist eine Liste}\}$$

und die Post-Assertionen:

$$\begin{aligned}
\text{post}_{\texttt{comma}} &= \{\texttt{comma(x,y)}|\ \texttt{x}, \texttt{y} \text{ sind Listen, so dass } \texttt{x=[','|y]}\} \\
\text{post}_{\texttt{lb}} &= \{\texttt{lb(x,y)}|\ \texttt{x}, \texttt{y} \text{ sind Listen, so dass } \texttt{x=['('|y]}\} \\
\text{post}_{\texttt{rb}} &= \{\texttt{rb(x,y)}|\ \texttt{x}, \texttt{y} \text{ sind Listen, so dass } \texttt{x=[')'|y]}\}
\end{aligned}$$

(iii) Für die dreistelligen Relationen `r(x,y,z)` aus `EXPRESSION` sei die Prä-Assertion:

$$\text{pre}_{\texttt{r}} = \{\texttt{r(x,y,z)}|\ \texttt{y} \text{ eine Liste}\}$$

und die Post-Assertionen:

a) Für die Alphabetsklauseln:

$$\begin{aligned}
\text{post}_{\texttt{variable}} &= \{\texttt{variable(x,y,z)}|\ \texttt{x} \text{ ist eine GULP-Liste und } \texttt{y}, \texttt{z} \text{ sind Listen, so dass } \texttt{y} = \texttt{t} * \texttt{z},\ \texttt{t} \in \mathcal{A}_N \text{ Variable und } \texttt{x} = M(\texttt{t})\} \\
\text{post}_{\texttt{const}} &= \{\texttt{const(x,y,z)}|\ \texttt{x} \text{ ist eine GULP-Liste und } \texttt{y}, \texttt{z} \text{ sind Listen, so dass } \texttt{y} = \texttt{t} * \texttt{z},\ \texttt{t} \in \mathcal{A}_N \text{ Konstante und } \texttt{x} = M(\texttt{t})\} \\
\text{post}_{\texttt{logsym}} &= \{\texttt{logsym(x,y,z)}|\ \texttt{x} \text{ ist eine GULP-Liste und } \texttt{y}, \texttt{z} \text{ sind Listen, so dass } \texttt{y} = \texttt{t} * \texttt{z},\ \texttt{t} \in \mathcal{A}_N \text{ Konnektionssymbol und } \texttt{x} = M(\texttt{t})\} \\
\text{post}_{\texttt{quantifier}} &= \{\texttt{quantifier(x,y,z)}|\ \texttt{x} \text{ ist eine GULP-Liste und } \texttt{y}, \texttt{z} \text{ sind Listen, so dass } \texttt{y} = \texttt{t} * \texttt{z},\ \texttt{t} \in \mathcal{A}_N \text{ Quantorensymbol und } \texttt{x} = M(\texttt{t})\}
\end{aligned}$$

$\text{post}_{\texttt{relationsymb}}$ = {relationsymb(x,y,z)| x ist eine GULP-Liste und y, z sind Listen, so dass $\texttt{y} = \texttt{t} * \texttt{z}$, $\texttt{t} \in \mathcal{A}_N$ Relationssymbol und $\texttt{x} = M(\texttt{t})$}

$\text{post}_{\texttt{funcsym}}$ = {funcsym(x,y,z)| x ist eine GULP-Liste und y, z sind Listen, so dass $\texttt{y} = \texttt{t} * \texttt{z}$, $\texttt{t} \in \mathcal{A}_N$ Funktionssymbol und $\texttt{x} = M(\texttt{t})$}

b) Für Formel-, Term- und Funktionsklauseln:

$\text{post}_{\texttt{expr}}$ = {expr(x,y,z)| x ist eine GULP-Liste und y, z sind Listen, so dass $\texttt{y} = \texttt{t} * \texttt{z}$, $\texttt{t} \in \mathcal{F}_N$ und $\texttt{x} = M(\texttt{t})$}

$\text{post}_{\texttt{term}}$ = {term(x,y,z)| x ist eine GULP-Liste und y, z sind Listen, so dass $\texttt{y} = \texttt{t} * \texttt{z}$, $\texttt{t} \in \mathcal{T}_N$ und $\texttt{x} = M(\texttt{t})$}

$\text{post}_{\texttt{function}}$ = {function(x,y,z)| x ist eine GULP-Liste und y, z sind Listen, so dass gilt $\texttt{y} = \texttt{t} * \texttt{z}$, $\texttt{t} \in \mathcal{T}_N$ ist eine Naproche-Funktion und $\texttt{x} = M(\texttt{t})$}

c) Für die die Variablen-Liste:

$\text{post}_{\texttt{variable\_list}}$ = {variable_list(x,y,z)|x, y, z Listen, so dass $\texttt{y} = [\texttt{v}_1, \ldots, \texttt{v}_n] * \texttt{z}$, $\texttt{v}_1, \ldots, \texttt{v}_n \in \mathcal{A}_N$ Variablensymbole und $\texttt{x} = [M(\texttt{v}_1), \ldots, M(\texttt{v}_n)]$}

Wir zeigen nun:

Satz 6.11 *Das Programm* EXPRESSION *ist well-asserted.*

Beweis Für jede Klausel $H \leftarrow B_1, \ldots, B_n$ aus EXPRESSION ist zu zeigen, dass für $j \in [1, n+1]$ gilt

$$\models \text{pre}(H), \text{post}(B_1), \ldots, \text{post}(B_{j-1}) \Rightarrow \text{pre}(B_j) \tag{6.1}$$

wobei $\text{pre}(B_{n+1}) := \text{post}(H)$.

Für die Klausel gleich(X,X). ist die Aussage klar. Für alle Alphabetsfakten folgt (6.1) direkt aus der Definition der Merkmalsliste. Für alle anderen Klauseln verwenden wir die allgemeine Schreibweise aus Beweis 6.6. Die Klauseln haben die Form:

$$\begin{aligned} \texttt{r}_0(\ldots, \texttt{Y}_0, \texttt{Y}_{n-1}) \leftarrow\ & \texttt{gleich}(\texttt{X}_0, \texttt{arity} \sim m.. \texttt{arguments} \sim [\texttt{X}_1, \ldots, \texttt{X}_m]), \\ & \texttt{r}_1(\ldots, \texttt{Y}_0, \texttt{Y}_1), \\ & \texttt{r}_2(\ldots, \texttt{Y}_1, \texttt{Y}_2), \\ & \vdots \\ & \texttt{r}_{n-1}(\ldots, \texttt{Y}_{n-2}, \texttt{Y}_{n-1}). \end{aligned}$$

für $n, m \in \mathbb{N}$, wobei das erste Atom im Körper der Klausel im Fall m=0 wegfällt und, wie in 6.6, die Punkte innerhalb der Atome entweder für einen Prolog-Term oder, wenn die Relation zweistellig ist, für die leere Menge stehen.

Es gilt

$$\models \text{pre}(\mathtt{r}_0(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_{n-1})) \Rightarrow \text{pre}(\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim m..\mathtt{arguments} \sim [\mathtt{X}_1, \ldots, \mathtt{X}_m]))$$

weil jede Substitution θ die Prologvariable $\mathtt{X}_0$ durch einen Term ersetzt.

Außerdem gilt für $i \in [1, n-1]$

$$\begin{aligned} \models\ & \text{pre}(\mathtt{r}_0(\ldots, \mathtt{Y}_0, \mathtt{Y}_{n-1})), \\ & \text{post}(\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim m..\mathtt{arguments} \sim [\mathtt{X}_1, \ldots, \mathtt{X}_m])), \\ & \text{post}(\mathtt{r}_1(\ldots, \mathtt{Y}_0, \mathtt{Y}_1)), \\ & \vdots \\ & \text{post}(\mathtt{r}_{i-1}(\ldots, \mathtt{Y}_{i-2}, \mathtt{Y}_{i-1})) \\ & \Rightarrow \text{pre}(\mathtt{r}_i(\ldots, \mathtt{Y}_{i-1}, \mathtt{Y}_i)) \end{aligned}$$

da aus $\mathtt{r}_{i-1}(\ldots, \mathtt{Y}_{i-2}, \mathtt{Y}_{i-1})\theta \in \text{post}_{\mathtt{r}_{i-1}}$ für eine Substitution θ folgt, dass $\mathtt{Y}_{i-1}\theta$ eine Liste ist. Damit gilt $\mathtt{r}_i(\ldots, \mathtt{Y}_{i-1}, \mathtt{Y}_i)\theta \in \text{pre}_{\mathtt{r}_i}$.

Es bleibt, Aussage 6.1 für $j = n+1$ zu zeigen.

- Für die *Termklauseln* gilt:

  $$\models \text{pre}(\mathtt{term}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)), \text{post}(\mathtt{r}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)) \Rightarrow \text{post}(\mathtt{term}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1))$$

  für $\mathtt{r} \in \{\mathtt{variable}, \mathtt{const}, \mathtt{function}\}$.
  Denn gilt für eine Substitution θ

  $$\mathtt{r}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)\theta \in \text{post}_{\mathtt{variable}} \cup \text{post}_{\mathtt{const}} \cup \text{post}_{\mathtt{function}},$$

  so sind $\mathtt{Y}_0\theta$ und $\mathtt{Y}_1\theta$ Listen mit $\mathtt{Y}_0\theta = \mathtt{t} * \mathtt{Y}_1\theta$ und $\mathtt{t}$ Naproche-Variable, Naproche-Konstante oder Naproche-Funktion. Also $\mathtt{t} \in \mathcal{T}_N$. Außerdem gilt $\mathtt{X}_0\theta = M(\mathtt{t})$. Damit folgt $\mathtt{term}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)\theta \in \text{post}_{\mathtt{term}}$.

- Für die *Funktionsklauseln* muss gezeigt werden:

  $$\begin{aligned} \models\ & \text{pre}(\mathtt{function}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_n)), \\ & \text{post}(\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim m..\mathtt{arguments} \sim [\mathtt{X}_1, \ldots, \mathtt{X}_m])), \\ & \text{post}(\mathtt{funcsym}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)), \\ & \text{post}(\mathtt{lb}(\mathtt{Y}_1, \mathtt{Y}_2)), \\ & \text{post}(\mathtt{term}(\mathtt{X}_1, \mathtt{Y}_2, \mathtt{Y}_3)), \\ & \vdots \\ & \text{post}(\mathtt{rb}(\mathtt{Y}_{n-1}, \mathtt{Y}_n)) \\ & \Rightarrow \text{post}(\mathtt{function}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_n)) \end{aligned}$$

  wobei die Auslassungspunkte, wenn $m > 1$, für $\text{post}(\mathtt{comma}(\mathtt{Y}_{2i-1}, \mathtt{Y}_{2i}))$, $\text{post}(\mathtt{term}(\mathtt{X}_i, \mathtt{Y}_{2i}, \mathtt{Y}_{2i+1}))$ mit $i \in [2, m]$ stehen.

Sei θ eine Substitution, für die die linke Seite der Implikation gelte. Damit sind $\mathtt{Y}_0\theta$ und $\mathtt{Y}_n\theta$ Listen und es gilt $\mathtt{Y}_0\theta = \mathtt{t}_1 * \ldots * \mathtt{t}_n * \mathtt{Y}_n\theta$ und die Liste $\mathtt{t}_1$ besteht aus einem Naproche-Funktionssymbol, $\mathtt{t}_2$ aus einer linken Klammer, $\mathtt{t}_3$ aus einem Naproche-Term, $\mathtt{t}_n$ aus einer rechten Klammer und, wenn $m > 1$, enthält weiterhin $\mathtt{t}_{2i}$ ein Komma und $\mathtt{t}_{2i+1}$ einen Naproche-Term für $i \in [2, m]$. Es gilt außerdem $\mathtt{X}_0\theta = (\mathtt{arity} \sim m..\mathtt{arguments} \sim [\mathtt{X}_1, \ldots, \mathtt{X}_m])\theta$ und $\mathtt{X}_0\theta = M(\mathtt{t}_1)$. Damit ist das Funktionssymbol in $\mathtt{t}_1$ m-stellig und die Liste $\mathtt{t} = \mathtt{t}_1 * \ldots * \mathtt{t}_n$ ist eine Naproche-Funktion.
Weil für die Naproche-Terme außerdem $\mathtt{X}_i\theta = M(\mathtt{t}_{2i+1})$ für $i \in [1, m]$ gilt, folgt:

$$\begin{aligned}\mathtt{X}_0\theta = \;& \mathtt{arity} \sim m..\\ & \mathtt{arguments} \sim [M(\mathtt{t}_3), \ldots, M(\mathtt{t}_{2m+1})]..\\ & \mathtt{name} \sim name(\mathtt{t}_1)..\\ & \mathtt{type} \sim function\end{aligned}$$

Damit ist $\mathtt{X}_0\theta = M(\mathtt{t})$. Also gilt $\mathtt{function}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_n)\theta \in \mathrm{post}_{\mathtt{function}}$.

- Wir betrachten die verschiedenen *Formelklauseln*:
  - Für die Relationsklauseln (E1)-(E4) untersuchen wir exemplarisch die Klausel (E1). Es gilt:

$$\begin{aligned}\models \;& \mathrm{pre}(\mathtt{expr}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)),\\ & \mathrm{post}(\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim 0)),\\ & \mathrm{post}(\mathtt{relationsymb}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1))\\ & \Rightarrow \mathrm{post}(\mathtt{expr}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1))\end{aligned}$$

    Sei θ eine Substitution, für die die linke Seite der Implikation erfüllt ist. Dann sind, wegen $\mathtt{relationsymb}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_1)\theta \in \mathrm{post}_{\mathtt{relationsymb}}$, $\mathtt{Y}_0\theta$ und $\mathtt{Y}_1\theta$ Listen, so dass $\mathtt{Y}_0\theta = \mathtt{t} * \mathtt{Y}_1\theta$, $\mathtt{t} \in \mathcal{A}_N$ Naproche-Relationssymbol, gilt und $\mathtt{X}_0\theta$ ist eine GULP-Liste, so dass $\mathtt{X}_0\theta = M(\mathtt{t})$ erfüllt ist. Wegen $\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim 0)\theta \in \mathrm{post}_{\mathtt{gleich}}$ besitzt des Relationssymbol $\mathtt{t}$ die Wertigkeit 0. Damit ist $\mathtt{t}$ eine Formel und das zu Zeigende ist erfüllt.
    Für (E2)-(E4) erfolgt der Beweis analog.
  - Für die Klauseln (E5) und (E6), die Formeln mit Konnektionssymbol behandeln, betrachten wir Klausel (E5):

$$\begin{aligned}\models \;& \mathrm{pre}(\mathtt{expr}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_4)),\\ & \mathrm{post}(\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim 1..\mathtt{arguments} \sim [\mathtt{X}_1])),\\ & \mathrm{post}(\mathtt{lb}(\mathtt{Y}_0, \mathtt{Y}_1)),\\ & \mathrm{post}(\mathtt{logsym}(\mathtt{X}_0, \mathtt{Y}_1, \mathtt{Y}_2)),\\ & \mathrm{post}(\mathtt{expr}(\mathtt{X}_1, \mathtt{Y}_2, \mathtt{Y}_3)),\\ & \mathrm{post}(\mathtt{rb}(\mathtt{Y}_3, \mathtt{Y}_4))\\ & \Rightarrow \mathrm{post}(\mathtt{expr}(\mathtt{X}_0, \mathtt{Y}_0, \mathtt{Y}_4))\end{aligned}$$

    Sei θ eine Substitution, für die die linke Seite der Implikation erfüllt ist. Dann sind $\mathtt{Y}_0\theta$ und $\mathtt{Y}_4\theta$ Listen, für die $\mathtt{Y}_0\theta = \mathtt{t}_1 * \mathtt{t}_2 * \mathtt{t}_3 * \mathtt{t}_4 * \mathtt{Y}_4\theta$ gilt.

Hierbei besteht die Liste $\mathtt{t}_1$ aus einer linken Klammer, $\mathtt{t}_2$ aus einem einstelligen Konnektionssymbol, $\mathtt{t}_3$ aus einer Naproche-Formel und $\mathtt{t}_4$ aus einer rechten Klammer. Damit ist $\mathtt{t} = \mathtt{t}_1 * \mathtt{t}_2 * \mathtt{t}_3 * \mathtt{t}_4$ eine Naproche Formel. Außerdem folgt aus $\mathtt{gleich}(\mathtt{X}_0, \mathtt{arity} \sim 1..\mathtt{arguments} \sim [\mathtt{X}_1])\theta \in \mathrm{post}_{\mathtt{gleich}}$, $\mathtt{logsym}(\mathtt{X}_0, \mathtt{Y}_1, \mathtt{Y}_2)\theta \in \mathrm{post}_{\mathtt{logsym}}$ und $\mathtt{expr}(\mathtt{X}_1, \mathtt{Y}_2, \mathtt{Y}_3)\theta \in \mathrm{post}_{\mathtt{expr}}$, dass die GULP-Liste $\mathtt{X}_0\theta$ die Eigenschaft $\mathtt{X}_0\theta = M(\mathtt{t})$ erfüllt.
Für (E6) erfolgt der Beweis analog.

– Für (E7) und (E8) kann das zu Zeigende ebenso wir für die vorherigen Klauseln bewiesen werden.

- Bezüglich der Klauseln für die *Variablenlisten* gilt:

$$\begin{aligned} \models\ & \mathrm{pre}(\mathtt{variable\_list}([\mathtt{X}], \mathtt{Y}_0, \mathtt{Y}_1)), \\ & \mathrm{post}(\mathtt{variable}(\mathtt{X}, \mathtt{Y}_0, \mathtt{Y}_1)) \\ & \Rightarrow \mathrm{post}(\mathtt{variable\_list}([\mathtt{X}], \mathtt{Y}_0, \mathtt{Y}_1)) \end{aligned}$$

Sei θ eine Substitution, so dass $\mathtt{variable}(\mathtt{X}, \mathtt{Y}_0, \mathtt{Y}_1)\theta \in \mathrm{post}_{\mathtt{variable}}$. Dann sind $\mathtt{Y}_0\theta$ und $\mathtt{Y}_1\theta$ Listen und es gilt $\mathtt{Y}_0\theta = \mathtt{t} * \mathtt{Y}_1\theta$, $\mathtt{t} \in \mathcal{A}_N$ Naproche-Variable, und $\mathtt{X}\theta = M(\mathtt{t})$. Daraus folgt $\mathtt{variable\_list}([\mathtt{X}], \mathtt{Y}_0, \mathtt{Y}_1)\theta \in \mathrm{post}_{\mathtt{variable\_list}}$.
Ebenso gilt:

$$\begin{aligned} \models\ & \mathrm{pre}(\mathtt{variable\_list}([\mathtt{X}|\mathtt{Xs}], \mathtt{Y}_0, \mathtt{Y}_2)), \\ & \mathrm{post}(\mathtt{variable}(\mathtt{X}, \mathtt{Y}_0, \mathtt{Y}_1)), \\ & \mathrm{post}(\mathtt{variable\_list}(\mathtt{Xs}, \mathtt{Y}_1, \mathtt{Y}_2)) \\ & \Rightarrow \mathrm{post}(\mathtt{variable\_list}([\mathtt{X}|\mathtt{Xs}], \mathtt{Y}_0, \mathtt{Y}_2)) \end{aligned}$$

Denn gilt für ein θ die linke Seite der Implikation, so ist die Liste $\mathtt{Y}_0\theta$ von der Form $\mathtt{Y}_0\theta = [\mathtt{t}_1] * [\mathtt{t}_2, \ldots, \mathtt{t}_n] * \mathtt{Y}_2\theta$, $\mathtt{t}_1, \ldots, \mathtt{t}_n \in \mathcal{A}_N$ Variablensymbole, und es gilt $\mathtt{X}\theta = M(\mathtt{t}_1)$ und $\mathtt{Xs}\theta = [M(\mathtt{t}_2), \ldots, M(\mathtt{t}_n)]$. Damit gilt $\mathtt{variable\_list}([\mathtt{X}|\mathtt{Xs}], \mathtt{Y}_0, \mathtt{Y}_1)\theta \in \mathrm{post}_{\mathtt{variable\_list}}$.

Damit sind alle Klauseln des Programms EXPRESSION well-asserted. □

Für den folgenden Satz sei post für EXPRESSION wie in Definition 5.49 definiert, also als die Vereinigung aller Grundinstanzen der Post-Assertionen aller Relationssymbole:

$$\mathrm{post} := \bigcup_{p \text{ in } \mathtt{EXPRESSION}} \mathrm{grund}(\mathrm{post}_p)$$

Wir können nun zeigen:

Satz 6.12 $\mathcal{M}_{(\mathrm{pre},\mathrm{post})}(\mathtt{EXPRESSION}) = \mathrm{post}$

Beweis Das Programm EXPRESSION ist akzeptabel und well-asserted, deswegen können wir Satz 5.53 anwenden. Wir zeigen:

$T_{\text{pre(EXPRESSION)}}(\text{post})$
$= \{H \in B_P | H \leftarrow B_1, \ldots, B_n \in \text{grund}(\text{pre}(\texttt{EXPRESSION})) \wedge \text{post} \models B_1, \ldots, B_n)\}$
$= \{H \in \text{pre} \,| H \leftarrow B_1, \ldots, B_n \in \text{grund}(\texttt{EXPRESSION}) \wedge \text{post} \models B_1, \ldots, B_n\}\}$
$= \text{post}$

Wegen der Well-Assertedness von `EXPRESSION` folgt für $H \leftarrow B_1, \ldots, B_n$ aus $\text{pre}(H)$ und $\text{post}(B_1), \ldots, \text{post}(B_n)$ auch $\text{post}(H)$, also gilt:

$$T_{\text{pre(EXPRESSION)}}(\text{post}) \subseteq \text{post}$$

Um $T_{\text{pre(EXPRESSION)}}(\text{post}) \supseteq \text{post}$ zu zeigen, betrachten wir die Elemente aus post. Sei $H \in \text{post}$. Wir unterscheiden folgende Fälle:

- Ist $H \in \text{grund}(\text{post}_{\texttt{gleich}})$, so ist es von der Form `gleich(x,y)`, `x` und `y` sind Grundterme und es gilt `x` $=$ `y`. Daraus folgt `gleich(x,y)` $\in \text{grund}(\text{pre}_{\texttt{gleich}})$ und weil die Gleichheitsrelation durch einen Fakt definiert wird gilt damit `gleich(x,y)` $\in T_{\text{pre(EXPRESSION)}}(\text{post})$.

- Ist $H \in \text{grund}(\text{post}_{\texttt{variable}})$. So ist $H =$ `variable`$(\texttt{x}, \texttt{y}, \texttt{z})$ ein Grundterm und `y` eine Liste. Es folgt, weil die Relation `variable` durch einen Fakt definiert wird, $H \in T_{\text{pre(EXPRESSION)}}(\text{post})$.
  Enthält H eine andere Alphabetsrelation, so erfolgt der Beweis analog.

- Ist $H \in \text{grund}(\text{post}_{\texttt{term}})$, so ist der Grundterm H von der Form `term`$(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z})$ und `t` $*$ `z` ist eine Liste, deswegen gilt $H \in$ pre. Weiterhin gilt `t` $\in \mathcal{T}_N$. Nach Definition 6.2 ist `t` damit entweder eine Naproche-Funktion, eine Naproche-Variable oder eine Naproche-Konstante und für `p` $\in \{$`term`, `function`, `constant`$\}$ gilt daher `p`$(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z}) \in$ post und `term`$(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z}) \leftarrow$ `p`$(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z}) \in \text{grund}(\texttt{EXPRESSION})$. Es folgt `term`$(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z}) \in T_{\text{pre(EXPRESSION)}}(\text{post})$.

- Ist $H \in \text{grund}(\text{post}_{\texttt{function}})$, so ist H von der Form `function`$(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z})$. H ist ein Grundterm und `t` $*$ `z` eine Liste, es folgt $H \in$ pre. Außerdem ist `t` eine Naproche-Funktion gemäß Definition 6.2. Damit besteht `t` aus einem n-stelligen Naproche-Funktionssymbol, einer linken Klammer, n vielen Termen, die je durch ein Komma getrennt sind, und einer rechten Klammer. Alle Symbole sind in $\mathcal{A}_N$ (vgl. Definition 6.1) enthalten. Damit ist das Funktionssymbol entweder einstellig, zweistellig oder dreistellig. Angenommen das Funktionssymbol ist einstellig. Dann ist `t` von der Form $\texttt{t} = \texttt{t}_1 * \texttt{t}_2 * \texttt{t}_3 * \texttt{t}_4$ mit $\texttt{t}_1$ Funktionssymbol, $\texttt{t}_2$ linke Klammer, $\texttt{t}_3$ Naproche-Term und $\texttt{t}_4$ rechte Klammer und $M(\texttt{t})$ ist von der Form:
  $$\begin{aligned} M(\texttt{t}) \;=\; & \texttt{name} \sim \mathit{name}(\texttt{t}_1).. \\ & \texttt{type} \sim \texttt{function}.. \\ & \texttt{arity} \sim 1.. \\ & \texttt{arguments} \sim [M(\texttt{t}_3)] \end{aligned}$$
  Nun betrachte die folgende Instanz der Funktionsklausel (F1):

$$\begin{aligned}\texttt{function}(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z}) \quad \leftarrow \quad & \texttt{gleich}(M(\texttt{t}), \texttt{arity} \sim 1.. \texttt{ arguments} \sim [M(\texttt{t}_3)]) \\ & \texttt{funcsym}(M(\texttt{t}_1), \texttt{t}_1 * \texttt{t}_2 * \texttt{t}_3 * \texttt{t}_4 * \texttt{z}, \texttt{t}_2 * \texttt{t}_3 * \texttt{t}_4 * \texttt{z}), \\ & \texttt{lb}(\texttt{t}_2 * \texttt{t}_3 * \texttt{t}_4 * \texttt{z}, \texttt{t}_3 * \texttt{t}_4 * \texttt{z}), \\ & \texttt{term}(M(\texttt{t}_3), \texttt{t}_3 * \texttt{t}_4 * \texttt{z}, \texttt{t}_4 * \texttt{z}), \\ & \texttt{rb}(\texttt{t}_4 * \texttt{z}, \texttt{z})\end{aligned}$$

Diese Instanz ist Element von grund(EXPRESSION) und alle atomaren Formeln des Körpers dieser Klausel sind in post enthalten. Es folgt $\texttt{function}(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z}) \in T_{\text{pre(EXPRESSION)}}(\text{post})$.
Für zwei- und dreistellige Funktionen erfolgt der Beweis analog.

- Ist $H \in \text{post}(\text{grund}_{\texttt{expr}})$ lässt sich die Behauptung auf die gleiche Weise wie für Funktionsklauseln zeigen. H ist dann von der Form $\texttt{expr}(M(\texttt{t}), \texttt{t} * \texttt{z}, \texttt{z})$ und t ist eine Naproche-Formel gemäß der Variation von Definition 6.3 und $M(\texttt{t})$ ist die zugehörige Merkmalsliste gemäß Definition 6.4. Für jeden Punkt dieser Definitionen lässt sich eine Grundinstanz einer Klausel aus (E1)-(E8) finden, deren Körperatome in post enthalten sind.
- Für $H \in \text{grund}_{\texttt{variable\_list}}$ erfolgt der Beweis ebenso wie für die vorherigen Fälle.

Es gilt also $T_{\text{pre(EXPRESSION)}}(\text{post}) = \text{post}$. □

Wir wollen die gefundenen Resultate nun auf die von uns betrachteten atomaren definiten Ziele der Form `term(X,y,Z)` und `expr(X,y,Z)`, mit `y` Grundliste, anwenden. Diese Ziele sind well-asserted. Außerdem sind grund(`term(X,y,Z)`) ∩ post und grund(`expr(X,y,Z)`) ∩ post endlich. Damit folgt:

Korollar 6.13 *Für definite Ziele der Form* ←`expr(X,y,Z)` *und* ←`term(X,y,Z)`*,* `y` *Grundliste,* `X`*,* `Z` *Variablen, gilt:*

$$\{\leftarrow \texttt{expr(X,y,Z)}\}\ \texttt{EXPRESSION}\ \text{grund}(\texttt{expr(X,y,Z)}) \cap \text{post}$$

und

$$\{\leftarrow \texttt{term(X,y,Z)}\}\ \texttt{EXPRESSION}\ \text{grund}(\texttt{term(X,y,Z)}) \cap \text{post}$$

Wir fassen die Ergebnisse der letzten Abschnitte zusammen. Damit alle bewiesenen Eigenschaften gelten, müssen die verwendeten definiten Ziele beschränkt und nicely moded sein. Außerdem müssen sie der in diesem Abschnitt beschriebenen Prä-Assertion genügen. Dies trifft für die definiten Ziele ← `expr(X,y,[]).` und ← `term(X,y,[]).` zu, wenn `y` eine Grundliste und `X` eine Variable ist. Dies ist sind genau die definiten Ziele, die im Naproche-System zum Aufruf von `EXPRESSION` verwendet werden. Damit arbeitet das Programm bezüglich der vorgestellten Punkte korrekt.

7 Prolog im Naproche-System

In den vorherigen Kapiteln haben wir Methoden kennengelernt, die Korrektheit von Prolog-Programmen zu überprüfen. Diese wurden dann exemplarisch auf die vereinfachte Version eines Programmes aus dem Naproche-System angewendet. An dieser Stelle behandeln wir die Möglichkeit, auf dem vorgestellten Wege die Korrektheit aller Programme des Systems zu beweisen.
Hierzu betrachten wir einzeln die wichtigsten im Naproche-System verwendeten Prologelemente[1], die über die in Kapitel 4 vorgestellte Programmiersprache, reines Prolog, hinausgehen und behandeln die Auswirkungen ihrer Verwendung auf Korrektheitsbeweise. Die Erklärungen beziehen sich hierbei auf die im Naproche-System verwendete Prolog-Implementation SWI-Prolog[2]. Grundlage für die Beschreibungen sind das SWI-Handbuch [Wie] und die Bücher von *Apt* [Apt97] und *Nilsson und Maluszynski* [NM95].

7.1 Operatoren

In Prolog werden Funktionen und Relationen in Präfix-Notation geschrieben, d.h. das Funktions- oder Relationssymbol steht vor seinen eingeklammerten Argumenten. Für manche Funktionen und Relationen ist es jedoch üblich, Infixschreibweisen bzw. Schreibweisen ohne Klammern zu verwenden. Hierzu zählen beispielsweise arithmetische Funktionen. Für diese ist meist festgelegt, wie das jeweilige Funktionssymbol bei mehrmaliger Anwendung und in Verbindung mit anderen Funktionssymbolen auszuwerten ist. So löst sich beispielsweise die Mehrdeutigkeit eines Ausdrucks der Form `-2+3`, der sonst entweder als `(-2)+3` oder als `-(2+3)` verstanden werden kann.
Funktionssymbole der oben beschriebenen Art nennt man *Operatoren*. Prolog ermöglicht dem Benutzer diese einzuführen und ihren syntaktischen Gebrauch festzulegen. Hierzu existiert der folgende Prolog-Befehl[3]:

```
:- op(Priorität, Typ, Name).
```

`Name` steht hierbei für den Namen des Operators. Die `Priorität` ist eine natürliche Zahl zwischen 0 und 1200, die jedem Operator seine *Bindungsstärke* zuweist. Je niedriger

[1] Zur Veranschaulichung ist ein Teil des Codes des Logik-Moduls von Naproche 0.2 an diese Arbeit angehängt, vgl. Anhang A.2.
[2] Informationen hierzu findet man auf der SWI-Homepage [SP].
[3] Wir schreiben diesen Befehl in seiner Prologschreibweise (d.h. mit `:-` statt ←), weil er keine Entsprechung in der logischen Programmierung besitzt.

die Zahl, desto stärker die Bindung. Besitzen die Infixoperatoren `f` die Priorität 10 und `g` die Priorität 20 und sind `s`, `t` und `u` Terme, so wird der Ausdruck `s f t g u` als `(s f t) g u` bzw. als `g(f(s,t),u)` verstanden.
Für `Typ` können die folgenden Typen eingesetzt werden: `xfx`, `xfy`, `yfx`, `fx`, `fy`, `xf`, `yf`. Der Buchstabe `f` steht für die Position des verwendeten Symbols, die Buchstaben `x` und `y` für die Position seiner Argumente. `y` wird verwendet, wenn die Priorität des Arguments kleiner oder gleich der Prioriät des Funktionssymbols, `x`, wenn diese echt größer sein soll. Ist `f` vom Typ `xfy`, so wird der Ausdruck `s f t f u` als `(s f t) f u` bzw. `f(f(s,t),u)` verstanden, ist `f` vom Typ `yfx` als `s f (t f u)` bzw. `f(s,f(t,u))`.
Alle Terme, denen kein Wert zugewiesen wurde oder die in Klammern stehen, besitzen Priorität 0.
Jeder Operator lässt sich ebenfalls in Präfixnotation schreiben.
Wegen der ambivalenten Syntax von Prolog können auch Relationssymbole als Operatoren verwendet werden. Hierbei ist jedoch zu beachten, dass die Argumente von Operatoren Terme sind. Somit ist für zweistellige Relations-Operatoren `xfx` als Typ zu wählen. Als Beispiel eines solchen Operators betrachten wir das Gleichheitszeichen in Prolog. Syntaktisch ist es folgendermaßen festgelegt[4]:

```
:- op(700, xfx, =).
```

Dieses Zeichen kann nicht hintereinander angewendet werden. Das definite Ziel ← `X = p(a)= Y.` führt im Gegensatz zu dem Ziel ← `X = p(a).` zu einem Fehler.
Auch im Naproche-System werden Operatoren deklariert. Da es sich bei der Einführung von Operatoren jedoch um rein syntaktische Veränderungen handelt und sich jeder Operator auch in Präfixnotation schreiben lässt, sind die vorgestellten Korrektheitsbeweise auch für reines Prolog mit eingeführten Operatoren möglich.

7.2 Arithmetik

In Abschnitt 4.1 wurde darauf hingewiesen, dass Prolog Zahlen als Konstanten behandelt. Weiterhin existieren, wie in den meisten Programmiersprachen, einige eingebaute arithmetische Funktionen und Relationen. Wir behandeln diese, soweit sie im Naproche-System verwendet werden. Deswegen können wir uns an dieser Stelle auf die Arithmetik ganzer Zahlen mit den Operationen „+“ und „−“, das Vorzeichen „−“ und die Relationen „>“ und „=“ beschränken.
Prolog verwendet arithmetische Funktionszeichen als Operatoren. Intern ist festgelegt:

```
:-  op(500, yfx, +).
:-  op(500, yfx, -).
:-  op(200, fy, -).
```

[4]Die hier und später angegebenen Werte sind jeweils die von SWI-Prolog verwendeten.

Die Menge der Terme, die man mittels der hier eingeführten Funktionssymbole, gemäß ihrer vorgeschriebenen Verwendung, und den Konstanten `0, -1, 1, -2, 2, ...` bilden kann, bezeichenen wir als *arithmetische Grundterme*.
Für die arithmetische Relation „>“ benutzt Prolog das Zeichen „`>`“, für „=“ wird das Zeichen „`=:=`“ verwendet. Als Operatoren sind sie wie folgt festgelegt:

```
:- op(700, xfx, >).
:- op(700, xfx, =:=).
```

Arithmetische Relationen sind in Prolog nach den üblichen Konventionen, jedoch nur für arithmetische Grundterme definiert. So gibt das Prolog-System für die definiten Ziele ← `5+3 > 1+2` und ← `1+2 > 5+3` die jeweiligen Ergebnisse, `true` bzw. `fail` aus, für ← `[] > 2` oder ← `X =:= 5+3` ergibt sich jedoch ein Fehler.
Um arithmetische Funktionen auszuwerten, besitzt Prolog einen weiteren Operator:

```
:- op(700, xfx, is).
```

Dieser vergleicht eine ganze Zahl oder eine Variable auf der linken Seite mit einem arithmetischen Grundausdruck auf der rechten Seite. Die Relation `is` ist erfüllt, wenn der ausgewertete Wert auf der rechten Seite dem Wert auf der linken Seite entspricht. Der Operator kann nicht in Verbindung mit anderen Termen verwendet werden. Für das definite Ziel ← `X is 5+3.` gibt das System `X = 8` als Lösung aus. Die Ziele ← `3+4 is 3+4.` und ← `Y is X+1.` führen zu Fehlern.
Das Naproche-System verwendet außerdem die Relation `succ/2`. Gilt für zwei Terme t_1 und t_2 die Relation `succ(`t_1`,`t_2`)`, so ist t_2 der Nachfolger von t_1, also t_1`+1 =` t_2.
Es finden sich nur wenige Stellen im Code des Naproche-Systems, an denen Arithmetik verwendet wird. Oft geschieht dies zur Nummerierung einzelner Elemente wie beispielsweise mathematischer Ausdrücke. Doch auch an diesen Stellen sollte die Korrektheit der Programme garantiert sein. Neben den Problemen, die sich für reine Prolog-Programme stellen, muss bei der Verwendung von Arithmetik auf den korrekten Gebrauch arithmetischer Relationen geachtet werden. Diese sind, wie beschrieben, nur für bestimmte Terme definiert. Einen Ansatz zur Lösung dieses Problems gibt *Apt* in [Apt97, Kap. 10]. Die dort beschriebenen Verfahren lassen sich größtenteils auf die entsprechenden Programme des Naproche-Systems übertragen.

7.3 Cut

Der Berechnungsprozess von Prolog kann sehr ineffizient sein. Wie wir in Abschnitt 4.3 gesehen haben, berechnet Prolog eine Antwort für ein Programm und ein Ziel, indem es der Reihe nach einen geordneten LD-Baum erzeugt und bewertet. Dieser kann unter Umständen sehr groß sein und nur wenige erfolgreiche Ableitungen enthalten. Um die Effizienz in solchen Fällen zu steigern, besitzt Prolog die eingebaute nullstellige Relation „`!`“, genannt „Cut“. Diese erlaubt es den Prolog-Baum an bestimmten Stellen zu kürzen und so die Suche einzuschränken. Bevor wir dies an einem Beispiel verdeutlichen, defi-

nieren wir zunächst formal die Wirkungsweise des Atoms. Hierzu sei ein Anfangsstück eines Baumes ein Teilbaum, der die Wurzel des Baumes enthält.

Definition 7.1 (Ursprung eines Cut-Atoms) *Sei $\mathcal{B}$ ein Zweig im Anfangsstück eines LD-Baumes und sei G ein Knoten in diesem Zweig, der das Cut-Atom an erster Stelle enthält. Dann verstehen wir unter dem* Ursprung *des Cut-Atoms den ersten Vorgänger von G in $\mathcal{B}$ (von $\leftarrow G$ aufwärts gezählt), der weniger Cut-Atome als G enthält.*

Damit legen wir fest:

Definition 7.2 *Wir erweitern die Notation des in Definition 4.4 eingeführten Prolog-Baumes für reines Prolog auf reines Prolog mit Cut. Hierzu fügen wir zu der Definition 4.4 des Operators* expand(T, G) *folgenden Fall hinzu:*

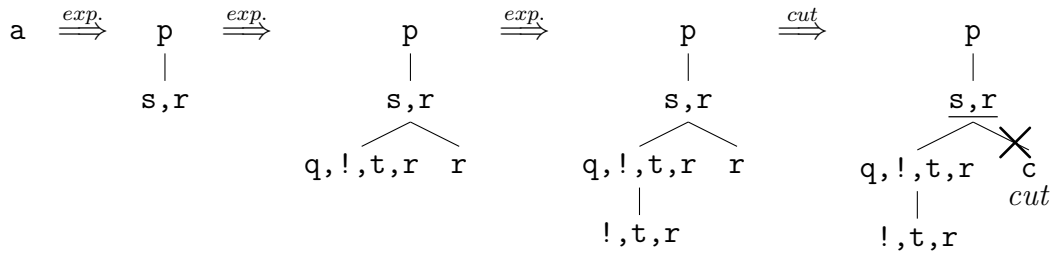
- ***prune**: das Atom an erster Stelle von G ist der Cut; sei G von der Form $\leftarrow !, A_1, \ldots, A_n$ und sei G' der Ursprung des Cut-Atoms. Lösche aus T alle Knoten, die Nachkommen von G' sind und rechts von dem Pfad liegen, der Q' mit Q verbindet, und Füge $\leftarrow A_1, \ldots, A_n$ als einzigen Tochterknoten von G hinzu.*

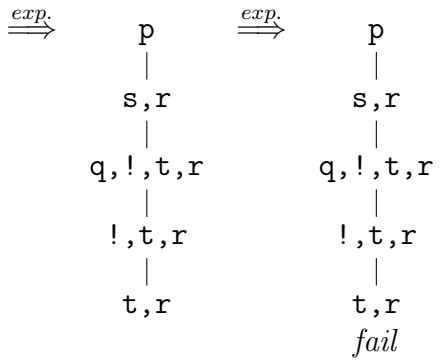
Der Prolog-Baum des definiten Zieles G sei der Grenzwert der wiederholten Anwendung von expand *auf das unmarkierte Ziel, das am weitesten links liegt.*

Dass ein solcher Grenzwert existiert, kann bei *Apt* [Apt97, S.286f] nachgelesen werden. Das Prolog-System arbeitet nun genau, wie in dieser Definition beschrieben. Wir betrachten folgendes Beispiel:

```
p   ←  s, r.
q.
s   ←  q, !, t.
s.
```

Der Prolog-Baum für das definite Ziel ← p wird schrittweise erzeugt:





Der fünfte Baum soll den Schritt **prune** verdeutlichen, er ist nicht durch den Operator expand erzeugt. Diese Tatsache kennzeichnen wir durch die entsprechende Beschriftung des Pfeils. Taucht im Berechnugsprozess ein Cut als zu verarbeitendes Atom auf, werden alle unmarkierten Zweige, die unterhalb vom Ursprung des Cut, aber nicht auf dem Pfad zu diesem liegen, abgeschnitten. In der Darstellung ist dies durch ein Kreuz gekennzeichnet, der Ursprung des Cut ist unterstrichen.
Die Definition macht deutlich, dass für die Verwendung des Cut die Reihenfolge der Klauseln von Bedeutung ist. Für das definite Ziel ← `p` endet die Berechnung mittels des Programmes

```
p  ←  q, !, r.
q
p.
```

erfolglos, mit

```
p.
p  ←  q, !, r.
q
```

dagegen erfolgreich.

Der Cut kann verschiedener Art sein. Werden nur Zweige abgeschnitten, die im Programm ohne das Cut-Atom mit *fail* markiert würden, ändert sich für die Korrektheit des Programmes nichts. Anders verhält es sich, wenn unendliche Zweige oder solche, die in einem Baum des entsprechenden Programmes mit *success* gekennzeichnet würden, wegfallen. Dies kann die Menge der mittels Prolog gefundenen Lösungen verändern. Im ersten Fall nennt man das Cut-Atom *Green Cut*, im zweiten *Red Cut*.
Weil auch das Naproche-System den beschriebenen Operator verwendet, ist für einen vollständigen Korrektheitsbeweis des Programmcodes die Überprüfung jedes benutzten Cuts notwendig. An den Stellen, an denen der Cut nicht bedeutungsverändernd eingesetzt wird, können weiterhin die vorgestellten Beweismethoden angewendet werden. Das System verwendet jedoch auch Red Cuts. Zum Beispiel sind einige der in Prolog eingebauten Mengenprädikate[5] oder auch die Negation[6] mit Hilfe solcher Cuts definiert.

[5]Informationen zu Definition und Wirkungsweise von Mengenprädikaten findet man in [Apt97, S.288ff].
[6]Mehr dazu in Abschnitt 7.5.

In diesen Fällen reicht der in dieser Arbeit beschriebene Ansatz für Korrektheitsbeweise nicht aus.

7.4 Meta-Variablen

Prolog erlaubt dem Benutzer, Variablen nicht nur als Terme zu verwenden, sondern auch als Atome. Eine so eingesetzte Variable nennt man *Meta-Variable*. Der Berechnungsprozess für entsprechende Ziele und Programme verläuft genauso wie der für reines Prolog mit dem Unterschied, dass die verwendeten mgus auch auf Meta-Variablen anwendbar sind. So liefert zum Beispiel das Programm

```
p(a).
a.
```

zusammen mit dem definiten Ziel ← `p(X), X.` das Ergebnis `X=a`. Voraussetzung für die Verwendung einer Meta-Variablen ist, dass ihr an der Stelle, an der sie im Programm aufgerufen wird, bereits ein Wert zugewiesen wurde. Für das obige Programm führt das definite Ziel ← `p(X), X, Y.` zu einem Fehler.
Meta-Variablen werden auch im Code des Naproche-Systems benutzt. Dies geschieht teilweise auf direktem oder fast direktem Wege, wie durch das Prädikat `call`:

```
call(X)  ←  X.
```

`call(X)` entspricht, wie man sich leicht klar macht, der Verwendung von `X` als Meta-Variable. Oft stehen im Code des Naproche-Systems aber auch eingebaute Prolog-Prädikate, die über Meta-Variablen definiert sind. So zum Beispiel `or`:

```
or(X,Y)  ←  X.
or(X,Y)  ←  Y.
```

Dieses Prädikat ermöglicht die Verwendung der Disjunktion im Körper von Klauseln. Prolog stellt dies mittels eines Operators dar:

```
:- op(1100, xfy, ;).
```

Dieser ist wie oben (mit `;` statt `or`) definiert.
Zusammen mit dem Cut kann durch Meta-Variablen die aus der imperativen Programmierung bekannte Relation `if_then_else` definiert werden:

```
if_then_else(P, Q, R)  ←  P, !, Q.
if_then_else(P, Q, R)  ←  R.
```

Ist `P` aus dem Programm ableitbar, so ist durch den Cut festgelegt, dass die Resolution nur dann erfolgreich ist, wenn auch `Q` gilt. Andernfalls wird eine Ableitung für `R` gesucht. In `P` darf hierbei das Cut-Atom nicht verwendet werden. Prolog repräsentiert die Relation mittels des Operators:

```
:- op( 1050, xfy, ->).
```

Mit der obigen Definition steht `P -> Q; R.` für `if_then_else(P, Q, R).` Schreibt man nur `P -> Q.`, so ist damit `if_then_else(P, Q, fail).` gemeint. `fail` ist hierbei ein eingebautes Prolog-Prädikat, für das jede Ableitung fehlschlägt. Auch dies findet sich im Programmcode des Naproche-Systems.
Weil der Cut-Operator im letzten Beispiel bedeutungsverändernd eingesetzt wird, kann für die Stellen, an denen das Prädikat `->` verwendet wird, der in dieser Arbeit vorgestellte deklarative Ansatz nicht verfolgt werden.
Anders verhält es sich für die Verwendung von Meta-Variablen in Verbindung mit reinem Prolog. *Apt* und *Ben-Eliyahu* zeigen in [ABE96], dass sich Meta-Variablen deklarativ interpretieren lassen. So kann man die Beweise für Korrektheit und Vollständigkeit der SLD-Resolution auf logische Programme mit Meta-Variablen übertragen. Dies kann dazu verwendet werden, Korrektheitsbeweise für Programme in reinem Prolog mit Meta-Variablen zu führen.

7.5 Negation

Prolog ermöglicht den Gebrauch negativer Literale im Körper von Klauseln und in Zielen. Dies ist folgendermaßen implementiert:

```
not(X)   ←  X, !, fail.
not(X).
```

Lässt sich `X` aus dem Programm ableiten, so wird `not(X)` zurückgewiesen. Umgekehrt gilt `not(X)`, wenn sich `X` nicht beweisen lässt. Alternativ zu der obigen Schreibweise mit dem Relationssymbol `not` ist in Prolog folgender Operator zur Negation von Atomen definiert:

```
:- op(900, fy, \+).
```

Dieser ist gleichbedeutend mit `not`. Wir benutzen in Anlehnung an die logische Negation im Folgenden das Zeichen „¬“. Viele Programme lassen sich durch Verwendung der Negation vereinfachen. Unter bestimmten Umständen kann ihr Gebrauch jedoch auch zu Komplikationen führen. Wir betrachten folgendes Programm:

```
p(X)     ←  ¬ q(X).
q(X)     ←  r(Y,X).
r(a,b).
```

Man macht sich leicht klar, dass `p(a)` aus diesem Programm ableitbar ist. Überraschend jedoch ist, dass Prolog für das definite Ziel ← `p(X)`. *fail* ausgibt. Dies liegt daran, dass das definite Ziel ← `q(X)`. erfolgreich mit der Belegung `X=b` abgeleitet werden kann. Das hier beschriebene Problem wird mit dem Begriff *floundering* bezeichnet.
Klauseln, deren Kopf ein Atom ist und deren Körper sowohl negative als auch positive Li-

terale enthalten kann, nennt man *allgemeine Klauseln*, Ziele, die aus beliebigen Literalen bestehen, heißen *allgemeine Ziele* und Programme aus allgemeinen Klauseln bezeichnet man als *allgemeine Programme*. Ist garantiert, dass alle negativen Literale, die in zur Ableitung verwendeten Zielen vorkommen, Grundliterale sind, so kann das beschriebene Problem ausgeschlossen werden. In diesem Fall lässt sich die Prolog-Negation auch deklarativ interpretieren. Hierfür wird die SLD-Resolution auf die sogenannte *SLDNF-Resolution* erweitert. Diese wurde von *K. L. Clark* [Cla78] eingeführt. Das Kürzel NF steht in diesem Begriff für *negation as finite failure*. Hierbei handelt es sich um eine Regel, die anschaulich folgendermaßen verstanden werden kann:
Für ein Grundatom `A` ist die Abeitung von ← ¬`A`. erfolgreich, wenn das Ziel ← `A` endlich fehlschlägt (finitely fails), d.h. dass der SLD-Baum dieses Ziels endlich ist und keine erfolgreiche Ableitung enthält. Umgekehrt schlägt die Ableitung von ← ¬`A`. endlich fehl, wenn das Ziel ← `A` erfolgreich abgeleitet werden kann.
Eine formale Definition der SLDNF-Resolution geben *Apt* und *Doets* in [AD94]. Für diese kann Vollständigkeit und Korrektheit nachgewiesen werden. Die deklarative Interpretation der Negation ermöglicht formale Programmverifikation für allgemeine Programme. Methoden hierzu findet man bei *Apt* [Apt95, S.83ff]. Weil Negation im Naproche-System selten vorkommt[7], werden diese Methoden im Rahmen dieses Textes nicht explizit behandelt.

7.6 Korrektheitsbeweise für das Naproche-System

Die letzten Abschnitte stellen die wichtigsten im Naproche-System verwendeten Prolog-Elemente und die Auswirkungen ihrer Verwendung auf die Korrektheit von Programmen vor. Erweitert man reines Prolog um einzelne dieser Elemente, so lassen sich für die meisten von ihnen mit Hilfe von Modifikationen Korrektheitsbeweise wie die vorher behandelten finden. Eine Ausnahme bildet das Cut-Atom. Dieses lässt sich kaum deklarativ interpretieren. Doch auch hier sind die gefundenen Verfahren hilfreich. Lassen sich Terminierung, Occur-Check-Freiheit und korrekte Funktion im Sinne der Spezifikation für ein Ziel und ein Programm in reinem Prolog nachweisen, so gelten die ersten beiden Eigenschaften auch für jede Version dieses Programmes, die sich nur durch Cut-Atome von der ursprünglichen unterscheidet, die letzte garantiert für diese zumindest, dass alle berechneten Antworten korrekt sind.
Weiterhin wurden eher technische Prädikate, die das Naproche-System verwendet, wie beispielsweise die Prädikate zum Einlesen des Eingabetextes, in diesem Kapitel weggelassen. Da diese jedoch den eigentlichen Berechnungsprozess der Programme des Systems kaum beeinflussen, können sie vernachlässigt werden.
Trotz der genannten Einschränkungen kann man für große Teile des Codes des Naproche-Systems auf die hier und vorher beschriebenen Weisen Korrektheitsbeweise finden. Viele Programme und Prädikate des Systems sind in reinem Prolog, stellenweise um Green

[7] Von den dieser Arbeit angehängten Programmen enthält nur fof_check.pl ein negatives Literal, vgl. Anhang A.2.2.

Cuts erweitert, geschrieben. So ist die Anwendung der gefundenen Beweisverfahren, obwohl sie gerade bezüglich der Korrektheit im Sinne einer Spezifikation sehr aufwendig sein können, prinzipiell möglich.

8 Zusammenfassung

In dieser Arbeit wurde aufgezeigt, wie sich die korrekte Funktionsweise von Prologprogrammen nachweisen lässt. Hierzu wurde im Einzelnen das Occur-Check-Problem, Terminierung und Korrektheit im Sinne der beabsichtigten Spezifikation behandelt. Für Ziele und Programme in reinem Prolog garantieren die Eigenschaften Nice-Modedness, Akzeptabilität und Well-Assertedness die fehlerfreie Prolog-Resolution. Am Beispiel der Formelgrammatik ist deutlich geworden, wie sich die gefundenen Methoden auf das Naproche-System anwenden lassen. Weiterhin wurde die Möglichkeit behandelt, Korrektheitsbeweise für das gesamte System zu führen. Dies ist für einen Großteil des Naproche-Systems möglich.
Damit leistet die Arbeit zum einen einen Beitrag, die Korrektheit von Naproche 0.2 nachzuweisen, zum anderen zeigt sie auf, welche Aspekte bei der Implementierung zukünftiger Versionen des Systems beachtet werden müssen. So kann man schon bei Implementierung eines Programmes darauf achten, dass die oben aufgezählten Eigenschaften erfüllt sind; je nach Komplexität kann es sogar hilfreich sein, eine konkrete Niveau-Abbildung, ein Moding und eine Spezifikation für jedes Prädikat im Programmcode zu vermerken.
Gerade weil es sich bei Naproche um ein Beweisprüfungssystem handelt und dieses, damit es Korrektheit nachweisen kann, korrekt arbeiten sollte, sind die in dieser Arbeit angesprochenen Themen auch für die Weiterentwicklung von Naproche von besonderer Bedeutung.

A Programm-Code

A.1 Formelgrammatik und mathematisches Lexikon

A.1.1 Originalcode von expr_grammar.pl

```
:-module(expr_grammar,[expr/5,term/4,variable/3,free_vars/3]).

%       This module parses a math-formula and gives out a DOBSOD-structure and
%       a list of free variables occurring in the formula.
%       A DOBSOD is a nested GULP list structure.
%       example:
%       x=x "\u2227" y=y
%       gives the DOBSOD:
%       DOBSOD = arity~2..args~
%                               [
%                                arity~2..args~
%                                             [
%                                              arity~0..type~variable..name~x,
%                                              arity~0..type~variable..name~x
%                                             ]..
%                                type~relation..name~ (=),
%                                arity~2..args~
%                                             [
%                                              arity~0..type~variable..name~y,
%                                              arity~0..type~variable..name~y
%                                             ]..
%                                type~relation..name~ (=)
%                               ]..
%                type~logical_symbol..name~ &,

%%      term(-DOBSOD:list,-List)
%       This predicate parses a term and gives out its DOBSOD-structure and a
%       list of free variables occurring in it.
%

%terms
term(X,Vars) -->
        (
                variable(X),{Vars=[X]}
                ;
                const(X,Vars)
                ;
                function(X,Vars)
        ),
        !.

%functions
function(X,Vars) -->
        {
                X=arity~1..args~[Y]
        },
        funcsym(X), !, lb, term(Y,Vars), rb,!
```

```
        ;
        {
                X=arity~2..args~[Y,Z]
        },
      funcsym(X), !, lb, term(Y,Vars1), comma, term(Z,Vars2), rb,
        {
                union(Vars1,Vars2,Vars)
        },
        !
        ;
        {
                X=arity~3..args~[Y,Z,YY]
        },
        funcsym(X), !, lb, term(Y,Vars1), comma, term(Z,Vars2), comma,
        term(YY,Vars3), rb,
        {
                union(Vars1,Vars2,Vari), union(Vari,Vars3,Vars)
        },
        !.

%%      expr(-DOBSOD:list, +Deepness:int,-Vars:list)
%
%       Calls the expression grammar and converts Input_expression into its
%       DOBSOD.
%       Finds the free variables occurring in the formula and gives it out as
%       a  list.
%
%       Example:
%
%       ==
%
%       ?- expr_grammar:expr(DOBSOD,10,Vars,"x=x",[]).
%
%       DOBSOD =        arity~2..args~
%                                       [
%                                               arity~0..type~variable..name~x,
%                                               arity~0..type~variable..name~x
%                                       ]..
%                       type~relation..name~ (=),
%       Vars = [[120]].
%
%       ?- expr_grammar:expr(X,10,Y,[120,61,120,8743,121,60,121],[]).
%
%       X = arity~2..args~[
%                           arity~2..args~[
%                                           arity~0..type~variable..name~x,
%                                           arity~0..type~variable..name~x
%                                         ]..
%                           type~relation..name~ (=),
%                           arity~2..args~[
%                                           arity~0..type~variable..name~y,
%                                           arity~0..type~variable..name~y
%                                         ]..type~relation..name~less
%                         ]..
%           type~logical_symbol..name~ &,
%       Y = [[121], [120]].
%
%       ==
%--------------expressions-------------------------
%brackets
expr(X,M,Vars) -->
        {
                succ(N,M)
        },
        lb,expr(X,N,Vars),rb.
```

```
%relations
expr(X,_,Var) -->
        {
                X = arity~0
        }, relationsymb(X),
        {
        Var = []
        }
        ;
        {
        X=args~[Y,Z]..arity~2
        },
        term(Y,Var1), relationsymb(X), term(Z,Var2),
        {
                union(Var1,Var2,Var)
        }
        ;
        {
                X=args~[Y]..arity~1
        },
        relationsymb(X), lb, term(Y,Var), rb
        ;
        {
                X=args~[Y,Z]..arity~2
        },
        relationsymb(X), lb, term(Y,Var1), comma, term(Z,Var2), rb,
        {
                union(Var1,Var2,Var)
        }
        .

%quantified expression
expr(X,M,Var) -->
        {
                succ(N,M),
                X=args~[Y,Z]
        },
        quantifier(X), variable_list(Y,Var1), expr(Z,N,Var2),
        {
                subtract(Var2,Var1,Var)
        }
      .

%logical expressions
expr(X,M,Var)    -->
        {
        succ(N,M)
        },
        (
                {
                        X=arity~1..args~[Y]
                },
                logsym(X), expr(Y,N,Var)
        ;
        {
                X=arity~2..args~[Z,Y]
                },
                expr(Z,N,Var1), logsym(X), expr(Y,N,Var2),
                {
                        union(Var2,Var1,Var)
                }
        ).

variable_list([X],[X])  -->
        variable(X).
```

```
variable_list([X|Rest],Var)  -->
        variable(X),
        variable_list(Rest,Var2),
        {
                union([X],Var2,Var)
        }.

%--------------terminal symbols-----------------------------------------

%brackets
lb -->
        "(".
rb -->
        ")".
%comma
comma -->
        ",".

% variables
variable(X) -->
        {
                X=type~variable,
                math_lexicon(Sign,X)

        },
        Sign.

% constants
const(X,[]) -->
        {
                X=type~constant,
                math_lexicon(Sign,X)
        },
        Sign.

%functionsymbols
funcsym(X) -->
        {
                X=type~function,
                math_lexicon(Sign,X)
        },
        Sign.

%relationsymbols
relationsymb(X) -->
        {
                X=type~relation,
                math_lexicon(Sign,X)
        },
        Sign.

%logical symbols
logsym(X) -->
        {
                X=type~logical_symbol,
                math_lexicon(Sign,X)
        },
        Sign.

%quantifiers
quantifier(X)-->
        {
                X=type~quantifier,
                math_lexicon(Sign,X)
        },
        Sign.
```

```
%--------------------------Free Var------------------------------

%% free_vars(+String:list(atom), -FreeVars:list(atom),DOBSOD_of_string:DOBSOD)
%
% True if FreeVars is the list of free (not quantified) variables within String,
% and DOBSOD_of_string is the DOBSOD form of the string.
% Note that spaces are removed from String before processing.
%
% NOTE: As of revision 16 the "squishing" is not needed here as it is handled
% by the XML preprocessor. Left in for compytibility purposes.
free_vars(String, FreeVars, DOBSOD_of_string) :-
        delete(String, 32, SquishedString),
        (
         phrase(expr(DOBSOD_of_string,10,TmpFreeVars), SquishedString),! ;
         phrase(term(DOBSOD_of_string,TmpFreeVars), SquishedString),!
        ),
        convert_to_math(TmpFreeVars , FreeVars).

convert_to_math([], []).
convert_to_math([X|Xs], [math(X)|Rest]) :-
        convert_to_math(Xs, Rest).
```

A.1.2 Originalcode von math_lexicon.pl

```
:- module(math_lexicon,[math_lexicon/2]).

/**     <module>  This module contains all mathematical items which math_grammar can parse
*
*
*/

%%      math_lexicon(?Item:list(int),DOBSOD:gulp_list)
%
%       math_lexicon lists all items and their properties which math_grammar can parse.
%
%       ==
%       math_lexicon("x",type~variable..arity~0..name~'x').
%       math_lexicon("f",type~function..arity~1..name~'f').
%       ==
%
%       @param Item     The Name of the item as a String.
%       @param DOBSOD   The DOBSOD representation of the Item.
%

% Variables
math_lexicon("t",type~variable..arity~0..name~'t').
math_lexicon("u",type~variable..arity~0..name~'u').
math_lexicon("v",type~variable..arity~0..name~'v').
math_lexicon("w",type~variable..arity~0..name~'w').
math_lexicon("x",type~variable..arity~0..name~'x').
math_lexicon("y",type~variable..arity~0..name~'y').
math_lexicon("z",type~variable..arity~0..name~'z').

% Constants
math_lexicon("\u2205",type~constant..arity~0..name~'emptyset').
math_lexicon("c",type~constant..arity~0..name~'c').

% Functions
math_lexicon("f",type~function..arity~1..name~'f').
math_lexicon("f2",type~function..arity~1..name~'f2').
math_lexicon("g",type~function..arity~2..name~'g').
math_lexicon("h",type~function..arity~3..name~'h').

% Relation
math_lexicon("contradiction",type~relation..arity~0..name~'$false').
math_lexicon("Ord",type~relation..arity~1..name~'ord').
math_lexicon("Trans",type~relation..arity~1..name~'trans').
math_lexicon("R",type~relation..arity~2..name~'r').
math_lexicon("=",type~relation..arity~2..name~'=').
math_lexicon("\u2260",type~relation..arity~2..name~'~=').
math_lexicon("\u2208",type~relation..arity~2..name~'in').
math_lexicon("\u2264",type~relation..arity~2..name~'leq').
math_lexicon("\u2265",type~relation..arity~2..name~'geq').
math_lexicon("\u003C",type~relation..arity~2..name~'less').
math_lexicon("\u003E",type~relation..arity~2..name~'greater').

% logical symbols
math_lexicon("\u00ac",type~logical_symbol..arity~1..name~'~').
math_lexicon("\u2227",type~logical_symbol..arity~2..name~'&').
math_lexicon("\u2228",type~logical_symbol..arity~2..name~'|').
math_lexicon("\u2192",type~logical_symbol..arity~2..name~'=>').
math_lexicon("\u2194",type~logical_symbol..arity~2..name~'<=>').

% quantifiers
math_lexicon("\u2200",type~quantifier..arity~2..name~'!').
math_lexicon("\u2203",type~quantifier..arity~2..name~'?').
```

A.1.3 Vereinfachte Version von expr_grammar.pl

```
% Programm: expression.pl
% Vereinfachte Version von expr_grammar.pl

%---------------terms-------------------------------------------
term(X) -->     variable(X).
term(X) -->     const(X).
term(X) -->     function(X).

%---------------functions---------------------------------------
function(X) -->
        {X=arity~1..args~[Y]},
        funcsym(X), lb, term(Y), rb.

function(X) -->
      {X=arity~2..args~[Y,Z]},
      funcsym(X), lb, term(Y), comma, term(Z), rb.

function(X) -->
      {X=arity~3..args~[Y,Z,YY]},
      funcsym(X), lb, term(Y), comma, term(Z), comma, term(YY), rb.

%---------------expressions-------------------------------------

%unnecessary brackets
expr(X) -->
        lb,expr(X),rb.

%relations
expr(X) -->
          {X=arity~0},
         relationsymb(X).

expr(X)-->
        {X=args~[Y,Z]..arity~2},
        term(Y), relationsymb(X), term(Z).

expr(X)-->
        {X=args~[Y]..arity~1},
        relationsymb(X), lb, term(Y), rb.

expr(X)-->
        {X=args~[Y,Z]..arity~2},
        relationsymb(X), lb, term(Y), comma, term(Z), rb.

%quantified expression
expr(X) -->
        {X=args~[Y,Z]},
        quantifier(X), variable_list(Y), expr(Z).

%logical expressions
expr(X)    -->
        {X=arity~1..args~[Y]},
        lb,logsym(X), expr(Y),rb.

 expr(X) -->
        {X=arity~2..args~[Z,Y]},
        lb,expr(Z), logsym(X),expr(Y),rb .

%---------------lists--of--variables------------------------------
variable_list([X])  -->
        variable(X).

variable_list([X|Rest])  -->
        variable(X),
```

```
        variable_list(Rest).

%---------------terminal--symbols---------------------------------
%brackets
lb -->
        "(".
rb -->
        ")".

%comma
comma -->
        ",".

% variables
variable(type~variable..arity~0..name~'x') --> "x".
variable(type~variable..arity~0..name~'y') --> "y".
variable(type~variable..arity~0..name~'z') --> "z".

% constants
const(type~constant..arity~0..name~'emptyset') -->   "\u2205".
const(type~constant..arity~0..name~'c') -->   "c".

%functionsymbols
funcsym(type~function..arity~1..name~'f') --> "f".
funcsym(type~function..arity~2..name~'g') --> "g".
funcsym(type~function..arity~3..name~'h') --> "h".

%relationsymbols
relationsymb(type~relation..arity~0..name~'$false') -->  "contradiction".
relationsymb(type~relation..arity~1..name~'ord') -->  "Ord".
relationsymb(type~relation..arity~1..name~'trans') -->  "Trans".
relationsymb(type~relation..arity~2..name~'r') -->  "R".
relationsymb(type~relation..arity~2..name~'=') -->  "=".
relationsymb(type~relation..arity~2..name~'in')-->"\u2208".
relationsymb(type~relation..arity~2..name~'less')-->"\u003C".
relationsymb(type~relation..arity~2..name~'greater')-->"\u003E".

% logical symbols
logsym(type~logical_symbol..arity~1..name~'~')-->"\u00ac".
logsym(type~logical_symbol..arity~2..name~'&')-->"\u2227".
logsym(type~logical_symbol..arity~2..name~'|')-->"\u2228".
logsym(type~logical_symbol..arity~2..name~'=>')-->"\u2192".
logsym(type~logical_symbol..arity~2..name~'<=>')-->"\u2194".

% quantifiers
quantifier(type~quantifier..arity~2..name~'!')-->"\u2200".
quantifier(type~quantifier..arity~2..name~'?')-->"\u2203".
```

A.2 Das Logik-Modul

Zur Veranschaulichung von Kapitel 7 ist hier ein repräsentativer Teil des Quellcodes des Logik-Moduls angehängt. Die Beispiele wurden so gewählt, dass sie die meisten dort erwähnten Prolog-Elemente enthalten.

A.2.1 checker.pl

```
:-module(check_prs,[check_prs/6]).

:- use_module(library(pldoc)).
:- ensure_loaded(naproche(gulp4swi)).
:- use_module('premises').
:- use_module('fof_check').
:- use_module(naproche(prs)).

/**     <module> High level proof checking predicate
 *
 *      This module provides predicates to check a PRS using a first order prover.
 *
 */



%%      check_prs(+PRS:prs,+Mid_begin:list,-Mid_end:list,+Premises_begin:list(DOBSOD),
%%      -Premises_end:list(DOBSOD), +Check_trigger:(check | nocheck)) is det.
%
%       True if PRS is logically valid.
%
%       Depending on the Check_trigger, the PRS is translated into TPTP FOL and checked,
%       using the prover Prover if Check_trigger = check, or just translated into TPTP FOL
%       if Check_trigger = nocheck.
%       A PRS is logically valid, if all its conditions are logically valid.
%
%       NOTE: In order to actually check the input you need to define the following predicates:
%       check_time(Time)
%       check_prover(Checker)
%       check_size(Outputsize)
%       These are normally defines in load.pl. fof_check will not work without them!
%
%       @param PRS is the PRS to be checked.
%       @param Mid_begin is the list of available Math_IDs before the start of the predicate.
%       @param Mid_end is the list of available Math_IDs at the end of the predicate
%       @param Premises_begin is the list of premises from which we try to prove the PRS.
%       @param Premises_end is the list containing Premises_begin and every formula we proved through
%        the PRS.
%       @param Check trigger gives us the option to either try to prove every formula we encounter
%        (check) or to just go through the structure of the proof without running a prover (nocheck).
%
%
%       @tbd Mid handling in Negation case. Should they be negated?

check_prs(PRS,Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        PRS = id~Id..conds~Conds,
        !,

        % discourse_to_prs gives us the conditions in the wrong order. Therefore we have to
        % reverse them before we can check them.
        reverse(Conds,Reversed_Conds),

        % Check whether PRS is a structure PRS, in our case a lemma or a theorem
        % e.g. check if Id = lemma.. or Id = theorem..
```

```
        ( ( atom_concat(theorem,_,Id) ; atom_concat(lemma,_,Id) ) ->
                (
                %theorem / lemma case
                Reversed_Conds = [Goal,Proof],

                % Get the Math IDs and the Premises of the Goal, but don't check
                check_prs(Goal,Mid_begin,Goal_Mid_end,Premises_begin,Goal_Premises_end,nocheck),

                % Check the Proof with updated Math IDs, discard Math IDs at the end
                % Assumptions for Theorem must be made again!
                check_prs(Proof,Goal_Mid_end,_,Premises_begin,Proof_Premises_end,Check_trigger),

                % If Check_trigger = check see if Proof is a proof for Goal
                ( Check_trigger = check ->
                        ( append(Premises_begin,Thm,Goal_Premises_end),
                          fof_check(Proof_Premises_end,Thm,Id)
                        );
                        true
                ),
                % Set New Math_Ids & Premises
                Goal_Mid_end = Mid_end,
                Premises_end = Goal_Premises_end
                )
                ;
                % If PRS is not a theorem just check the conditions
                check_conditions(Id,Reversed_Conds,Mid_begin,Mid_end,Premises_begin,Premises_end,
                                                                                Check_trigger)
                ).

% ---------------------------------- check_neg_prs ------------------------------------------------

%%      check_neg_prs(+NEG_PRS:neg(prs),+Mid_begin:list,-Mid_end:list,+Premises_begin:list(DOBSOD),
%%       -Premises_end:list(DOBSOD), +Check_trigger:(check | nocheck)) is det.
%
%       True if NEG_PRS is logically valid, takes a negated PRS as input.
%
%       Depending on the Check_trigger, the NEG_PRS is translated into TPTP FOL and checked,
%       using the prover Prover if Check_trigger = check, or just translated into TPTP FOL
%       if Check_trigger = nocheck.
%       A NEG_PRS is logically valid, if all its conditions are logically valid.
%
%       NOTE: In order to actually check the input you need to define the following predicates:
%       check_time(Time)
%       check_prover(Checker)
%       check_size(Outputsize)
%       These are normally defines in load.pl. fof_check will not work without them!
%
%       @param NEG_PRS is the negated PRS to be checked.
%       @param Mid_begin is the list of available Math_IDs before the start of the predicate.
%       @param Mid_end is the list of available Math_IDs at the end of the predicate
%       @param Premises_begin is the list of premises from which we try to prove the PRS.
%       @param Premises_end is the list containing Premises_begin and every formula we proved through
%        the PRS.
%       @param Check trigger gives us the option to either try to prove every formula we encounter
%        (check) or to just go through the structure of the proof without running a prover (nocheck).


check_neg_prs(neg(PRS),Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        !,
        % Check the PRS
        check_prs(PRS,Mid_begin,Mid_end,Premises_begin,PRS_Premises_end,Check_trigger),

        % Get the premises which are added by the PRS
        append(Premises_begin,PRS_Premises,PRS_Premises_end),

        % Negate the PRS_Premises
```

```
        negate_formulas(PRS_Premises,Negated_Premises),

        % Append the negated Premises
        append(Premises_begin,Negated_Premises,Premises_end).



% ---------------------------------- check_conditions -------------------------------------------

%%      check_conditions(+Id:atom,+Conditions:list,+Mid_begin:list,-Mid_end:list,
%%      +Premises_begin:list(DOBSOD),-Premises_end:list(DOBSOD),
%%      +Check_trigger:(check|nocheck)) is det.
%
%       True if every Element of the List is logically valid, or if the list is empty
%
%       check_conditions checks the Elements of the List one ofter the other.
%
%       Possible Conditions X are:
%
%       * X is a PRS
%       * X = math_id(_,_)
%       * X = holds(Y)
%       * X = PRS_A := PRS_B    Definition
%       * X = PRS_A => PRS_B    Assumption
%       * X = neg(PRS)          Negation
%       * X = PRS_A ==> PRS_B   For all
%       * X = PRS_A ==> PRS_B   Implication
%       * X = contradiction     Contradiction
%
%       NOTE: In order to actually check the input you need to define the following predicates:
%       check_time(Time)
%       check_prover(Checker)
%       check_size(Outputsize)
%       These are normally defines in load.pl. fof_check will not work without them!
%
%       @param Id is the ID of the PRS we are checking
%       @param Conditions is the list of conditions which we try to prove.
%       @param Mid_begin is the list of available Math_IDs before the start of the predicate.
%       @param Mid_end is the list of available Math_IDs at the end of the predicate
%       @param Premises_begin is the list of premises from which we try to prove the PRS.
%       @param Premises_end is the list containing Premises_begin and every formula we proved through
%        the PRS.
%       @param Check trigger gives us the option to either try to prove every formula we encounter
%        (check) or to just go through the structure of the proof without running a prover (nocheck).


% ------------------------ Empty List ---------------------------------------
% Empty list is valid.
% Mid_begin = Mid_end
% Premises_begin = Premises_end
check_conditions(_,[],Mid_begin,Mid_begin,Premises_begin,Premises_begin,_) :- !.


% ------------------------ PRS ----------------------------------------
% X is a PRS
% Check PRS then proceed with the updated Math_IDs and Premises.
check_conditions(Id,[X|Rest],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        is_prs(X),
        !,
        check_prs(X,Mid_begin,X_Mid_end,Premises_begin,X_Premises_end,Check_trigger),

        % Use the updated Mid and Premises for the remaining check.
        check_conditions(Id,Rest,X_Mid_end,Mid_end,X_Premises_end,Premises_end,Check_trigger).


% ------------------------ math_id(_,_) ---------------------------------------
```

```
% X = math_id(_,_)
% Update Mid_begin and proceed
check_conditions(Id,[X|Rest],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        X = math_id(_,_),
        !,
        append(Mid_begin,[X],New_Mid_begin),
        check_conditions(Id,Rest,New_Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger).

% ------------------------ holds(_) --------------------------------------
% X = holds(Y)
% We use the Prover to check whether Y follows from the premises.
% If yes, then the TPTP representation of Y is added to the premises, else
% we throw an error.
% Check_trigger specifies whether we do the actual check or just update the premises.
check_conditions(Id,[ holds(Y) |Rest],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        !,
        % Find the Formula corresponding to Y.
        % If it can't be found in Mid_begin throw an error
        Z = math_id(Y,Formula_PRS),
        member(Z,Mid_begin),
        % If FAIL THROW ERROR !!!

        % Get rid of the math(..) part.
        Formula_PRS = math(Formula_FOL),
        (((Formula_FOL = type~quantifier),!);
         ((Formula_FOL = type~relation),!);
         ((Formula_FOL = type~logical_symbol),!)),!,

        % If Check_trigger = check use Prover to check whether Formula follows from the premises
        % If it can't be proven throw an error.
        % If Check_trigger = nocheck do nothing.
        ( Check_trigger = nocheck -> true
                ; fof_check(Premises_begin,[Formula_FOL],Id)
        ),
        % If FAIL, THROW ERROR!!!

        % Update premises
        append(Premises_begin,[Formula_FOL],New_Premises_begin),

        !,
        check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------ Definition --------------------------------------
% X = A := B
% A and B both can be either a PRS or a negated PRS.
% A Definition is just one more premise. Therefore we update the premises and proceed.
check_conditions(Id,[ X |Rest],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        X = (A:=B),
        !,

        % Extract the list of Premises from A and B
        ( is_prs(A) ->
         check_prs(A,Mid_begin,A_Mid_end,Premises_begin,A_Premises_end,nocheck);
         check_neg_prs(A,Mid_begin,A_Mid_end,Premises_begin,A_Premises_end,nocheck)
        ),
        append(Premises_begin,List_A,A_Premises_end),!,

        ( is_prs(B) ->
         check_prs(B,A_Mid_end,_,A_Premises_end,B_Premises_end,nocheck);
         check_neg_prs(B,A_Mid_end,_,A_Premises_end,B_Premises_end,nocheck)
        ),
        append(A_Premises_end,List_B,B_Premises_end),!,

        % Update the premises
        update_definitions(Premises_begin,New_Premises_begin,List_A,List_B),
```

```
        !,
        check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------ Assumption ---------------------------------------
% X = A => B
% A and B both can be either a PRS or a negated PRS.
% This results in a premises update:
% For all premises X from B we add
% ! [Free variables in A] : Fol_A -> X
% to the list of premises available
check_conditions(Id,[ X |Rest],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        X = (A=>B),
        !,

        % Check A. The values of Mid_tmp and Premises_tmp will be given to
        % B as _begin values.
        % We use nocheck as these are Assumptions and need not be checked.
        ( is_prs(A) ->
         check_prs(A,Mid_begin,Mid_tmp,Premises_begin,A_Premises_end,nocheck);
         check_neg_prs(A,Mid_begin,Mid_tmp,Premises_begin,A_Premises_end,nocheck)
        ),!,

        % Extract the list of premises from A
        append(Premises_begin,List_A,A_Premises_end),


        % Check B with all the Premises and Mid from A included
        % The Mid_begin values are Mid_tmp and the Premises_begin values are Premises_tmpa,
        % both of which we got from check_prs(A,..)
        % The Mid_end value is not important ( ?? )
        % The Premises_end value will later be used for all-quantification.
        %
        % If Check_trigger = nocheck then don't check B, else do check it
        ( is_prs(B) ->
         ( Check_trigger = nocheck ->
                check_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,nocheck);
                check_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,check)
         );
         ( Check_trigger = nocheck ->
                check_neg_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,nocheck);
                check_neg_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,check)
         )
        ),
        !,

        % Two cases:
        % A : We deal with a normal assumption
        % B : The user made a prove by contradiction
        % The last entry in B_Premises_end determines that
        ( append(_,[type~relation..arity~0..name~'$false'],B_Premises_end) ->
                (
                % Contradiction case
                % Negate Assumption and proceed
                negate_formulas(List_A,Negated_A),
                append(Premises_begin,Negated_A,New_Premises_begin)
                )
                ;
                (
                % For all Formulas X in Premises_tmpb / Premises_tmp_a add
                % a new over all free variables in A quantified Statement of the form
                % ! [Free variables in A] : Fol_A -> X
                % to our Premises storage
                append(A_Premises_end,Premises_B,B_Premises_end),
                update_assumption(Premises_begin,New_Premises_begin,List_A,Premises_B)
                )
```

```
        ),

        % Reset the Mid values to before the Assumption
        % The Premises_begin becomes the New_Premises_begin which we got
        % from the last predicate.
        !,
        check_conditions(Id,Rest,Mid_tmp,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- Negation ---------------------------------------
% X = neg(PRS)
% Takes all premises from PRS and negates them.
% Check occurs after the negation.
check_conditions(Id,[neg(PRS) | Rest ],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        !,
        % Get the Premises and Mids of PRS
        % Check_trigger is nocheck as we want to prove the negates formulas!
        check_prs(PRS,Mid_begin,New_Mid_begin,Premises_begin,Tmp_Premises_end,nocheck),

        % Get the new premises and negate them
        append(Premises_begin,Premises_PRS,Tmp_Premises_end),
        negate_formulas(Premises_PRS,Negated_Premises_PRS),

        % Check the Premises if Check_trigger = check
        % otherwise just append them
        ( Check_trigger = check ->
                fof_check(Premises_begin,Negated_Premises_PRS,Id);
                true
        ),
        append(Premises_begin,Negated_Premises_PRS,New_Premises_begin),

        % Check the rest of the conditions with New_Mid_begin and New_Premises_begin
        check_conditions(Id,Rest,New_Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).

% ------------------------- Implication & for all/exists ---------------------------------
% X = A ==> B
% A and B both can be either a PRS or a negated PRS.
% Find the Formulas in PRS_A and adds them as Premises. Then proceeds to check PRS_B with the updated
% premises
check_conditions(Id,[ X | Rest ],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        X = (A==>B),
        !,

        % Get the Premises and Mids from A
        % We don't check as these are Assumptions!
        ( is_prs(A) ->
                check_prs(A,Mid_begin,A_Mid_begin,Premises_begin,A_Premises_end,nocheck);
                check_neg_prs(A,Mid_begin,A_Mid_begin,Premises_begin,A_Premises_end,nocheck)
        ),!,

        % Check B with the updated Premises
        % The Math IDs of A and B will not matter for the Rest of the proof.
        ( is_prs(B) ->
                check_prs(B,A_Mid_begin,_,A_Premises_end,B_Premises_end,Check_trigger);
                check_neg_prs(B,A_Mid_begin,_,A_Premises_end,B_Premises_end,Check_trigger)
        ),!,

        % Get the Formulas in A and B
        append(Premises_begin,List_A,A_Premises_end),
        append(A_Premises_end,List_B,B_Premises_end),

        % Update the original premises:
        % For each formula F in B add
        % ! [Free A] : A => F
        % to the premises.
        ( List_A = [] ->
```

```
                (
                % A comes from a for all statement in natural language
                % We quantify over each Mref of A
                A = mrefs~Mrefs_A..id~Id_A,
                 (atom_concat('prefix_forall',_,Id_A) ->
                  update_for_all(Premises_begin,New_Premises_begin,Mrefs_A,List_B);
                  update_there_exists(Premises_begin,New_Premises_begin,Mrefs_A,List_B)
                 )
                ) ;
                % A comes from a normal Implication
                update_implication(Premises_begin,New_Premises_begin,List_A,List_B)
        ),

        % Check the rest of the conditions with Mid_begin and New_Premises_begin
        check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- contradiction ----------------------------------------
check_conditions(Id,[contradiction|Rest],Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
        !,

        % Check whether we can conclude $false from the premises so far.
        ( Check_trigger = check ->
                fof_check(Premises_begin,[type~relation..arity~0..name~'$false'],Id);
                true
        ),

        % Add [contradiction] to the premises
        append(Premises_begin,[type~relation..arity~0..name~'$false'],New_Premises_begin),

        % Check the rest of the conditions with New_Premises_begin
        check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- Everything else --------------------------------------
% Every other case
%check_conditions([_|Rest],Mid_begin,Mid_end,Premises_begin,Premises_end) :-
%       !,
%       % throw error: 'Unknown condition'
%       check_conditions(Rest,Mid_begin,Mid_end,Premises_begin,Premises_end). % or fail?
```

A.2.2 fof_check.pl

```
:- module(fof_check,[fof_check/3,check_theorem/0]).

:- use_module(library(pldoc)).

:- use_module(translation_tptp,[prepare_tptp/4]).

%%      fof_check(+Premises:list(DOBSOD),+Formulae:list(DOBSOD),+PRS_ID:atom)
%
%       This is the main predicate for checking whether a conjecture is provable from a given set of
%       premises using standard TPTP Checkers, like Otter 3.3, etc. The predicate takes in a list of
%       premises in the form of DOBSODs as well as a list of formulae, and the id of the PRS in question.
%       With these data, the premises and the formulae are passed to the specified TPTP Checker as
%       'axioms' and 'conjectures' and the result is written in three formats: a short version in the
%       file 'final_output' in the user's working directory, the exact input to the prover in a file
%       named <inputPRS_OD> and  a detailed output version in the file with name of type <outputPRS_ID>
%       in the folder Output, also present in the working directory. Specifications for SystemsOnTPTP are
%       made in load and proved here.
%
%
%       Note: Make sure that you set $NAPROCHE_HOME_PATH as the path to your Naproche directory before
%       you run this program!
%
%       Example:
%
%       Human  : The equality relation is transitive. That is, given x=y and y=z, it implies x=z.
%
%       ==
%
%       fof_check([type~relation ..name~'=' ..arity~2 ..args~[type~variable ..name~x ..arity~0,
%                                                             type~variable ..name~y ..arity~0],
%                                                               %Input premises in DOBSOD
%                  type~relation ..name~'=' ..arity~2 ..args~[type~variable ..name~y ..arity~0,
%                                                            type~variable ..name~z ..arity~0]],
%                 [type~relation ..name~'=' ..arity~2 ..args~[type~variable ..name~x ..arity~0,
%                                                             type~variable ..name~z ..arity~0]],
%                                                               %Input formulae in DOBSOD
%                 0                                                                          %PRS_ID
%                 ).
%
%       ==

fof_check(Premises,Formulae,PRS_ID) :-
        % Get SystemsOnTPTP Specifications
        check_time(Time),
        check_prover(Checker),
        check_size(Outputsize),
        % Prepare the premises and formulae,
        % e.g. append fof(name,axiom... ) to each
        prepare_premises(Premises,Prepared_premises,0),
        prepare_formulae(Formulae,Prepared_formulae,0),

        % Using the prolog predicate append the two lists into a concatenated list of Prepared_predicates
        append(Prepared_premises,Prepared_formulae,Prepared_predicates),

        % Write Prepared_predicates to <inputPRS_ID>, run the checker, and read the result from the file
        % <outputPRS_ID>
        getenv('NAPROCHE_HOME_PATH',Naproche_home),
        atom_concat(Naproche_home,'/output/input',Naproche_input)
        atom_concat(Naproche_input,PRS_ID,Input_File),
        write_to_file(Input_File,Prepared_predicates),
        atom_concat('$NAPROCHE_HOME_PATH/output/output',PRS_ID,File),

        % Run the checker
        write('Running Prover \n'),
```

```
        write('perl $TPTP_HOME/SystemExecution/SystemOnTPTP \n'),
        concat_atom(['perl $TPTP_HOME/SystemExecution/SystemOnTPTP -q',Outputsize,' ',Checker,' ',Time,'
                                            ',Input_File,'>',File],Command),
        shell(Command),

        % File Output
        % atom_concat(Naproche_home,'/output/final_output',final_output),
        open(final_output,append,OS),
        nl(OS),
        write(OS,PRS_ID),
        tab(OS,5),
        write(OS,Prepared_formulae),
        tab(OS,5),

        % Bash Output
        write(PRS_ID),
        write('\t'),
        write(Prepared_formulae),
        write('\t'),

        % check whether the check was successful
        ( Outputsize=3 ->
                ( read_from_file(File,"Theorem") ->
                        (write(OS,'Theorem'),
                        write('Theorem \n \n'))
                        ;
                        (write(OS,'No Proof'),
                        write('No Proof \n'))
                )
                ;
                (
                (Outputsize=0;Outputsize=1;Outputsize=2),!,
                name(Checker,Prover),
                append(Prover,"saysTheorem",Search_string),
                ( read_from_file(File,Search_string) ->
                        (write(OS,'Theorem'),
                        write('Theorem'))
                        ;
                        (write(OS,'No Proof'),
                        write('No Proof'))
                )
                )
        ),
        close(OS).

%%      check_theorem
%
%       This predicate checks whether the final_output has any string of the form "No Proof". If it
%       doesn't, this implies the entire string of arguments have been proved.


check_theorem :- catch(\+(read_from_file('$NAPROCHE_HOME_PATH/final_output',"NoProof")),_,
                 write('Nothing to check')).

%%      prepare_premises(+Premises:list(DOBSOD),-Prepared_premises:list(atom),+I:int)
%
%       This predicate is used to convert a list of premises which are in the form of a list of DOBSODs
%       into a list of axioms in the TPTP format. This is the set of premises which would be further
%       passed on to the checker to check the validity of the conjecture. The input is the list of
%       DOBSODs and the counter I, which is used to uniquely name the axioms in the TPTP Format. These
%       entities are further passed on to the predicate prepare_tptp which translates them into atoms.
%       The output consists of the list of Prepared Premises in TPTP format.
%
%       Example:
%
%       ==
```

```
%
%       prepare_premises([type~logical_symbol
%                         ..name~ '~'
%                         ..arity~1
%                         ..args~[type~variable ..name~x ..arity~0],type~variable ..name~y ..arity~0,
%                                 type~variable ..name~z ..arity~0],
%                         X,
%                         0
%                        ).                                                          %Input list
%
%       X= [fof(1,axiom,'~(vx)'),fof(2,axiom,'vy'),fof(3,axiom,'vz')]                    %Output
%
%       ==


prepare_premises(Premises,Prepared_premises,I) :-
        Premises=[H1|T1],
        !,
        prepare_tptp(H1,H2,[],_),
        Newi is I+1,
        Prepared_premises=[Head|Tail],
        Head= fof(Newi,axiom,H2),
        prepare_premises(T1,Tail,Newi).
prepare_premises([],[],_) :- !.

%%      prepare_formulae(+Formulae:list(gulp),-Prepared_formulae:list(atom),+J:int)
%
%       This predicate is used to convert a list of formulae which are in the form of a list of DOBSODs
%       into a list of conjecutures in the TPTP format. These conjectures are then passed on to the
%       checker to check their validity. The input is the list of DOBSODs, and the counter J which is
%       used to uniquely name the conjectures inthe TPTP Format. These entities are further passed on to
%       the predicate prepare_tptp which translates them into atoms. The output consists of the list of
%       Prepared Formulae in TPTP format.
%
%       Example:
%
%       ==
%
%       prepare_formulae([type~logical_symbol
%                         ..name~ '~'
%                         ..arity~1
%                         ..args~[type~variable ..name~x ..arity~0],
%                                 type~variable ..name~y ..arity~0,
%                                 type~variable ..name~z ..arity~0],
%                         X,
%                         0
%                        ).
%       X= [fof(1,conjecture,'~(vx)'),fof(2,conjecture,'vy'),fof(3,conjecture,'vz')].
%
%       ==


prepare_formulae(Formulae,Prepared_formulae,J) :-
        Formulae=[H2|T2],
        !,
        prepare_tptp(H2,H3,[],_),
        Newj is J+1,
        Prepared_formulae=[Head|Tail],
        Head= fof(Newj,conjecture,H3),
        prepare_formulae(T2,Tail,Newj).
prepare_formulae([],[],_) :- !.


%%      write_to_file(+File:atom,+Prepared_predicates:list(atom))
%
%       Once the axioms and conjectures have been prepared in the TPTP Format, we append them into a
```

```
%       single list called Prepare_predicates and pass it onto this predicate which writes it into the
%       file File.

write_to_file(File,Prepared_predicates) :-
        Prepared_predicates=[Head|Tail],
        !,
        open(File,append,OS),
        write(OS,Head),
        write(OS,.),
        nl(OS),
        close(OS),
        write_to_file(File,Tail).
write_to_file(_,[]) :- !.

%%      read_from_file(+File:atom,+Word:list(int))
%
%       Once the TPTP Checker checks the validity (or the lack of it) of the conjecture, the result is
%       written in the file "output". This predicate is designed to read a "word" or a list of integers
%       from a file "File". The predicate uses the predicates check_word/3 and sublist/2. The former
%       returns the contents of the file in as a list of ASCII Codes and the latter checks the presence
%       of the given word in the file by checking if the list of integers is a sublist of the list of
%       ASCII Codes. For our purposes, File is the file "output" and the word to be read is "Theorem"
%       for checking the validity of the fof.


read_from_file(File,Word) :-
        atom_concat('$NAPROCHE/',File_in,File),
        open(File_in,read,OS),
        get0(OS,Char),
        check_word(Char,Charlist,OS),
        sublist(Word,Charlist),
        close(OS).


%%      check_word(+Char:int,-Charlist:list(int),+OS)
%
%       This predicate reads in the list of characters from the file stream OS.
%       Here
%       10 = "Return"
%       32 = "Space Bar"
%       -1 = "End of Stream"
%       In each of these cases, the next list is the empty list, indicating the end of file.

check_word(-1,[],_) :- !.
check_word(end_of_file,[],_) :- !.
check_word(Char,[Char|Chars],OS) :-
        !,
        get(OS,NextChar),
        check_word(NextChar,Chars,OS).

%%      sublist(+X:list, +Y:list)
%
%       The predicate checks if the list is a sublist of the list Y. It uses the predicates "prefix" and
%       "suffix" and the prolog predicate "append".

prefix(X,Y) :- append(X,_,Y).
suffix(X,Y) :- append(_,X,Y).
sublist(X,Y):- suffix(S,Y),prefix(X,S).
```

Literaturverzeichnis

[ABE96] Apt, K. R. und R. Ben-Eliyahu: *Meta-variables in logic programming, or in praise of ambivalent syntax.* Fundam. Inf., 28(1-2):23–36, 1996.

[AD94] Apt, K. R. und K. Doets: *A New Definition of SLDNF-Resolution.* The Journal of Logic Programming, 18:177–190, 1994.

[AP93] Apt, K. R. und D. Pedreschi: *Reasoning about termination of pure Prolog programs.* Information and Computation, 106(1):109–157, 1993.

[AP94] Apt, K. R. und A. Pellegrini: *On the occur-check-free Prolog programs.* ACM Trans. Program. Lang. Syst., 16(3):687–726, 1994.

[Apt95] Apt, K. R.: *Program verification and Prolog.* In: *Specification and validation methods*, Seiten 55–95. Oxford University Press, Inc., New York, NY, USA, 1995.

[Apt97] Apt, K. R.: *From Logic Programming to Prolog.* Prentince-Hall, London, 1997.

[BBS06] Blackburn, P., J. Bos und K. Striegnitz: *Learn Prolog Now!*, Band 7 der Reihe *Texts in Computing.* College Publications, 2006.

[Bez93] Bezem, M. A.: *Strong Termination of Logic Programs.* Journal of Logic Programming, 15(1&2):79–98, 1993.

[Bli91] Blizard, W. D.: *The development of multiset theory.* Modern Logic, 1(4):319–352, 1991.

[Cla78] Clark, K. L.: *Negation as Failure.* In: Gallaire, H. und J. Minker (Herausgeber): *Logic and Databases*, Seiten 293–322. Plenum Press, New York, USA, 1978.

[Cov94] Covington, M. A.: *GULP 3.1: An Extension of Prolog for Unification-Based Grammar.* Technischer Bericht AI-1994-06, Artificial Intelligence Center, The University of Georgia, 1994.

[CR96] Colmerauer, A. und P. Roussel: *The birth of Prolog.* History of programming languages—II, Seiten 331–367, 1996.

[Cra09] Cramer, M.: *Mathematisch-logische Aspekte von Beweisrepräsentationsstrukturen.* Masterarbeit, Rheinische Friedrich-Wilhelms-Universität Bonn, 2009.

[Ded61] Dedekind, R.: *Was sind und was sollen die Zahlen?* Vieweg, Braunschweig, 9., unveränderte Auflage, 1961.

[DM79] Dershowitz, N. und Z. Manna: *Proving termination with multiset orderings*. Commun. ACM, 22(8):465–476, 1979.

[DM85] Deransart, P. und J. Maluszynski: *Relating Logic Programs and Attribute Grammars*. Journal of Logic Programming, 2(2):119–155, 1985.

[DM05] Drabent, W. und M. Milkowska: *Proving correctness and completeness of normal programs – a declarative approach*. Theory Pract. Log. Program., 5(6):669–711, 2005.

[Doe94] Doets, K.: *From Logic to Logic Programming*. The MIT Press, Cambridge, Massachusetts, London, England, 1994.

[EFT07] Ebbinghaus, H.-D., J. Flum und W. Thomas: *Einführung in die mathematische Logik*. Spektrum Akademischer Verlag, Berlin Heidelberg, 5. Auflage, 2007.

[Her67] Herbrand, J.: *Investigations in Proof Theory*. In: Heijenpoort, J. van (Herausgeber): *From Frege To Gödel: A Source Book in Mathematical Logic, 1879-1931*, Seiten 525–581. Harvard University Press, Cambridge, Mass., 1967.

[Jec03] Jech, T.: *Set Theory: The Third Millenium Edition, Revised and Expanded*. Springer-Verlag, Berlin Heidelberg, 3. Auflage, 2003.

[Küh09] Kühlwein, D.: *A calculus for Proof Representation Structures*. Diplomarbeit, Rheinische Friedrich-Wilhelms-Universität Bonn, 2009.

[Kol08] Kolev, N.: *Generating Proof Representation Structures in the Project Naproche*. Magisterarbeit, Rheinische Friedrich-Wilhelms-Universität Bonn, 2008.

[Kow74] Kowalski, R.: *Predicate Logic as a Programming Language*. Information Processing, 74:569–574, 1974.

[Kow79] Kowalski, R.: *Algorithm = logic + control*. Commun. ACM, 22(7):424–436, 1979.

[Kow88] Kowalski, R.: *The early years of logic programming*. Commun. ACM, 31(1):38–43, 1988.

[Llo84] Lloyd, J. W.: *Foundations of logic programming*. Springer-Verlag, Berlin, 1984.

[Llo87] Lloyd, J. W.: *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[MM82] Martelli, A. und U. Montanari: *An Efficient Unification Algorithm*. ACM Trans. Program. Lang. Syst., 4(2):258–282, 1982.

[NM95] Nilsson, U. und J. Maluszynski: *Logic, Programming and Prolog*. John Whily & Sons Ltd., New York, NY, USA, 2. Auflage, 1995.

[NP] Naproche-Projekt. http://www.naproche.net.

[PR97] Pedreschi, D. und S. Ruggieri: *Verification of Logic Programs*. Technischer Bericht, University of Pisa, 1997.

[Rau08] Rautenberg, W.: *Einführung in die Mathematische Logik*. Vierweg + Teubner, Wiesbaden, 3. Auflage, 2008.

[Rob65] Robinson, J. A.: *A machine-oriented logic based on the resolution principle*. J. of the ACM, 12:23–41, 1965.

[Sch00] Schöning, U.: *Logik für Informatiker*. Spektrum Akademischer Verlag, Heidelberg, 5. Auflage, 2000.

[SG96] Stroetmann, K. und T. Glass: *A Declarative Semantics for the Prolog Cut Operator*. In: *ELP '96: Proceedings of the 5th International Workshop on Extensions of Logic Programming*, Seiten 255–271, London, UK, 1996. Springer-Verlag.

[SP] SWI-Prolog. http://www.swi-prolog.org/.

[vEK76] Emden, M.H. van und R.A. Kowalski: *The Semantics of Predicate Logic as a Programming Language*. Journal of the ACM, 23(4):733–742, 1976.

[Wie] Wielemarker, J.: *SWI-Prolog reference manual*. Verfügbar auf: http://www.swi-prolog.org/pldoc/refman/.

[WR63] Whitehead, A. N. und B. Russell: *Principia Mathematica*. Cambridge University Press, Cambridge, Nachdruck der 1927 erschienenen 2. Auflage, 1963.

Index