Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Software wird in immer mehr Bereichen des täglichen Lebens eingesetzt. Sie findet sich in der Luftfahrt, in Ampelsteuerungen aber auch in Haushaltsgeräten wieder. Durch Fehler in der Software kann großer Schaden entstehen, weshalb korrekte Software sehr wichtig ist.

Die formalen Methoden bieten durch Beweise über Programme die Möglichkeit, die Korrektheit von Software nachzuweisen. Für einen Programmbeweis müssen die Anforderungen, die das Programm erfüllen soll, in formalen Spezifikationen gegeben sein. Um eine formale Spezifikation zu erhalten, müssen die informell gegebenen Anforderungen an ein Programm in Formeln ausgedrückt werden. Diese Formeln bilden dann die formale Spezifikation. Bei der Umsetzung von informellen Anforderungen in eine formale Spezifikation (durch den Menschen) können Fehler entstehen. Fehler liegen in formalen Spezifikationen dann vor, wenn die formale Spezifikation nicht genau die informellen Anforderungen beschreibt. Um Fehler in formalen Spezifikationen auszuschließen, führt man eine Validierung durch.

Bei der Validierung versucht man Aussagen, die in den informellen Anforderungen gelten, in der formalen Spezifikation zu beweisen. Gilt eine solche Aussage in der formalen Spezifikation nicht, liegt ein Fehler vor. Um herauszufinden, ob eine Aussage nicht gültig ist, kann man versuchen, ein Gegenbeispiel zu finden. Wie die formale Spezifikation liegt die Aussage für die Validierung, als eine Formel vor. Gibt es eine Belegung der Variablen dieser Formel, so daß die negierte Aussage gilt, hat man ein Gegenbeispiel gefunden. Wir wollen eine Methode erarbeiten, mit der man eine Gegenbeispiel-Suche in einen Beweiskalkül integrieren kann. Da diese Methode in das Spezifikations- und Verifikationssystem KIV integriert werden soll, hat die Verwendung eines Beweiskalküls für die Gegenbeispiel-Suche den Vorteil, daß in KIV kein zusätzlicher Kalkül für die Gegenbeispiel-Suche eingebaut werden muß.

Bei einem Beweisversuch für eine Formel treten meist mehrere Fälle auf. Kann man alle Fälle beweisen, d.h. man kann alle Fälle zu **true** reduzieren, hat man einen Beweis gefunden. Reduziert man aber einen der Fälle zu **false**, kann kein Beweis mehr für die Formel gefunden werden. Für die Gegenbeispiel-Suche versuchen wir deshalb, Fallunterscheidungen über die Belegungen der Variablen zu machen, um einen der Fälle zu **false** zu reduzieren. Gelingt dies, kann es keinen Beweis geben und wir haben die Existenz eines Gegenbeispiels gezeigt. Die Existenz eines Gegenbeispiels besagt nur, daß ein Fehler vorliegt, sie gibt jedoch keinen Hinweis auf die Fehlerursache. Deshalb wollen wir zusätzlich die konkreten Belegungen für die Variablen berechnen, die zu dem Gegenbeispiel geführt haben. Die Belegungen können aus dem Beweisversuch, der zu einem Gegenbeispiel führt, berechnet werden, indem man die Variablenzuweisungen aus den Fallunterscheidungen aufsammelt.

Der zweite wichtige Aspekt bei der Fehlersuche ist, den Fehler zu lokalisieren. Versucht man in einer fehlerhaften Spezifikation eine Validierungsaussage formal zu beweisen, anstatt sofort ein Gegenbeispiel für diese Aussage zu suchen, entsteht während des Beweisversuchs eine Formel, die in der Spezifikation nicht gilt. Der Benutzer weiß aber nicht, daß dies der Fall ist, denn es könnte für diese Formel auch einen Beweis geben, der nur noch nicht gefunden wurde. Damit man sich überzeugen kann, daß diese Formel nicht gültig ist, startet man eine Gegenbeispiel-Suche. Wenn

die Gegenbeispiel-Suche ein Gegenbeispiel findet, ist die Formel tatsächlich falsch. Nun hat man das Problem, daß man ein Gegenbeispiel für eine Formel mitten in einem Beweisversuch hat, jedoch nicht für die Ausgangsformel des Beweises, welche die Validierungsaussage darstellt. Eine Rückverfolgung soll nun ein Gegenbeispiel von Formel zu Formel so anpassen, daß es ein Gegenbeispiel für vorhergehende Formeln im Beweis darstellt. Gelangt die Rückverfolgung an den Beweisanfang des Beweisversuchs, erhält man ein Gegenbeispiel für die Validierungsaussage, und kann damit den Fehler in der Spezifikation finden. Ähnlich wie bei der Berechnung der konkreten Variablenbelegung für das Gegenbeispiel, sammelt man für die Rückverfolgung die im Beweis vorhandenen Bedingungen an die Variablen auf, und paßt damit das Gegenbeispiel für die vorhergehende Formel an.

Neben Spezifikationsfehlern gibt es bei der Programmverifikation noch weitere Fehlerquellen. So kann das Programm, für das man einen Beweis sucht, fehlerhaft sein, es kann während des Beweisens ein falsche Entscheidung über den Beweisfortgang getroffen worden sein oder man versucht ein inkorrektes Lemma zu beweisen. In allen Fällen gelangt man bei einem Beweisversuch an eine Formel, die man nicht mehr beweisen kann. Dann startet man die Gegenbeispiel-Suche, die ein Gegenbeispiel für diese Formel liefert. Die Rückverfolgung berechnet daraus ein Gegenbeispiel für die frühestmögliche Stelle im Beweisversuch, für die ein Gegenbeispiel existiert. Diese Stelle ist nicht immer der Beweisanfang, denn hat man während eines Beweisversuchs eine falsche Entscheidung getroffen, kann es geschehen, daß für die Formeln nach dieser Entscheidung Gegenbeispiele existieren, obwohl die Formeln vor dieser falschen Entscheidung gültig sind. An dieser Stelle muß die Rückverfolgung dann abbrechen. Bei Spezifikationsfehlern, Programmfehlern und falschen Lemmata kann die Rückverfolgung immer ein Gegenbeispiel für den Beweisanfang finden. Mit Hilfe des Gegenbeispiels versucht man dann, die Fehlerursache herauszufinden, den Fehler zu beseitigen und startet schlußendlich erneut einen Beweisversuch. Durch diesen Zyklus beseitigt man Schritt für Schritt alle Fehler.

Einteilung der Arbeit

Das folgende Kapitel verdeutlicht an einem kleinen Beispiel, wie die Gegenbeispiel-Suche in den Arbeitsablauf eines Beweisversuchs eingegliedert werden soll und welche sonstigen Anforderungen an die Gegenbeispiel-Suche gestellt werden müssen. Im Kapitel 3 werden fehlerhafte Beispiele vorgestellt, für die Gegenbeispiele gefunden werden sollen. Diese Beispiele werden zum Test verschiedener Methoden eingesetzt, die Gegenbeispiele finden sollen. Im Kapitel 4 wird untersucht, ob automatische Theorembeweiser geeignet sind, Gegenbeispiele zu finden. Die theoretischen Grundlagen für eine Gegenbeispiel-Suche durch Beweisfortführung, welche in KIV integriert wurde, werden in Kapitel 5 geschaffen. Die Integration zeigt Kapitel 6, und in Kapitel 7 werden Experimente mit dieser Methode besprochen. Schlußendlich folgt in Kapitel 8 eine Zusammenfassung der Arbeit und ein Ausblick, welche weiteren Möglichkeiten die entwickelte Methode für die Gegenbeispiel-Suche bietet.

Kapitel 2

Ein einführendes Beispiel

Wir wollen an einem Beispiel verdeutlichen, wie die Gegenbeispiel-Suche helfen kann, einen Fehler in einem Lemma zu finden. Dazu starten wir einen Beweisversuch für das fehlerhafte Lemma. Wenn wir in einen Fall des Beweises gelangen, den wir vermutlich nicht beweisen können, starten wir die Gegenbeispiel-Suche. Ist die Suche erfolgreich, liegt mit Sicherheit ein Fehler vor und das Gegenbeispiel gibt uns Hinweise darauf, wie der Fehler aussehen kann.

Anhand dieses Beispiels wollen wir dann Anforderungen erarbeiten, die an die Gegenbeispiel-Suche gestellt werden müssen, damit sie ein gutes Hilfsmittel für die Fehlersuche und Fehlerbeseitigung ist.

Damit das Beispiel leichter verständlich wird, geben wir zuvor eine kleine Einführung in die algebraische Spezifikationsmethode, wie sie in KIV verwendet wird und erläutern die Notationen, die wir während der Arbeit verwenden werden.

2.1 Formale Grundlagen

In diesem Abschnitt legen wir einige Notationen fest, die im Laufe der Arbeit verwendet werden. Als erstes gehen wir auf algebraische Spezifikationen ein. Es werden nur die Punkte behandelt, die für diese Arbeit wichtig sind. Für eine weitergehende Einführung in algebraische Spezifikationen siehe [Par93], [Rei96] und [Wir90].

2.1.1 Algebraische Spezifikation

Eine algebraische Spezifikation $SP = (\Sigma, T, C)$ besteht aus der Signatur $\Sigma = (S, OP)$, den Axiomen $T \subseteq For(\Sigma, X)$ und $C \subseteq F$ den Konstruktoren, die aus der **generated by** Klausel einer Spezifikation hervorgehen (siehe Abbildung 2.1). Dabei ist S eine endliche Menge von Sorten, $OP = F \cup P$ eine Menge mit den Funktionssymbolen F und den Prädikatsymbolen P. $For(\Sigma, X)$ ist die Menge der Formeln erster Ordnung, die über der Signatur Σ und den Variablen X gebildet werden können.

Eine Algebra $\mathcal{A}$ heißt erzeugt, $\mathcal{A} \in Gen(\Sigma, C)$, falls es für jedes Element a der Trägermenge von $\mathcal{A}$ einen Grundterm τ mit $a = \tau_{\mathcal{A}}$ gibt, wobei $\tau_{\mathcal{A}}$ der Wert des Terms τ in $\mathcal{A}$ ist. $\mathcal{A}$ ist ein Modell von T, d.h. $\mathcal{A} \models T$, wenn in $\mathcal{A}$ die Axiome T gelten. Die Modelle $Mod(SP) = \{\mathcal{A} \in Gen(\Sigma, C) : \mathcal{A} \models T\}$ der Spezifikation SP sind diejenigen Algebren, die erzeugt sind und in denen die Axiome gelten.

Es gibt die Möglichkeit, nicht erzeugte Sorten als Parametersorten einer Funktion, deren Zielsorte erzeugt ist, zu verwenden. Werden Parametersorten verwendet, heißt eine Algebra $\mathcal{A}$ erzeugt, wenn es für jedes Trägerelement der erzeugten Sorten $a \in \mathcal{A}$ einen Term $\tau \in T(\Sigma_C, X_0)$ und $v : X_0 \to \mathcal{A}$ mit $a = \tau_{\mathcal{A},v}$ gibt. Dabei ist Σ_C die Konstruktorsignatur, X_0 die Menge der Variablen über den Parametersorten und v eine Abbildung der Variablen in die Trägermenge von $\mathcal{A}$. Parametersorten fungieren als Platzhalter für andere Sorten, d.h. eine Parametersorte kann später durch

Nat =
specification
sorts nat;
functions
0 : $\rightarrow$ nat;
. +1, . -1 : nat $\rightarrow$ nat;
. + ., . * . : nat $\times$ nat $\rightarrow$ nat;
predicates
. | . : nat $\times$ nat;
prime : nat;
variables n, m, k: nat;
nat **generated by** 0, +1;
axioms
ax-01: $\neg\ 0 = n\ +1$,
ax-02: $n\ +1\ -1\ = n$,
ax-03: $n + 0 = n$,
ax-04: $n + m\ +1\ = (n + m)\ +1$,
ax-05: $n * 0 = 0$,
ax-06: $n * m\ +1\ = n * m + n$,
ax-07: $n \mid m \leftrightarrow \exists\ k.\ n * k = m$,
ax-08: $prime(n) \leftrightarrow\ \neg\ n = 0 \wedge \neg\ n = 0\ +1$
$\wedge\ \forall\ k.\ k \mid n \rightarrow k = 0\ +1\ \vee\ k = n$
end specification

Abbildung 2.1: Spezifikation: natürliche Zahlen

eine (erzeugte) Sorte ersetzt werden. Deshalb nennt man Spezifikationen, die Parametersorten enthalten, generisch oder parametrisiert.

Eine strukturierte Spezifikation ist eine Vereinigung $SP_1 + SP_2$ zweier Spezifikationen oder eine Anreicherung einer Spezifikation **enrich** SP **by** (S, F, X, T, C) mit zusätzlichen Sorten, Funktionen, Variablen, Axiomen und Konstruktoren für die neuen Sorten. Für die Semantik strukturierter Spezifikationen siehe [Rei96].

Sehen wir uns ein Beispiel einer Spezifikation, wie sie in KIV aussieht, an (siehe Abb. 2.1). Die Sortenmenge der Spezifikation ist $S = \{nat\}$, die Funktionssymbole sind $F = \{0,\ +1,\ -1, +, *\}$ und die Prädikatsymbole sind $P = \{|, prime\}$. Die Axiomenmenge T ist die Formelmenge nach dem Schlüsselwort **axioms**. Die Konstruktoren $C = \{0,\ +1\}$ gehen aus der Erzeugtheitsklausel **generated by** hervor. Es werden keine Parametersorten verwendet. Die Spezifikation der natürliche Zahlen werden wir öfters in den Beispielen verwenden. Sie enthält die Funktion -1, die für das Argument 0 nicht spezifiziert ist. Um die Lesbarkeit zu erhöhen vereinbaren wir, daß für die Konstruktorterme 0, 0 +1, 0 +1 +1... die Zahlen 0, 1, 2... geschrieben werden.

2.1.2 Notation

Wenn wir allgemein über Spezifikationen reden, verwenden wir für Funktionen die Symbole f und g. Um nicht immer zwischen Funktionen und Prädikaten unterscheiden zu müssen, fassen wir ab jetzt Prädikate als boolesche Funktionen auf und meinen, wenn wir von Funktionen sprechen, immer Funktionen und Prädikate. Variablen werden mit x, y, z bezeichnet. Für Variablen, deren Sorte eine Parametersorte ist, verwenden wir die Bezeichner *ele* bzw. ele_0. Für Terme benutzen wir die griechischen Kleinbuchstaben ρ und τ, für Formeln φ, ψ und χ. Griechische Großbuchstaben Γ und Δ benutzen wir für Formelmengen. Für $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ bzw. $\Delta = \{\psi_1, \ldots, \psi_m\}$ ist $\bigwedge \Gamma = \varphi_1 \wedge \ldots \wedge \varphi_n$ und $\bigvee \Delta = \psi_1 \vee \ldots \vee \psi_m$. Eine **Sequenz** hat die Form $\Gamma \vdash \Delta$ und die Bedeutung $\bigwedge \Gamma \rightarrow \bigvee \Delta$. Dabei nennen wir Γ den **Antezedent** und Δ den **Sukzedent** der Sequenz. Wenn

aus dem Kontext die Bedeutung klar ist, schreiben wir statt $\bigwedge \Gamma$ auch Γ bzw. statt $\bigvee \Delta$ schreiben wir Δ. Für eine Sequenz $\Gamma \vdash \Delta$ ist die Formeldarstellung $\mathit{formula}(\Gamma \vdash \Delta) = \bigwedge \Gamma \rightarrow \bigvee \Delta$, und mit der eingeführten Kurzschreibweise ist $\mathit{formula}(\Gamma \vdash \Delta) = \Gamma \rightarrow \Delta$.

Die Funktionen **vars**, **free**, **param** und **gen** können sowohl auf Formeln als auch auf Sequenzen angewendet werden. Für eine Formel φ ist $\mathit{vars}(\varphi)$ die Menge der Variablenbezeichner von φ. Die Menge $\mathit{free}(\varphi)$ ergibt die freien Variablen der Formel. Freie Variablen, deren Sorten Parametersorten sind, erhält man mit $\mathit{param}(\varphi)$ und mit $\mathit{gen}(\varphi)$ diejenigen freien Variablen, deren Sorten erzeugt sind. Für Sequenzen seq ist $\mathit{vars}(\mathit{seq}) = \mathit{vars}(\mathit{formula}(\mathit{seq}))$ und liefert damit alle Variablen der Sequenz. Entsprechendes gilt für *free*, *param* und *gen*. Die Funktion **sort** ergibt, auf Terme angewandt, die Sorte der Terme. Für eine Funktion f liefert $\mathit{sort}(f)$ die Zielsorte der Funktion und für eine Variable x die Sorte der Variable. Eine **neue Variable** x bezüglich einer Formel φ (bzw. einer Sequenz) ist eine Variable, die nicht in den Variablen der Formel (Sequenz) vorkommt, d.h. $x \notin \mathit{vars}(\varphi)$.

Eine **Regel** hat die Form $\frac{p_1, \ldots p_n}{c}$. Dabei ist c die **Konklusio** der Regel und p_i sind die **Prämissen**. Die Konklusio c und die Prämissen p_i sind Sequenzen. Die in KIV verwendeten Regeln sind in Anhang B aufgelistet. Eine **Ableitung** ist eine mehrfache Anwendung von Regeln. Dabei werden Regeln immer nur auf Prämissen angewendet. Da die Regel mehrere Prämissen haben können, entsteht eine verzweigte Struktur, ein sogenannter **Ableitungs-** oder **Beweisbaum**, dessen Knoten Sequenzen sind. Eine Regeln kann man als einen Ableitungsbaum mit nur einem Ableitungsschritt ansehen. Deshalb nennen wir auch die Wurzel des Ableitungsbaumes Konklusio und die Blätter des Ableitungsbaumes Prämissen.

Ein **Pfad** in einem Ableitungsbaum ist eine Folge von Sequenzen, die durch Regelanwendungen verbunden sind. Die erste Sequenz der Folge ist die Konklusio des Ableitungsbaumes. Der Pfad zu einer Prämisse p_i ist die Folge von Sequenzen, deren erste Sequenz die Wurzel und deren letzte Sequenz die Prämisse p_i ist.

Zum Abschluß dieses Abschnittes zeigen wir noch eine weitere Spezifikation, die in den Beispielen immer wieder verwendet wird. Es handelt sich um eine generische Listenspezifikation (siehe Abbildung 2.2), die durch die Sorte **list** dargestellt wird. Auf der Parametersorte **elem** muß eine Ordnung definiert sein, um geordnete Listen erzeugen zu können. Damit die Parametersorte **elem** in der Listenspezifikation verwendet werden kann, muß sie der Spezifikation durch das Schlüsselwort **using** bekannt gemacht werden. Die Liste besteht aus den Konstruktoren @, für die leere Liste, und $\oplus$, der an eine Liste vorne ein neues Element anfügt. Die Operatoren *.first* und *.last* selektieren aus einer Liste das erste bzw. letzte Element. Der Operator $\odot$ hängt an die erste Liste eine zweite Liste an und *.rest* liefert für eine Liste dieselbe Liste ohne deren erstes Element zurück. Die Funktion *merge* fügt zwei geordnete Listen zu einer geordneten Liste zusammen, während die Funktion *sort* mit Hilfe von *merge* eine Liste sortiert. Das Prädikat *ord* trifft auf eine Liste zu, wenn sie bezüglich der $<$-Ordnung geordnet ist.

Sieht man sich die Spezifikation genauer an, erkennt man, daß die Spezifikation des Prädikates *ord* und der Funktion *merge* nicht zusammenpassen. Bekommt die Funktion *merge* zwei geordnete Listen, die beide das Element *ele* besitzen, übergeben, wird das Element *ele* durch das Axiom ax-10 zweimal in die Ergebnisliste übernommen ($\mathit{merge}(\mathit{ele} \oplus x,\ \mathit{ele} \oplus y) \overset{\text{ax-10}}{\Leftrightarrow} \underline{\mathit{ele}} \oplus \mathit{merge}(\underline{\mathit{ele}} \oplus x,\ y)$). Für die Ergebnisliste kann deshalb nicht das Prädikat *ord* gelten. Auf diesen Spezifikationsfehler wird in weiteren Beispielen noch eingegangen.

2.1.3 Programme in KIV

In der Dynamische Logik, die in KIV verwendet wird, gibt es zusätzlich zu den prädikatenlogischen Formeln noch die Konstrukte $[\alpha]$ und $\langle\alpha\rangle$, wobei α ein Pascal-artiges Programm ist. Für ein Programm α bedeutet:

$[\alpha]\varphi$: wenn α terminiert, dann gilt nach Ausführung von α die Nachbedingung φ

$\langle\alpha\rangle\varphi$: α terminiert und es gilt nach der Ausführung von α die Nachbedingung φ

List =
specification
using elem
sorts list;
functions
$@$: $\rightarrow$ list;
$. \oplus .$: elem $\times$ list $\rightarrow$ list;
$. .first$, $. .last$: list $\rightarrow$ elem;
$. \odot .$, $merge$: list $\times$ list $\rightarrow$ list;
$. .rest$, $sort$: list $\rightarrow$ list;
predicates
ord : list;
variables
x, y, z : list;
ele, ele_0 : elem;
list **generated by** $@$, $\oplus$;
axioms
ax-01: $(ele \oplus x).first = ele$,
ax-02: $(ele \oplus x).rest = x$,
ax-03: $(ele \oplus @).last = ele$,
ax-04: $(ele \oplus x).last = x.last$,
ax-05: $@ \odot x = x$,
ax-06: $ele \oplus x \odot y = ele \oplus (x \odot y)$,
ax-07: $merge(@, y) = y$,
ax-08: $merge(x, @) = x$,
ax-09: $ele < ele_0 \rightarrow merge(ele \oplus x, ele_0 \oplus y) = ele \oplus merge(x, ele_0 \oplus y)$,
ax-10: $\neg\ ele < ele_0 \rightarrow merge(ele \oplus x, ele_0 \oplus y) = ele_0 \oplus merge(ele \oplus x, y)$,
ax-11: $sort(@) = @$,
ax-12: $sort(x \odot y) = merge(sort(x), sort(y))$,
ax-13: $ord(@)$,
ax-14: $ord(ele \oplus @)$,
ax-15: $ord(ele \oplus ele_0 \oplus x) \leftrightarrow ele < ele_0 \wedge ord(ele_0 \oplus x)$
end specification

Abbildung 2.2: Spezifikation: generische Listen

Im folgenden bezeichnen wir Formeln der Dynamischen Logik mit **DL-Formeln**. Enthalten die Formeln nur prädikatenlogische Konstrukte, bezeichnen wir sie mit **PL-Formeln**. Die Buchstaben α und β stehen für Programme, die folgende Form haben können:

skip	die leere Anweisung
abort	die niemals terminierende Anweisung
$x := \tau$	Zuweisung
if ε **then** α **else** β	Konditional
$\alpha;\ \beta$	zusammengesetzte Anweisung
while ε **do** α	while-Schleife
var $x_1 = \tau_1, \ldots, x_n = \tau_n$ **in** **begin** α **end**	Variablendeklaration für das Programm α; die Variable x_i wird dabei mit dem Term τ_i initialisiert
$prog(\underline{\tau};\ \textbf{var}\ \underline{z})$	Prozeduraufruf der Prozedur *prog* mit den Termen $\underline{\tau}$ und den Variablen $\underline{z}$
proc δ **in begin** α **end**	Prozedurdeklaration am Anfang des Programms α, wobei δ eine Liste von Deklarationen ist

Die Deklarationsliste δ hat folgende Form:

$$\begin{array}{rcl} \delta & = & prog_1(\underline{x_1};\ \textbf{var}\ \underline{y_1})\ \textbf{begin}\ \alpha_1\ \textbf{end} \\ & & \vdots \qquad\qquad\qquad\qquad \vdots \\ & & prog_n(\underline{x_n};\ \textbf{var}\ \underline{y_n})\ \textbf{begin}\ \alpha_n\ \textbf{end} \end{array}$$

Die Semantik der *value*- und *var*-Parameter ist wie in Pascal. Die Programmregeln der Dynamischen Logik befinden sich im Anhang B.1.

2.2 Das Beispiel

In diesem Abschnitt wollen wir an einem fehlerhaften Lemma zeigen, wie die Gegenbeispiel-Suche helfen kann, den Fehler zu finden. Eine pessimistische Strategie für die Gegenbeispiel-Suche wäre, ein Gegenbeispiel zu suchen, bevor man das Lemma zu beweisen versucht. Dieses Vorgehen hätte den Vorteil, daß keine Energie in einen Beweisversuch, der keinen Erfolg bringen kann, gesteckt werden muß, wenn für das Lemma ein Gegenbeispiel gefunden wird.

Diese Strategie wollen wir nicht verfolgen, sondern wir wählen eine optimistischere. Erst wenn bei einem Beweisversuch der Benutzer an eine Stelle gelangt, an der er vermutet, daß der Beweis nicht erfolgreich zu Ende geführt werden kann, startet er die Gegenbeispiel-Suche. Bei Fehlern in Programmen ist es meist so, daß nur in einem bestimmten Programmzweig ein Fehler auftritt. Der Beweis kann dann nur für diesen Zweig nicht geführt werden.

Ähnlich verhält es sich bei Fehlern in der Spezifikation oder in Lemmata. Dort werden z.B. Vorbedingungen vergessen oder zu schwach gewählt. Beginnt man einen Beweis, können Aussagen, die nicht von dieser Vorbedingung abhängen, bewiesen werden. Ein Beweisast kann nur dann nicht zu Ende bewiesen werden, wenn die Aussage in diesem Ast von der fehlerhaften Vorbedingung abhängt. Startet man die Gegenbeispiel-Suche erst, wenn man in einen Beweisast gelangt, der nicht mehr bewiesen werden kann, findet die Gegenbeispiel-Suche schneller ein Gegenbeispiel, da sie schon Informationen hat, in welchem Bereich sie die Variablenbelegungen suchen muß. Außerdem erhält der Benutzer Informationen, die auf die Fehlerursache hinweisen, denn er weiß, in welchem Beweisast der Fehler aufgetreten ist.

Für das Beispiel benutzen wir die in Abbildung 2.2 gezeigte Spezifikation der generischen Listen, wobei wir für die generischen Elemente die natürlichen Zahlen verwenden (siehe Abbildung 2.1). Das Lemma soll ein Kriterium angeben, unter dem die Konkatenation zweier geordneter Listen x und y wieder geordnet sind. Dies ist z.B. der Fall, wenn das *letzte* Element der ersten

Liste x kleiner als das *erste* Element der zweiten Liste y ist. Beim Formulieren des Lemmas ist ein (Schreib-)Fehler unterlaufen.

Beispiel 2.1
$\vdash ord(x) \land ord(y) \land x.first < y.first \rightarrow ord(x \odot y)$

Diese Sequenz besagt, daß die Ergebnisliste geordnet ist, wenn zwei geordnete Listen aneinandergehängt werden, wobei das *erste* Element der ersten Liste kleiner als das *erste* Element der zweiten Liste ist. Dies ist natürlich nicht korrekt, und wir haben damit ein fehlerhaftes Lemma formuliert.

Der Beweisversuch

Wir gehen nun davon aus, daß uns der Fehler nicht bekannt ist, und starten einen Beweisversuch.

---

1) structural induction for y

$$ord(x),\ ord(y),\ x.first < y.first \vdash ord(x \odot y)$$

Die Aussage soll für alle Listen x und y gelten, deshalb benötigt man einen Induktionsbeweis über x und y. Zuerst führen wir die strukturelle Induktion über y durch. Da Listen leer sein oder durch hinzufügen eines Elements an eine bestehende Liste entstehen können, ergeben sich daraus zwei Fälle für die Liste y. Im ersten Fall ist y die leere Liste. Wenn an x die leeren Liste angehängt wird, ist die Ergebnisliste wieder x. Da x aber nach Voraussetzung geordnet ist, ist auch die Ergebnisliste geordnet. Dieser Fall kann also geschlossen werden und wird, um den Beweis zu verkürzen, nicht aufgeführt. Es bleibt der zweite Fall, in dem y dadurch entsteht, daß zur Liste y_0 das Element n hinzugefügt wird. Der $\forall$-Quantor in der Sequenz stellt die Induktionshypothese dar.

---

2) structural induction for x

$$\begin{array}{l} ord(x),\ ord(n \oplus y_0),\ x.first < n, \\ \forall\, x.ord(x) \land ord(y_0) \land x.first < y_0.first \rightarrow ord(x \odot y_0) \\ \vdash ord(x \odot n \oplus y_0) \end{array}$$

Nun führen wir die strukturelle Induktion über die Liste x durch. Es entstehen wiederum zwei Fälle. Ist x die leere Liste, so ist die Konkatenation von x und $n \oplus y_0$ gleich $n \oplus y_0$ und diese Liste ist nach Voraussetzung schon geordnet. Dieser Fall kann wieder geschlossen werden und wird ebenfalls nicht aufgeführt. Ist x zusammengesetzt, erhalten wir den Fall 3).

---

3) all left 5 with y_0, n_0

$$\begin{array}{l} ord(n \oplus x_0),\ ord(n_0 \oplus y_0),\ n < n_0, \\ \forall\, x.ord(x) \land ord(y_0) \land x.first < y_0.first \rightarrow ord(x \odot y_0), \\ \forall\, y_0,\ n. \qquad ord(x_0) \\ \qquad\qquad \land\ ord(n \oplus y_0) \\ \qquad\qquad \land\ x_0.first < n \\ \qquad\qquad \land\ (\forall\, x.ord(x) \land ord(y_0) \land x.first < y_0.first \rightarrow ord(x \odot y_0)) \\ \qquad\quad \rightarrow ord(x_0 \odot n \oplus y_0) \\ \vdash ord((n \oplus x_0) \odot n_0 \oplus y_0) \end{array}$$

Dieser Beweisast repräsentiert den Fall, in dem die Liste x aus $n \oplus x_0$ und y aus $n_0 \oplus y_0$ zusammengesetzt sind. Die $\forall$-quantifizierten Formeln in der Sequenz sind die beiden Induktionshypothesen. Um in diesem Fall weiterrechnen zu können, muß eine Induktionshypothese angewendet werden. Wir wenden die zweite Induktionshypothese mit y_0 und n_0 an. Die Induktionshypothese wird für den weiteren Beweis nicht mehr benötigt und kann deshalb entfernt werden. Nach Anwendung der Induktion erhalten wir die Sequenz

---

4) case distinction left 7

$$
\begin{array}{l}
\mathit{ord}(x_0),\ \mathit{ord}(y_0),\ \mathit{ord}(n \oplus x_0),\ \mathit{ord}(n_0 \oplus y_0),\ n < n_0, \\
\quad (x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0)) \wedge x_0.\mathit{first} < n_0 \\
\rightarrow \mathit{ord}(x_0 \odot n_0 \oplus y_0), \\
\forall\, x.x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0) \\
\vdash \mathit{ord}((n \oplus x_0) \odot n_0 \oplus y_0)
\end{array}
$$

Die entstandene Implikation im Antezedenten repräsentiert die Induktionshypothese für die Variablen y_0 und n_0. Mit Hilfe der Induktionshypothese müssen wir nun die Sequenz beweisen. Um den Beweis zu strukturieren, lösen wir die Implikation der Induktionshypothese auf und erhalten die Fälle 5) und 8).

---

5) case distinction right 1

$$
\begin{array}{l}
\mathit{ord}(x_0),\ \mathit{ord}(y_0),\ \mathit{ord}(n \oplus x_0),\ \mathit{ord}(n_0 \oplus y_0),\ n < n_0, \\
\forall\, x.x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0) \\
\vdash (x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0)) \wedge x_0.\mathit{first} < n_0, \\
\mathit{ord}((n \oplus x_0) \odot n_0 \oplus y_0)
\end{array}
$$

In diesem Fall müssen wir $(x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0)) \wedge x_0.\mathit{first} < n_0$ zeigen. Dazu unterteilen wir den Beweis wieder und lösen durch eine Fallunterscheidung die Konjunktion im Sukzedenten auf. Es entstehen die Fälle 6) und 7).

•6) all left 8 with x

$$
\begin{array}{l}
\mathit{ord}(x_0),\ \mathit{ord}(y_0),\ \mathit{ord}(n \oplus x_0),\ \mathit{ord}(n_0 \oplus y_0),\ n < n_0, \\
\forall\, x.x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0) \\
\vdash x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0),\ \mathit{ord}((n \oplus x_0) \odot n_0 \oplus y_0)
\end{array}
$$

Den ersten Fall der Konjunktion können wir lösen, indem wir den $\forall$-Quantor im Antezedenten mit x instantiieren. Dann erhalten wir im Antezedenten und im Sukzedenten dieselbe Formel und dieser Beweisast ist geschlossen.

◦7) (from 5) open

$$
\begin{array}{l}
\mathit{ord}(x_0),\ \mathit{ord}(y_0),\ \mathit{ord}(n \oplus x_0),\ \mathit{ord}(n_0 \oplus y_0),\ n < n_0, \\
\forall\, x.x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0) \\
\vdash x_0.\mathit{first} < n_0,\ \mathit{ord}((n \oplus x_0) \odot n_0 \oplus y_0)
\end{array}
$$

In diesem Beweisast müssen wir $x_0.\mathit{first} < n_0$ zeigen. Es ist aber völlig unklar, wie dies geschehen soll. Für die ganze Sequenz gibt es keine geeignete Strategie, um den Beweis fortzuführen. Da nicht sicher ist, ob diese Stelle überhaupt beweisbar ist, könnte eine Gegenbeispiel-Suche gestartet werden. Zuerst führen wir jedoch den restliche Beweis noch zu Ende.

•8) (from 4) simplifier

$$
\begin{array}{l}
\mathit{ord}(x_0 \odot n_0 \oplus y_0),\ \mathit{ord}(x_0),\ \mathit{ord}(y_0),\ \mathit{ord}(n \oplus x_0),\ \mathit{ord}(n_0 \oplus y_0),\ n < n_0, \\
\forall\, x.x.\mathit{first} < y_0.\mathit{first} \wedge \mathit{ord}(x) \rightarrow \mathit{ord}(x \odot y_0) \\
\vdash \mathit{ord}((n \oplus x_0) \odot n_0 \oplus y_0)
\end{array}
$$

Nach Induktionsannahme ist die Liste $x_0 \odot n_0 \oplus y_0$ geordnet. Da auch die Liste $n \oplus x_0$ geordnet ist, ist n kleiner als alle Elemente in x_0 und damit kleiner als alle Elemente in $x_0 \odot n_0 \oplus y_0$. Deshalb ist auch $(n \oplus x_0) \odot n_0 \oplus y_0$ geordnet und der Beweisast kann geschlossen werden.

Es bleibt also nur ein offener Beweisast, nämlich der Schritt 7). Der Mechanismus zur Konstruktion von Gegenbeispielen müßte nun an dieser Stelle versuchen, ein Gegenbeispiel zu finden.

Die Gegenbeispiel-Suche

Finden wir eine Belegung, mit der die Aussage der Sequenz nicht gilt, kann die Sequenz nicht bewiesen werden und wir haben ein Gegenbeispiel gefunden. Um für den Schritt 7) des obigen Beweisversuchs ein Gegenbeispiel zu finden, versuchen wir die Variablen so zu belegen, daß eine falsche Behauptung entsteht.

---

1) $y_0 = @$

$$\begin{array}{l} \quad ord(x_0), ord(y_0), ord(n \oplus x_0), ord(n_0 \oplus y_0), n < n_0, \\ \quad \forall\, x.x.first < y_0.first \wedge ord(x) \rightarrow ord(x \odot y_0) \\ \vdash x_0.first < n_0, ord(n \oplus x_0 \odot n_0 \oplus y_0) \end{array}$$

Wenn wir für y_0 die leere Liste einsetzen, erhalten wir folgende Sequenz.

---

2) simplifier

$$\begin{array}{l} \quad ord(x_0), ord(@), ord(n \oplus x_0), ord(n_0 \oplus @), n < n_0, \\ \quad \forall\, x.x.first < @.first \wedge ord(x) \rightarrow ord(x \odot @) \\ \vdash x_0.first < n_0, ord(n \oplus x_0 \odot n_0 \oplus @) \end{array}$$

Nun können einige Vereinfachungsschritte vorgenommen werden. Da eine leere und eine einelementige Liste immer geordnet ist, fallen die Prädikate $ord(@)$ und $ord(n_0 \oplus @)$ aus der Sequenz weg. Der $\forall$-Quantor im Antezedenten kann wegfallen, da für alle Listen $x \odot @$ wieder x ist. Dadurch wird der Vordersatz der Implikation gleich dem Nachsatz und kann zu **true** vereinfacht werden.

---

3) $x_0 = n_1 \oplus @$

$$ord(x_0),\ ord(n \oplus x_0),\ n < n_0 \vdash x_0.first < n_0,\ ord(n \oplus x_0 \odot n_0 \oplus @)$$

Setzen wir für x_0 die leere Liste ein, kann der Ast geschlossen werden. Deshalb nehmen wir für die Liste x_0 an, daß sie einelementig ist und das Element n_1 sei. Da wir uns für die Belegung von n_1 noch alle Möglichkeiten offen lassen wollen, wählen wir eine Variable, die noch nicht in der Sequenz vorkommt.

---

4) simplifier

$$ord(n_1 \oplus @),\ ord(n \oplus n_1 \oplus @),\ n < n_0 \vdash n_1 \oplus @.first < n_0,\ ord(n \oplus n_1 \oplus @ \odot n_0 \oplus @)$$

Mit dieser Einschränkung kann die Sequenz wieder vereinfacht werden. Die Prädikate *ord* können aufgelöst werden, da keine Variablen mehr über Listen vorhanden sind. Deshalb können die Bedingungen *ord* auf Listen in <-Bedingungen über den natürlichen Zahlen ausgedrückt werden.

$\circ$5)

$$n < n_0,\ n < n_1 \vdash n_1 < n_0$$

Das Negat der Sequenz, die Sequenz $\vdash n < n_0 \wedge n < n_1 \wedge \neg\ n_1 < n_0$, ist mit den Belegungen $n = 0$, $n_0 = 1$ und $n_1 = 1$ erfüllbar und wir haben damit ein Gegenbeispiel gefunden. Die Gegenbeispiel-Suche müßte eigentlich für die Sequenz $n < n_0,\ n < n_1 \vdash n_1 < n_0$ die entsprechenden Belegungen für ein Gegenbeispiel suchen. Wir geben diese jedoch selbst an, um das Beispiel zu verkürzen.

An diesem Punkt wissen wir, daß ein Fehler vorliegt, aber wir haben noch keinen Hinweis darauf, wie der Fehler zustande kommt. Die Belegung der Variablen, die zum Gegenbeispiel geführt hat, könnte Hinweise auf die Fehlerursache geben.

Berechnung der Belegungen

Im Schritt 5) des Gegenbeispiel-Beweises haben wir die Bedingungen $n = 0$, $n_0 = 1$ und $n_1 = 1$. Davon ausgehend versuchen wir nun, die Bedingungen für den Schritt 1) des Gegenbeispiel-Beweises zu berechnen.

In Schritt 3) haben wir die Bedingung $x_0 = n_1 \oplus @$ und in Schritt 1) die Bedingung $y_0 = @$ eingeführt. Dadurch erhalten wir insgesamt die Bedingungen $n = 0$, $n_0 = 1$, $n_1 = 1$, $x_0 = 1 \oplus @$ und $y_0 = @$. Diese Belegungen ergeben ein Gegenbeispiel für die Formeln im Schritt 1) des Gegenbeispiel-Beweises und damit auch für die Formeln im Schritt 7) des Ausgangsbeweises. Um den Fehler im Ausgangsbeweis zu lokalisieren, führen wir eine Rückverfolgung durch.

Die Rückverfolgung

Vom Schritt 7) des Ausgangsbeweises muß nun das konkrete Gegenbeispiel noch bis zur Konklusio zurückberechnet werden. Das Gegenbeispiel läßt sich ohne Änderung bis zum Schritt 3) zurückverfolgen. Von Schritt 2) zu Schritt 3) wird eine Induktion über x durchgeführt. Für die Variable x wird im Schritt 3) der Term $n \oplus x_0$ eingesetzt. Bei der Rückberechnung wird diese Bedingung mit in das Gegenbeispiel aufgenommen. Da $x_0 = 1 \oplus @$ ist, können wir $x = 0 \oplus 1 \oplus @$ für die Belegung aufnehmen. Dasselbe geschieht von Schritt 1) zu Schritt 2). Dort wird für y der Term $n_0 \oplus y_0$ eingesetzt. Auch diese Bedingung muß mit in das Gegenbeispiel aufgenommen werden, wobei wir für $y = 1 \oplus @$ einsetzen. Für den Schritt 1) erhalten wir somit $n = 0$, $n_0 = 1$ $n_1 = 1$, $x = 0 \oplus 1 \oplus @$ und $y = 1 \oplus @$ als Belegung für das Gegenbeispiel. Wie sich leicht überlegen läßt, ist dies tatsächlich ein Gegenbeispiel für die Aussage

$$\vdash \mathit{ord}(x) \wedge \mathit{ord}(y) \wedge x.\mathit{first} < y.\mathit{first} \rightarrow \mathit{ord}(x \odot y).$$

Das zweite (bzw. letzte) Element der Liste x ist größer als das erste Element der Liste y. Deshalb ist die zusammengesetzte Liste nicht geordnet. Daraus läßt sich erkennen, daß die Bedingung $x.\mathit{first} < y.\mathit{first}$ eigentlich $x.\mathit{last} < y.\mathit{first}$ heißen müßte und wir haben den Fehler gefunden.

Dieses Beispiel sollte verdeutlichen, wir man ein Gegenbeispiel finden und mit dessen Hilfe den Fehler lokalisieren kann. Welche Anforderungen die Gegenbeispiel-Suche erfüllen sollte, wollen wir im nächsten Abschnitt besprechen. Dabei greifen wir auf die Erkenntnisse zurück, die wir aus dem obigen Beispiel gewonnen haben.

2.3 Anforderungen an die Gegenbeispiel-Suche

Wir haben bei dem Beweisversuch zu Beispiel 2.1 erst im Verlauf des Beweises bemerkt, daß die Aussage nicht korrekt ist. Dies ist bei interaktiven Beweisen die Regel. Deshalb sollte die Gegenbeispiel-Suche von einem aktuellen Beweis aus aktiviert werden können. Für die Prämisse des aktuellen Beweises, von der der Benutzer annimmt, daß sie nicht korrekt ist, soll ein Gegenbeispiel gesucht werden. Dieses Vorgehen hat den Vorteil, daß auch falsche Beweisentscheidungen entdeckt werden können, wenn diese aus einer beweisbaren Aussage eine unbeweisbare machen. Dann kann für eine Prämisse ein Gegenbeispiel gefunden werden, obwohl für die Konklusio keines existiert.

In KIV gibt es Mechanismen die es erlauben, fehlgeschlagene Beweise für die korrigierte Aussage wiederzuverwenden. Dies ist dann sinnvoll, wenn ein Teil des Beweises schon erfolgreich abgeschlossen wurde. Im Beispiel 2.1 sind schon 4 Beweisäste geschlossen. Für die korrigierte Aussage kann der fehlgeschlagene Beweis nachgespielt werden, und die Hoffnung ist, daß wieder nur ein offener Beweisast übrigbleibt, der nun aber bewiesen werden kann. Die übrigen Fälle sollten, mit Hilfe des fehlegeschlagenen Beweisversuchs, automatisch geschlossen werden. Damit der Beweisversuch zum Nachspielen zur Verfügung steht, darf ihn die Gegenbeispiel-Suche nicht verändern. Die Suche sollte deshalb als eine Art Nebenrechnung ablaufen.

Hat die Gegenbeispiel-Suche bestätigt, daß die Prämisse nicht beweisbar ist, kann es daran liegen, daß während des Beweises eine falsche Entscheidung getroffen wurde oder daß die Konklusio nicht beweisbar ist. Wenn die Konklusio nicht beweisbar ist, gilt die Aussage in der gegebenen Spezifikation nicht. Entweder ist die Aussage selbst falsch, oder die Spezifikation formalisiert nicht das Modell, das sich der Benutzer vorstellt. Dann liegt ein Spezifikationsfehler vor. In diesen beiden Fällen gibt es ein Gegenbeispiel für die Konklusio. Für den Benutzer wäre es hilfreich, wenn dies die Gegenbeispiel-Suche feststellen würde. Sie sollte versuchen, nachdem sie ein Gegenbeispiel für eine Prämisse gefunden hat, auch eines für die Konklusio zu finden.

Ein Fehler in einem Beweisversuch wird durch eine Anwendung einer Regel verursacht, die aus eine beweisbaren Aussage eine unbeweisbare macht. Wurde in einem Beweis eine solche Regel angewendet, gibt es für eine der Prämissen dieser Regel ein Gegenbeispiel, aber nicht für deren Konklusio. Könnte die Gegenbeispiel-Suche diese Stelle herausfinden, hätte sie für den Benutzer die Stelle gefunden, an der er einen Fehler gemacht hat, und der Benutzer könnte den Fehler beheben.

Die letzten beiden Punkte können zusammengefaßt werden. Das Ziel ist, die Stelle im Pfad von der Konklusio zu der widerlegten Prämisse zu finden, an der zum ersten Mal ein Gegenbeispiel existiert. Ist dies die Konklusio, so liegt ein falsches Lemma oder eine fehlerhafte Spezifikation vor. Befindet sich die Stelle mitten im Beweisbaum, handelt es sich um einen Fehler in der Beweisführung. Dieses Vorgehen bezeichnen wir als **Rückverfolgung**. Ist man an der Stelle im Beweis angelangt, an der zum ersten Mal ein Gegenbeispiel gefunden werden kann, möchte man den Fehler beseitigen. Dazu wären Hinweise auf die Fehlerursache hilfreich.

Liegt ein Fehler im Beweisversuch vor, hat man schon den Hinweis, daß an der Stelle, an der die Rückverfolgung abbrach, der Fehler begangen wurde. Kann das Gegenbeispiel jedoch bis zur Konklusio zurückberechnet werden, weiß man nur, daß die Aussage nicht beweisbar ist. Im Beispiel 2.1 erhielt man ein Gegenbeispiel, falls $x = 0 \oplus 1 \oplus @$ und $y = 0 \oplus @$ gewählt wurden. Dann ist bei der Konkatenation der Listen das zweite (bzw. letzte) Element der ersten Liste kleiner als das folgende, erste Element der zweiten Liste. Daran sieht man, daß die Voraussetzung $x.first < y.first$ falsch ist, denn sie müßte $x.last < y.first$ heißen. Die konkrete Instanz, mit der ein Gegenbeispiel gefunden werden kann, liefert also Hinweise auf den Fehler. Die Gegenbeispiel-Suche sollte deshalb ein konkretes Gegenbeispiel zurückliefern.

Das konkrete Gegenbeispiel gibt Hinweis auf *einen* Fall, in dem ein Fehler vorliegt. Im obigen Beispiel kann immer dann ein Gegenbeispiel gefunden werden, wenn $x.last \not< y.first$ ist. Allgemeine Gegenbeispiele vereinfachen die Fehlersuche erheblich, aber es ist schwierig, sie zu berechnen. Je allgemeiner aber ein Gegenbeispiel ist, desto mehr Hinweise gibt es für die Fehlersuche.

Ein weiterer Punkt ist die Dauer, die für die Suche eines Gegenbeispiels zur Verfügung steht. KIV ist ein interaktives Beweissystem. Das bedeutet, daß bei den Beweisversuchen regelmäßig der Benutzer mit dem System arbeitet. Läßt sich der Benutzer ein Gegenbeispiel suchen, möchte er nicht zu lange auf ein Ergebnis warten. Welche Dauer toleriert wird, hängt von dem jeweiligen Benutzer ab. Deshalb sprechen wir von einer „angemessenen" Zeit, die die Gegenbeispiel-Suche benötigen darf. Je nach Größe des Problems wird der Benutzer wohl ein bis zwei Minuten Wartezeit in Kauf nehmen.

In diesem Abschnitt haben wir gezeigt, welche Anforderungen an eine Gegenbeispiel-Suche gestellt werden müssen. Wir fassen diese nochmals zusammen. Die Gegenbeispiel-Suche sollte

- aus aktuellem Beweis aufrufbar sein,
- den aktuellen Beweis nicht verändern,
- ein konkretes Gegenbeispiel liefern,
- ein möglichst allgemeines Gegenbeispiel liefern,
- das Gegenbeispiel soweit wie möglich zurückverfolgen,
- und nur eine „angemessene" Zeit benötigen.

Kapitel 3

Realistische Beispiele und erwartete Ergebnisse

In diesem Kapitel werden einige Beispiele vorgestellt, die in Kapitel 4 und Kapitel 7 benutzt werden, um Strategien für die Gegenbeispiel-Suche zu vergleichen und zu testen. Die in den Beispielen enthaltenen Fehler decken die Fehlerklassen Programmfehler, Spezifikationsfehler, falsch formulierte Lemmata und falsche Entscheidung bei der Beweisführung ab. Die Gegenbeispiel-Suche soll in das KIV-System integrierte werden, weshalb versucht wurde, solche Beispiele auszuwählen, die tatsächlich in der praktischen Arbeit mit KIV auftreten können oder schon aufgetreten sind.

Einige der Beispiele stammen aus Fallstudien, die mit KIV bearbeitet wurden. Die enthaltenen Fehler wurden während der Verifikation der Fallstudien entdeckt. Andere Beispiele sind aus dem Praktikum „Programmverifikation", das regelmäßig an der Universität Ulm angeboten wird. Es werden Fehler gezeigt, die im Laufe dieses Praktikum schon öfters aufgetreten sind.

Natürlich sind nicht alle Beispiele so real, wie die gerade beschriebenen, denn Fehler in größeren Systemen können leicht 20-30 Seiten Spezifikation benötigen, um den Fehler und die Fehlerstelle verständlich zu machen. Solche Fehler können an dieser Stelle nicht als Beispiele dienen. Deshalb verwenden wir für die Vergleiche auch einige selbst erstellte Beispiele, die dazu dienen sollen, die Fähigkeiten der Gegenbeispiel-Suche zu testen.

Für die Beispiele wird gezeigt, welche Ergebnisse von der Gegenbeispiel-Suche erwartet werden. Dazu werden die Fehlerursachen aufgezeigt und die Erwartung an die Rückverfolgung beschrieben. Die ausführlichen Spezifikationen befinden sich im Anhang A. Die Spezifikationen im Anhang sind vom KIV-System generiert worden. Deshalb können sich die Bezeichner leicht von den unten angegebenen unterscheiden, denn die hier verwendeten Bezeichner sind zum besseren Verständnis etwas abgeändert worden. Zunächst werden Beispiele vorgestellt, die in der praktischen Arbeit mit dem KIV-System aufgetaucht sind.

3.1 Binärsuche auf Arrays

Bei diesem Beispiel handelt es sich um einen Praktikumsversuch, der die die Hoare'sche Beweisführung verdeutlichen soll [SS98]. Um mit den Hoare-Regeln einen Beweis über eine *while*-Schleife führen zu können, muß der Benutzer eine Invariante angeben, die vor, während und nach der Schleife gültig sein muß. Meist ist es schwierig, die Invariante auf Anhieb richtig zu wählen.

Wir betrachten einen Versuch, bei dem im Praktikum immer wieder falsche Invarianten gewählt werden. Es soll die Korrektheit des *while*-Programms **bin_search#** (siehe Abbildung 3.1) bewiesen werden, welches die binäre Suche implementiert. Damit das Programm das gesuchte Element *searched* finden kann, muß das Array a im Bereich zwischen min und max sortiert und das gesuchte Element *searched* im sortierten Bereich des Arrays enthalten sein. Bei der Binärsuche wird das mittlere Element des Arrays mit dem gesuchten Element verglichen. Ist das gesucht Element größer als das mittlere Element, wird in der oberen Hälfte des Arrays weitergesucht, ansonsten in der

$lower := min$;
$upper := max$;
while $lower < upper$ **do**
begin
    $middle := (lower + upper) \mid 2$;
    if $get(a, middle) \ll searched$
    then $lower := middle + 1$
    else $upper := middle$
end

Abbildung 3.1: Programm: *bin_search#*

unteren Hälfte. Dieses Vorgehen wird solange wiederholt, bis der Suchbereich nur noch ein Element enthält. Das Programm ist korrekt und kann mit der Invariante

$$\underline{min \leq lower_0} \wedge lower_0 \leq n \wedge n \leq upper_0 \wedge \underline{upper_0 \leq max} \wedge <_{sor}(a,\ min,\ max) \wedge \underbrace{get(a,\ n) = searched}$$

bewiesen werden. Oft ist es schwierig, die korrekte Invariante für ein *while*-Programme auf Anhieb zu finden. Werden in der angegebenen Invariante die unterstrichenen Bedingungen vergessen, gelangt man nach einigen Beweisschritten zu der Beweisverpflichtung

Beispiel 3.1 (hoare1)

$$\begin{array}{l} lower_0 < min,\ lower_0 < upper_0,\ <_{sor}(a,\ min,\ max) \\ get(a,\ (lower_0 + upper_0) \mid 2) \ll get(a,\ n),\ n < min \\ \vdash lower_0 = upper_0,\ (lower_0 + upper_0) \mid 2 < n,\ upper_0 < n,\ n < lower_0 \end{array}$$

Diese Aussage behandelt den Fall der binären Suche, in dem das Element, welches zum Aufteilen des Suchbereichs dient, kleiner als das gesuchte Element ist. Man muß also in der oberen Hälfte des Arrays weitersuchen. Die Grenzen $lower_0$ und $upper_0$ schränken den Suchbereich ein, der für die weitere Suche noch zu betrachten ist, während *min* und *max* denjenigen Bereich abgrenzen, in welchem das Array a sortiert ist. Die Funktion *get* liefert das Element der entsprechenden Stelle im Arrays und die Variablen n speichert den Index des gesuchten Elements. Die Bedeutung der Sequenz sollte sein, wenn das Array a im Bereich $min \ldots max$ sortiert und das gesuchte Element größer als das mittlere Element des Arrays ist, dann ist der Index des gesuchten Elements größer, als der Index des Elements, welches zum Aufteilen des Suchbereichs dient.

Da jedoch die Invariante falsch gewählt wurde, hat die Aussage nicht die gewünschte Bedeutung, und man kann ein Gegenbeispiel finden, in welchem der Index des gesuchten Elements kleiner als der Index des aufteilenden Elements ist. Eine solche Belegung der Variablen ist:

Belegung 3.1

$$lower_0 = 0,\ n = 0,\ min = 1,\ upper_0 = 2,\ max = 1,\ a = [ele_0,\ ele],\ ele \ll ele_0$$

Die Arrays sind über einer Parametersorte mit den Variablen *ele* bzw. ele_0 spezifiziert. Da Parametersorten nicht erzeugt sind, kann man die Variablen nicht durch Konstruktorterme ersetzten, mit deren Hilfe man entscheiden kann, ob ein Element kleiner als ein anderes ist. Deshalb muß man explizit $ele \ll ele_0$ fordern, um ein Gegenbeispiel zu erhalten. Die Gegenbeispiel-Suche sollte die Belegung finden. Da die obige Aussage nicht direkt nach der Angabe der Invariante auftrat, sollte die Rückverfolgung das gefundene Gegenbeispiel bis zu der Stelle zurückpropagieren können, an der die falsche Invariante eingegeben wurde.

Ein weiteres Beispiel für das gleiche Programm, jedoch mit einer anderen, fehlerhaften Invariante, führt zu der folgenden Aussage, die auch nicht bewiesen werden kann.

```
split#(x; var y1, y2)
begin
  if x = @ then begin y1 := @; y2 := @ end else
    if x.rest = @ then begin y1 := @; y2 := x end else
      begin
        split#(x.rest.rest ; y1, y2);
        y1 := x.first ⊕ y1;
        y2 := x.first ⊕ y2      /* Fehler, eigentlich: y2 := x.rest.first ⊕ y2 */
      end
end
```

Abbildung 3.2: Programm: *split#*

Beispiel 3.2 (hoare2)

$$\begin{array}{l} \quad min < max,\ min < upper_0,\ lower_0 < max,\ lower_0 < upper_0,\ <_{sor}(a,\ min,\ max) \\ \quad get(a,\ (lower_0 + upper_0) \mid 2) \ll searched \\ \vdash lower_0 = upper_0,\ (lower_0 + upper_0) \mid 2 < min,\ (lower_0 + upper_0) \mid 2 < n,\ max < n, \\ \quad n < min,\ max < upper_0,\ upper_0 < n,\ n < lower_0,\ lower_0 < min \end{array}$$

Hier wurde die durch eine Wellenlinie unterstrichene Bedingung $get(a, n) = searched$ in der Invariante vergessen. Dadurch geht die Verbindung zwischen dem Index n, für das gesuchte Element, und dem gesuchten Element *searched* verloren. Deshalb existiert für diese Formel ein Gegenbeispiel.

Belegung 3.2

$$\begin{array}{l} min = 0,\ max = 1,\ lower_0 = 0,\ upper_0 = 1,\ a = [ele,\ ele_0],\ ele \ll ele_0,\ n = 0, \\ searched = ele_0 \end{array}$$

Diese Instantiierung sollte von der Gegenbeispiel-Suche gefunden und, wie oben, bis zur Eingabe der Invariante zurückverfolgt werden können. In beiden Beispielen wurde ein Fehler in der Beweisführung begangen. Deshalb kann das Gegenbeispiel nicht bis zur Konklusio zurückverfolgt werden, sondern nur bis zu der Stelle, an der die falsche Invariante eingegeben wurde. Die ausführliche Spezifikation befindet sich im Anhang A.1.

3.2 Mergesort auf Listen

Dieses Beispiel stammt aus einer Reihe von Aufgaben für ein Beweiserhappening, bei dem verschiedene Beweissysteme miteinander verglichen werden sollten. Das Ziel des gewählten Beispiels ist, die Implementierung von Mergesort auf Listen zu verifizieren. Bei der Implementierung der **split#**-Routine, die eine Liste in zwei möglichst gleichgroße Teillisten zerlegen soll, ist jedoch ein Fehler unterlaufen (siehe Abbildung 3.2). Anstatt die Liste in eine Teilliste der geraden Positionen und in eine Teilliste der ungeraden Positionen zu zerlegen, wird aus der Ausgangsliste zweimal die Liste mit den ungeraden Positionen selektiert.

Bei dem Versuch der Verifikation von Mergesort mit KIV wurde dieser Fehler entdeckt. Er trat bei einem Hilfslemma zutage, das die Aussage „die Konkatenation der Listen, die durch *split#* entstehen, ist eine Permutation der Ausgangsliste" formuliert. Beim Versuch dieses Lemma zu beweisen, kam man schlußendlich auf drei Aussagen, die nicht beweisen werden konnten. Es werden hier alle drei Aussagen genannt, da die Gegenbeispiele für die Aussagen mit steigendem Aufwand verbunden sind und man an ihnen gut die Fähigkeiten der Gegenbeispiel-Suche testen kann. Das Prädikat *is_perm* gilt für zwei Listen, falls die Listen Permutationen voneinander sind, und die Funktion *del_once* löscht aus einer Liste das angegebene Element einmal.

Beispiel 3.3 (merge1)

$$ele_0 \neq ele,\ \neg\ is\_perm(ele_0 \oplus @,\ ele \oplus @),$$
$$\neg\ is\_perm(ele \oplus ele_0 \oplus @,\ ele \oplus ele \oplus @) \vdash$$

Eigentlich muß man hier gar kein Gegenbeispiel mehr suchen, denn sobald $ele_0 \neq ele$ gilt, ist eine Lösung gefunden. Man hat hier ein typisches Problem parametrisierter Spezifikationen. Da über die Parametersorte keine Axiome angegeben wurden, gibt es Modelle, in denen die Bedingung $ele_0 \neq ele$ erfüllt werden kann, jedoch in anderen Modellen (z.B. einem einelementigen Modell) kann die Bedingung nicht erfüllt werden. Für ein Gegenbeispiel genügt jedoch *ein* Modell, in dem das Gegenbeispiel gilt. Um ein Modell zu erhalten, in dem ein Gegenbeispiel vorliegt, fordern wir explizit

Belegung 3.3

$$ele_0 \neq ele$$

Das zweite Beispiel ist etwas komplizierter aufgebaut.

Beispiel 3.4 (merge2)

$$ele_0 \neq ele,\ \neg\ is\_perm(ele_0 \oplus ele_1 \oplus @,\ ele \oplus ele_1 \oplus @),$$
$$\neg\ is\_perm(ele \oplus ele_0 \oplus ele_1 \oplus @,\ ele \oplus ele \oplus ele_1 \oplus @) \vdash$$

Man hat jedoch das gleiche Gegenbeispiel.

Belegung 3.4

$$ele_0 \neq ele$$

Für des Element ele_1 ist es egal, welchen Wert es annimmt, deshalb gibt es auch im Gegenbeispiel keine Aussage darüber. Die dritte Aussage hat, im Gegensatz zu den beiden von oben, nicht nur freie Variablen für Elemente, sondern auch noch über Listen.

Beispiel 3.5 (merge3)

$$ele_0 \neq ele,\ ele_2 \in x_4 \oplus ele_1 \oplus x_8,$$
$$is\_perm(x_6,\ del\_once(ele_2,\ x_4 \oplus ele_1 \oplus x_8)),$$
$$\neg\ is\_perm(ele_0 \oplus ele_1 \oplus ele_2 \oplus x_6,\ ele_1 \oplus x_4 \oplus ele \oplus ele_1 \oplus x_8),$$
$$\neg\ is\_perm(ele \oplus ele_0 \oplus ele_1 \oplus ele_2 \oplus x_6,\ ele \oplus ele_1 \oplus x_4 \oplus ele \oplus ele_1 \oplus x_8) \vdash$$

Das Prädikat $ele \in x$ trifft dann zu, wenn das Element ele in der Liste x vorkommt. Ein Gegenbeispiel lautet:

Belegung 3.5

$$x_0 = @,\ x_3 = @,\ x_4 = @,\ x_6 = @,\ x_8 = @,\ ele_1 = ele_2,\ ele_0 \neq ele$$

Dies ist zwar immer noch ein einfaches Beispiel, da alle Listen mit dem Konstruktor @, also der leeren Liste, instantiiert werden müssen, jedoch treten schon eine ganze Menge freie Variablen auf, für die eine Belegung gefunden werden muß.

Die Rückverfolgung sollte in allen drei Fällen das Gegenbeispiel bis zur Konklusio zurückberechnen können, da ein Programmfehler in der *split#*-Routine vorliegt. Die ausführliche Spezifikation ist im Anhang A.2 abgebildet.

3.3 Reguläre Ausdrücke

Nun folgen zwei Beispiele über reguläre Ausdrücke, die eine reguläre Sprache, die Wörter enthält, definieren soll. Ein Wort s (String) besteht aus keinem (leeres Wort), einem oder mehreren Buchstaben c (Character). Der reguläre Ausdruck

- ε beschreibt die Sprache, die das leere Wort enthält
- c beschreibt die Sprache, die den String, der aus dem Buchstaben c besteht, enthält
- $s_1 s_2$ beschreibt das Kreuzprodukt der Sprache, die s_1 enthält, mit der Sprache, die s_2 enthält
- $s_1 \mid s_2$ beschreibt die Vereinigung der Sprache, die s_1 enthält, mit der Sprache, die s_2 enthält
- s^* beschreibt die Sprache, die das leere Wort oder beliebig viele Strings s aneinandergehängt enthält

Bei beiden Beispielen liegt eine falsche Spezifikation des Prädikates *matches* vor, welches erkennen soll, ob ein String in einer Sprache, die durch einen regulären Ausdruck beschrieben wird, enthalten ist. Für die Anwendung des *-Operators ist *matches* falsch spezifiziert worden. Dieser Spezifikationsfehler wurde bei einem Beweisversuch mit KIV entdeckt, als man Aussagen beweisen wollte, die für reguläre Ausdrücken gelten. Die eine Aussage war, daß „der String s genau dann in der Sprache $@_r{}^*$, d.h. den *-Operator angewandt auf den leere regulären Ausdruck, enthalten ist, wenn s das leere Wort ist". Diese Aussage ist richtig, denn der *-Operator erkennt immer das leere Wort. Beim Versuch diese Aussage mit der fehlerhaften Spezifikation zu beweisen, stieß man auf die Aussage

Beispiel 3.6 (match1)

$$\vdash @_r{}^* \; matches \; @_s$$

Der leere reguläre Ausdruck $@_r$ beschreibt die Sprache, die keine Wörter enthält. Das leere Wort $@_s$ ist in der Sprache enthalten, die der regulären Ausdruck ε beschrieben. In dem Beispiel sind keine freien Variablen mehr in der Sequenz vorhanden. Die Gegenbeispiel-Suche müßte zeigen, daß diese Aussage in der fehlerhaften Spezifikation nicht gilt und das Gegenbeispiel bis zum Anfang des Beweises zurückverfolgen können.

Beim zweiten Beispiel liegt ein ähnlicher Sachverhalt vor. Die zu beweisende Aussage lautet „der *-Operator auf eine Sprache, die nur den Buchstaben c enthält, angewandt, enthält dann den String s, wenn der String s nur aus Buchstaben c besteht". Auch hier gibt es keine freien Variablen zu instantiieren. Ein Beweisversuch führte zu folgender Sequenz:

Beispiel 3.7 (match2)

$$\vdash mkreg(c)^* \; matches \; @_s$$

Die Funktion *mkreg* macht aus einem Symbol einen regulären Ausdruck, der genau dieses Symbol beschreibt. Die Gegenbeispiel-Suche müßte wiederum die Aussage treffen, daß die Behauptung in der falschen Spezifikation nicht gilt, und die Rückverfolgung müßte diese Behauptung bis zur Konklusio zurückverfolgen können. Die vollständige Spezifikation befindet sich im Anhang A.3.

3.4 Wege in Graphen

Die beiden folgenden Beispiele wurden für Experimente mit der Gegenbeispiel-Suche konstruiert. Ein Graph entsteht aus einem leeren Graphen, in den Knoten und Kanten eingefügt werden. Wenn in einen Graphen eine Kante, deren Knoten noch nicht vorhanden sind, eingefügt wird, werden

mit der Kante auch die Knoten eingefügt. Über dieser Spezifikation sind die Prädikate *pathp* und *spathp* definiert. Sie haben beide zwei Argumente, eine Liste von Knoten und einen Graphen. Das Prädikat *pathp* ist wahr, wenn die Knoten in der Liste einen Pfad im Graphen bilden. Das Prädikat *spathp* soll einen Pfad beschreiben, in dem keine Knoten doppelt vorkommen dürfen. Die Funktionen $+_e$ und $+_v$ fügen Kanten bzw. Knoten in einen Graphen ein und die Prädikate $\in_e$ und $\in_v$ testen, ob die angegebenen Kanten bzw. Knoten im Graphen vorhanden sind. Ein leerer Graph wird durch $@g$ beschrieben und $v_1 \mapsto v_2$ stellt eine Kante vom Knoten v_1 zum Knoten v_2 dar. Es wurden nun zwei Behauptungen aufgestellt:

Beispiel 3.8 (graph1)

$$\vdash pathp(x,\ g) \leftrightarrow spath(x,\ g)$$

und

Beispiel 3.9 (graph2)

$$\vdash (\forall\ v.\ v \in_v g \rightarrow v \mapsto v \notin_e g) \rightarrow pathp(x,\ g) \leftrightarrow spath(x,\ g)$$

Das erste Beispiel besagt, daß jeder Pfad in einem Graphen ein Pfad ist, in dem keine Knoten doppelt vorkommen. Im zweiten Beispiel werden nur Graphen betrachtet, die keine Schlingen besitzen. Beide Aussagen gelten natürlich nicht. Die Belegung für das Gegenbeispiel für die erste Behauptung lautet

Belegung 3.6

$$x = (v,\ v),\ g = @g +_e\ e,\ mit\ e = v \mapsto v$$

Da wir uns schon am Anfang der Sequenz befinden, muß die Rückverfolgung das Gegenbeispiel nicht weiter anpassen. Die Belegung für das zweite Gegenbeispiel ist

Belegung 3.7

$$x = (v_1,\ v_2,\ v_1),\ g = (@g +_e e_1) +_e e_2,\ mit\ e_1 = v_1 \mapsto v_2\ und\ e_2 = v_2 \mapsto v_1$$

Diese beiden fehlerhaften Lemmata wurden als Beispiele ausgewählt, da für die Belegung der Variablen für den Graphen eine komplizierte Termstruktur gefunden werden muß. Dies ist sicher eines der schwierigsten Beispiele, die für die Experimente betrachtet werden. Die Graphen sind im Anhang A.4 spezifiziert.

3.5 Binär-Arithmetik

In diesem Beispiel werden natürlichen Zahlen durch Binärwörter dargestellt und die Operationen auf den natürlichen Zahlen durch Prozeduren über den Binärwörtern implementiert. Dabei ist in der Prozedur **succ#**, die den Nachfolger einer natürlichen Zahl berechnen soll, ein Fehler enthalten (siehe Abbildung 3.3). In diesem Beispiel sind b_1 und b_2 Variablen für binäre Zahlen, die Funktion *pop* schneidet von einer Binärzahl das letzte Bit ab und die Funktion *top* liefert das letzte Bit einer Binärzahl. Die Konstruktoren _0 und _1 repräsentieren das Bit 0 bzw. 1 und die Konstruktoren .0 und .1 hängen an eine Binärzahl das Bit 0 bzw. 1 an. Für die geraden Zahlen werden die Nachfolger nicht korrekt berechnet. Dieser Fehler wird erkannt, wenn man die Beweisverpflichtung „der Vorgänger des Nachfolgers einer Zahl n, ist wieder n" beweisen möchte. Die Beweisverpflichtung sieht im Sequenzenkalkül folgendermaßen aus:

succ#(b_1; **var** b_2)
begin
  if $b_1 = \_0$ **then** $b_2 := \_1$ **else**
    if $b_1 = \_1$ **then** $b_2 := \_1.0$ **else** /* Fehler: **if** b_1 top = _0 ... */
      begin succ#(b_1 *pop*; b_2); $b_2 := b_2.0$ **end**
end

Abbildung 3.3: Programm: *succ#*

Beispiel 3.10 (binär1)

$$\langle r\#(b)\rangle\ true \vdash \langle succ\#(b;\ b_0)\rangle\ \langle pred\#(b_0;\ b_1)\rangle\ b_1 = b$$

Das Programm *succ#* berechnet den Nachfolger (fehlerhaft) und das Programm *pred#* den Vorgänger einer binären Zahl. Die Programme müssen nur für Binärzahlen ohne führende Nullen korrekt arbeiten. Deshalb wird für die Binärzahl b vorausgesetzt, daß sie keine führende Nullen besitzt. Diese Voraussetzung wird durch das Programm $r\#$ gewährleistet. Wenn man das Programm analysiert, sieht man, daß für alle geraden Zahlen der Nachfolger falsch berechnet wird, also stellt insbesondere

Belegung 3.8

$$b = \_1.0$$

ein Gegenbeispiel dar. Eine Rückverfolgung ist nicht mehr notwendig, da die Gegenbeispiel-Suche am Beweisanfang ansetzt. Versucht man zuerst, das Programm zu beweisen, erhält man einen nicht beweisbaren Ast im Beweisbaum, dessen Prämisse folgendermaßen aussieht:

Beispiel 3.11 (binär2)

$$\langle pred\#(b_4;\ b_1)\rangle b_1 = b\ pop,\ \langle succ\#(b\ pop;\ b_0)\rangle b_0 = b_4,\ \langle rh\#(b\ pop)\rangle true,\ b\ top = \_0,$$
$$b_4 \neq \_1,\ b \neq \_0,\ b \neq \_1 \vdash$$

Das Programm *rh#* ist eine Hilfsprozedur für das Programm $r\#$. Das Gegenbeispiel für diese Prämisse lautet ebenfalls

Belegung 3.9

$$b = \_1.0$$

Der Wert für die Variable b_4 wird durch die Programmausführung von *succ#* bestimmt. In diesem Fall sollte die Gegenbeispiel-Suche nicht nur die Belegung finden, sondern die Rückverfolgung sollte, da ein Programmfehler vorliegt, das Gegenbeispiel bis zur Konklusio zurückberechnen können. Die Spezifikation der Binärzahlen befindet sich im Anhang A.5.

3.6 Intervall-Listen

In diesem Beispiel wird eine Implementierung von Intervall-Listen angegeben. Mit Hilfe von Intervall-Listen können Mengen von natürlichen Zahlen dargestellt werden. Sind in einer Menge aufeinanderfolgende Zahlen enthalten, kann eine solche Zahlenreihe durch den Anfangswert und

```
insert#(x, n; var y0)
begin
  if x = @ then y0 := [n, n] ⊕ x else
    var m1 = (x.first).1, m2 = (x.first).2 in
    if n < m1 then
      if n = m1 −1 then y0 := [m1 −1 , m2] ⊕ x.rest else y0 := [n, n] ⊕ x
    else
      if ¬ m2 < n then y0 := x else
        if n = m2 +1 then
          if x .rest = @ then
            y0 := [1, m2 +1 ] ⊕ x.rest
          else
            /* Fehler: falls n +1 = ((x.rest).first).1 müssen die Intervalle zusammenfallen */
            x0 := [m1, m2 +1 ] ⊕ x.rest
        else
          begin insert#(x.rest, n; y0); y0 := [m1, m2] ⊕ y0 end
end
```

Abbildung 3.4: Programm: *insert#*

den Endwert beschrieben werden. Z.B. kann die Menge {1, 2, 3, 5, 6, 9} durch eine Intervall-Liste ([1, 3], [5, 6], [9, 9]) dargestellt werden. In der Prozedur *insert#*, die ein Element in eine Intervall-Liste einfügt, ist ein Fehler enthalten (siehe Abbildung 3.4). Wird in der obigen Liste die Zahl 4 einsortiert, müssen die Intervalle [1, 3] und [5, 6] zu einem einzigen Intervall [1, 6] zusammenfallen. Dieser Fall wurde bei der Implementierung vergessen. Wie das Beispiel 3.2 stammt auch diese Implementierung aus einer Aufgabenstellung zu einem Beweiserhappening und ist kein „erfundenes" Beispiel. Die zu beweisende Sequenz lautet

Beispiel 3.12 (iv-list1)

$$is\_iv\text{-}list(x),\ \langle insert\#(x,\ n;\ y)\rangle y = y_1 \vdash is\_iv\text{-}list(y_1)$$

Das Prädikat *is_iv-list* trifft für eine Liste zu, falls sie eine korrekte Intervall-Liste darstellt. Das Beispiel soll die Aussage formulieren, daß man eine korrekte Intervall-Liste y_1 erhält, wenn in eine korrekte Intervall-Liste x ein Element n durch *insert#* eingefügt wird. Das Programm arbeitet aber falsch und deshalb ist die Ergebnisliste y_1 nicht immer eine korrekte Intervall-Liste. Ein einfaches Gegenbeispiel ist:

Belegung 3.10

$$x = ([0,\ 0],\ [2,\ 2]),\ n\ = 1$$

Die Gegenbeispiel-Suche sollte dieses Gegenbeispiel finden. Eine Rückverfolgung ist nicht mehr notwendig. Auch bei diesem Beispiel kann zuerst ein Beweisversuch gestartet werden, und das Lemma kann, bis auf einen Ast, bewiesen werden. An diesem Ast entsteht die Beweisverpflichtung

Beispiel 3.13 (iv-list2)

$$x \neq @,\ is\_iv\text{-}list([n_0,\ n_1] \oplus x),\ \neg\ is\_iv\text{-}list([n_0,\ n_1 +1\ ] \oplus x),\ \neg\ n_1 +1\ < n_0 \vdash$$

Für diese Sequenz ist

Belegung 3.11

$$x = ([2,\ 2]),\ n_0 = 0,\ n_1 = 0$$

ein Gegenbeispiel, welches die Gegenbeispiel-Suche finden sollte. Da der Fehler in der Routine *insert#* liegt, sollte in diesem Fall die Rückverfolgung das Gegenbeispiel bis zur Konklusio des Beweisversuchs zurückberechnen können. Die ausführliche Spezifikation für die Intervall-Listen findet man im Anhang A.6.

3.7 Compiler für boolesche Ausdrücke

Das letzte Beispiel ist ein Compiler für boolesch Ausdrücke, der die Operatoren Negation ($\neg$) und Implikation ($\rightarrow$) der Quellsprache in eine Zielsprache übersetzt und die Argumente der Operatoren auf einem Stack ablegt. Bei der Abarbeitung des Programms der Zielsprache soll, mit Hilfe des Stacks, der Ausdruck ausgewertet werden. Der Fehler besteht darin, daß der spezifizierte Compiler die beiden Argumente der Implikation vertauscht auf den Stack legt, und deshalb die Auswertung des übersetzten Ausdrucks ein falsches Ergebnis liefert. Die Beweisverpflichtung lautet, daß die Abarbeitung des übersetzte Ausdrucks denselben Wert liefern soll, wie die Auswertung des Originalausdrucks. Für Testzwecke geben wir auch hier wieder zwei Beispiele an. Zum einen wird für die Beweisverpflichtung ein Gegenbeispiel gesucht und im zweiten Beispiel für eine Sequenz, die während eines Beweisversuchs entstand.

Beispiel 3.14 (compiler1)

$$\vdash run(compile(t) \odot x,\ s) = run(x,\ push(val(t),\ s))$$

Diese Sequenz drückt die Beweisverpflichtung aus. Wenn der durch *compile* übersetzte Ausdruck t durch *run* ausgeführt wird, soll auf dem Stack derselbe Wert liegen, wie wenn der ausgewertete Ausdruck $val(t)$ direkt auf den Stack gelegt wird. Durch den Fehler im Compiler erhält man das Gegenbeispiel

Belegung 3.12

$$t = 0\ \rightarrow 1,\ x = @\ und\ s = @_s$$

wobei $@_s$ der leere Stack ist. Dieses Gegenbeispiel entsteht, da $0 \rightarrow 1 =$ **true** aber $1 \rightarrow 0 =$ **false** ist, und der Compiler die Argumente vertauscht auf den Stack legt. Versucht man die Beweisverpflichtung zu beweisen, gelangt man an die folgende Prämisse

Beispiel 3.15 (compiler2)

$$val(t_1) = 0 \vdash run(x,\ push(1,\ s)) = run(x,\ push(val(t_0 \rightarrow t_1),\ s))$$

und erhält mit

Belegung 3.13

$$t_1 = 0,\ t_0 = 1,\ x = @\ und\ s = @_s,$$

ein Gegenbeispiel. Die Gegenbeispiel-Suche sollte in beiden Fällen die entsprechenden Belegungen finden und für das zweite Beispiel sollte die Rückverfolgung bis zur Konklusio des Beweises gelangen. Die komplette, fehlerhafte Spezifikation für das Beispiel ist im Anhang A.7 abgedruckt.

Kapitel 4

Eine Lösung mit automatischen Theorembeweisern

In diesem Kapitel besprechen wir einen Lösungsansatz für die Gegenbeispiel-Suche mit automatischen Theorembeweisern. Automatische Theorembeweiser lösen Probleme ohne Benutzerinteraktion. Damit erfüllen sie eine der wichtigen Voraussetzungen, die an den Mechanismus der Gegenbeispiel-Suche gestellt wird. Eine weitere wesentliche Voraussetzung ist, daß die Methode nicht nur bestätigt, daß ein Gegenbeispiel existiert, sondern dem Benutzer konkrete bzw. allgemeine Gegenbeispiele liefert. Diese Anforderung kann von automatischen Theorembeweisern erfüllt werden, die für eine $\exists$-abgeschlossene Formel φ, die Belegungen der Variablen, die durch den Quantor gebunden sind, berechnen können, für die φ gültig ist. Dazu benutzen die automatischen Theorembeweiser ein spezielles Prädikat *answer*, mit dem der Benutzer angeben kann, für welche Variablen er die Belegungen ausgegeben bekommen möchte. Deshalb werden diese automatischen Theorembeweiser auch automatische Theorembeweiser mit Antwortprädikat genannt.

Um konkrete Gegenbeispiele mit automatische Theorembeweiser, die kein Antwortprädikat besitzen, zu berechnen, gibt es die Möglichkeit, konkrete Werte für die Lösung vorzugeben. Der automatische Theorembeweiser muß dann (schnell) testen, ob die Vorgabe tatsächlich eine Lösung ist. Wenn ja, wurde ein konkretes Gegenbeispiel gefunden, ansonsten müssen andere Werte getestet werden.

Die Anforderungen, daß die Gegenbeispiel-Suche von einem aktuellen Beweis aus aufgerufen werden muß und daß sie den Originalbeweis nicht verändert, sind Anforderungen an die Implementierung und haben nichts mit der verwendeten Methode zu tun. Die Rückverfolgung ist auch unabhängig von der Methode, mit der die Gegenbeispiele gesucht werden. Damit können automatischen Theorembeweiser alle Anforderungen erfüllen, die die Gegenbeispiel-Suche stellt.

In diesem Kapitel untersuchen wir, ob automatischen Theorembeweiser in der Lage sind, Gegenbeispiel in einer angemessenen Zeit zu finden. Wir sehen uns im ersten Abschnitt an, wie die Anfrage an die automatischen Theorembeweiser gestellt werden muß, damit das Ergebnis ein Gegenbeispiel für eine Prämisse in einem Beweisbaum ist. Danach stellen wir kurz die getesteten automatischen Theorembeweiser vor und zeigen im darauffolgenden Abschnitt, mit welchen Beispielen sie getestet und welche Ergebnisse erreicht wurden. Im letzten Abschnitt werten wir diese Ergebnisse aus.

4.1 Die Anfrage an die automatischen Theorembeweiser

Wenn automatische Theorembeweiser Gegenbeispiele suchen sollen, müssen sie für die Negation einer Aussage einen Beweis finden. Da die freien Variablen einer Formel $\forall$-quantifiziert sind, gibt es für die Formel φ ein Gegenbeispiel, wenn es eine Belegung der freien Variablen gibt, so daß die negierte Aussage von φ gilt. Sind $\underline{x}$ die freien Variablen von φ, muß ein automatischer Theorembeweiser für $\varphi' = \exists\ \underline{x}.\ \neg\ \varphi$ einen Beweis finden. Für eine Sequenz $\Gamma \vdash \Delta$ erhält man ein

Gegenbeispiel, wenn die Formel $\exists\ \underline{x}.\ \neg\ (\Gamma \rightarrow \Delta)$ bzw. die Sequenz $\vdash \exists\ \underline{x}.\ \neg\ (\Gamma \rightarrow \Delta)$ bewiesen wird. Dabei ist $\underline{x} = \mathit{free}(\Gamma) \cup \mathit{free}(\Delta)$.

Besitzt der automatische Theorembeweiser ein Antwortprädikat *answer*, kann man ihm die Anfrage $\varphi' \wedge \mathit{answer}(\underline{x})$ stellen. Der automatische Theorembeweiser versucht dann, die Formel φ' zu beweisen und bindet im letzten Schritt die Belegungen, mit denen er den Beweis gefunden hat, an die Variablen $\underline{x}$ des Antwortprädikates. Diese Variablenbindung gibt der automatische Theorembeweiser als Resultat an den Benutzer. Wählt man $\underline{x} = \mathit{free}(\varphi)$, so erhält man die konkreten Belegungen für das Gegenbeispiel. Existiert eine Lösung für die Anfrage $\exists\ \underline{x}.\ \neg\ \varphi$, kann ein automatischer Theorembeweiser ohne Antwortprädikat dies nur bestätigen. Damit können wir dem Benutzer kein konkretes Gegenbeispiel liefern, um ihn bei der Fehlersuche zu unterstützen. Um trotzdem konkrete Gegenbeispiele zu erhalten, können wir dem automatischen Theorembeweiser eine potentielle Lösung vorgeben. Er überprüft dann, ob diese Lösung tatsächlich eine ist. Dadurch erhalten wir eine *generate & test*-Strategie, bei der die automatischen Theorembeweiser den Test durchführen. Sei der Vektor $\underline{t}$ eine Belegung für die freien Variablen $\underline{x}$ einer Formel φ, muß an den automatischen Theorembeweiser die Anfrage $\varphi' = \neg\ \varphi^{\underline{t}}_{\underline{x}}$ gestellt werden. Da keine freien Variablen mehr in φ' vorkommen, muß die Anfrage nicht $\exists$-abgeschlossen werden. Bestätigt der automatische Theorembeweiser die Gültigkeit der Aussage, ist $\underline{t}$ ein konkretes Gegenbeispiel für φ. Damit ein existierendes Gegenbeispiel auch gefunden wird, müssen für $\underline{t}$ nach und nach alle potentiellen Lösungen eingesetzt werden, bis eine Lösung gefunden wird.

Im folgenden wollen wir testen, ob automatische Theorembeweiser in der Lage sind, ein Gegenbeispiel für eine Formel φ zu finden. Dazu testen wir einige der Beispiele aus Abschnitt 3. Die automatischen Theorembeweiser erhalten dazu sowohl die Anfrage $\exists\ \underline{x}.\ \neg\ \varphi$ als auch $\neg\ \varphi^{\underline{t}}_{\underline{x}}$ gestellt. Der Vektor $\underline{t}$ repräsentiert dabei eine korrekte Belegung für ein Gegenbeispiel.

Neben der Formel, die das Gegenbeispiel darstellt, müssen den automatischen Theorembeweisern noch die Axiome der Theorie übergeben werden, in der das Gegenbeispiel gesucht werden soll. Was dabei zu beachten ist, sehen wir im folgenden Abschnitt.

4.1.1 Die Theorie für den Gegenbeispielbeweis

Wird, während eines Beweisversuchs für ein Lemma, die Gegenbeispiel-Suche mit der Formel φ aufgerufen, ist die Theorie für φ dieselbe, wie die Theorie für das Lemma, denn das Gegenbeispiel soll gerade in dieser Theorie gelten. Der Beweisversuch findet jedoch in KIV statt, deshalb muß den automatischen Theorembeweisern die Theorie noch bekannt gemacht werden. Dazu werden ihnen die Axiome der Spezifikation übergeben.

In KIV ist es möglich, strukturierte Spezifikationen zu erstellen. Die Theorie einer Spezifikation *Sp* besteht deshalb nicht nur aus den Axiomen der Spezifikation *Sp*, sondern zusätzlich aus allen Axiomen der Spezifikationen, auf die *Sp* aufbaut. Deshalb müssen den automatischen Theorembeweisern all diese Axiome übergeben werden.

Meist werden für die Beweise jedoch nicht alle Axiome der Theorie benötigt. Im einführenden Beispiel 2.1 über Listen auf natürlichen Zahlen, wird ein Gegenbeispiel für die Aussage $\vdash \mathit{ord}(x) \wedge \mathit{ord}(y) \wedge x.\mathit{first} < y.\mathit{first} \rightarrow \mathit{ord}(x \odot y)$ gesucht. Für ein Gegenbeispiel kann $x = 0 \oplus 1 \oplus @$ und $y = 1 \oplus @$ gewählt werden. Wie wir gesehen haben, müßten alle Axiome der Listenspezifikation und der Spezifikation der natürlichen Zahlen an den automatischen Theorembeweiser übergeben werden. Für den Beweis benötigt er aber keine Axiome über die Addition oder über Primzahlen, die in der Spezifikation der natürlichen Zahlen vorkommen. Da der automatische Theorembeweiser nicht weiß, welche Axiome für den Beweis benötigt werden, betrachtet er alle Axiome, die ihm zur Verfügung gestellt werden. Dadurch können unnötige Beweisschritte entstehen, die die Beweissuche verlängern.

Damit den automatischen Theorembeweisern nur die benötigten Axiome zur Verfügung gestellt werden, wurde die Axiomenreduktion [RS98] entwickelt, die Axiome, die für den Beweis nicht notwendig sind, aus der Theorie entfernt. Für die Testbeispiele wurde immer ein Versuch mit der kompletten und ein Versuch mit der reduzieren Axiomenmenge durchgeführt.

4.2 Die getesteten automatischen Theorembeweiser

Für die Testbeispiele wurde der Tableaubeweiser **PROTEIN** [Sch96] und der Resolutionsbeweiser **SPASS** [Wei97] herangezogen. PROTEIN besitzt ein Antwortprädikat, das die Belegung, die zu einer Lösung geführt hat, an den Benutzer als Ergebnis zurück gibt. KIV besitzt einen **Simplifier**, der Formeln vereinfacht darstellen und manchmal sogar beweisen kann. Kann der Simplifier das Problem nicht selbstständig lösen, ist es interessant zu wissen, wieviele Benutzerinteraktionen zur Lösung in KIV notwendig sind, da man vielleicht eine Erweiterung in KIV implementieren könnte, die ohne diese Interaktionen auskommt. Deshalb wird neben den beiden automatischen Theorembeweisern auch der KIV-Simplifier mit den Testbeispielen getestet.

4.3 Die Beispiele

Für diese Tests wurden, aus den in Kapitel 3 vorgestellten Beispielen, die Binärsuche auf Arrays, Mergesort auf Listen und die regulären Ausdrücke ausgewählt. Die Beispiele sind in der oben angegebenen Form umgeschrieben worden, damit die automatischen Theorembeweiser ein Gegenbeispiel konstruieren konnten. Wenn die Beispiele nicht bewiesen werden konnten, wurde teilweise von den vorgestellten Beispielen abgewichen. Deshalb geben wir in den einzelnen Abschnitten nochmals die Sequenzen an, für die tatsächlich ein Gegenbeispiel gesucht worden ist, und präsentieren die Ergebnisse.

Für die automatischen Theorembeweiser geben wir die Zeit an, die sie benötigten, um die Lösung zu finden. Bei den Tests wurde den automatischen Theorembeweisern eine Zeitschranke von zwei Minuten gesetzt. Wenn sie in dieser Zeit die Lösung nicht finden konnten, wurde der Versuch abgebrochen, und es erscheint ein — in der Tabelle. Die Tests wurden mit der reduzierten und der unreduzierten Axiomenmenge durchgeführt. Deshalb stehen pro automatischen Theorembeweiser und Beispiel immer zwei Einträge in der Ergebnistabelle. Der linke Eintrag zeigt die Zeit für die unreduzierte Axiomenmenge, der rechte für die reduzierte.

Wenn der Simplifier ein Problem selbstständig lösen konnte, so geschah dies meist in ein bis zwei Sekunden. Da die exakte Zeitmessung für den Simplifier schwierig ist, werden im Überblick die Anzahl der Interaktionen angegeben, die in KIV notwendig sind, um das Gegenbeispiel zu finden. Wenn die Belegungen, die zu einer Lösung führen, nicht angegeben sind, ist eine Interaktion immer die Instantiierung des $\exists$-Quantors. Damit gibt der Benutzer das konkrete Gegenbeispiel vor. Dies ist aber die wesentliche Aufgabe der Gegenbeispiel-Suche, die nicht vom Benutzer übernommen werden sollte. In diesem Fall ist der Vergleich zwischen den automatischen Theorembeweisern und KIV nicht korrekt, und die Anzahl der Interaktionen werden mit einem * gekennzeichnet. Die Tests wurden auf einer Sun Ultrasparc 1 durchgeführt.

4.3.1 Binärsuche auf Arrays

Für das Beispiel *hoare1* (3.1)

$$\begin{array}{l} lower_0 < min,\ lower_0 < upper_0,\ <_{sor}(a,\ min,\ max) \\ get(a,\ (lower_0 + upper_0) \mid 2) \ll get(a,\ n),\ n < min \\ \vdash lower_0 = upper_0,\ (lower_0 + upper_0) \mid 2 < n,\ upper_0 < n,\ n < lower_0 \end{array}$$

konnte keiner der automatischen Theorembeweiser ein Gegenbeispiel finden. Deshalb wurde versucht, ein Gegenbeispiel für die Aussage „ein einelementiges Array ist nicht sortiert" zu suchen.

Beispiel 4.1 (hoare1′)

$$<_{sor}(put(mkarray(1),\ 0,\ ele),\ 0,\ 0) \vdash$$

Jedoch auch hierfür fand keiner der automatischen Theorembeweiser ein Gegenbeispiel. Deshalb wurde für die Aussage die Definition von $<_{sor}$ eingesetzt. Dadurch entsteht das Beispiel

Beispiel	*hoare1*	*hoare1′*	*hoare1″*	*hoare1‴*
PROTEIN	— / —	— / —	— / —	— / —
SPASS	— / —	— / —	— / —	46.3 / 5
KIV	4*	0	0	0

Abbildung 4.1: Ergebnisse: Binärsuche auf Arrays

Beispiel 4.2 (hoare1″)

$$\begin{array}{l} 0 < size(put(mkarray(1),\ 0,\ ele)) \wedge \\ (\forall\ m_0,\ n_0.\quad 0 \leq m_0 \wedge m_0 < n_0 \wedge n_0 \leq 0 \\ \quad \rightarrow get(put(mkarray(0+1),\ 0,\ ele),\ m_0) \ll get(put(mkarray(0+1),\ 0,\ ele),\ n_0)) \\ \vdash \end{array}$$

Selbst für diese Form wurde kein Gegenbeispiel gefunden. Wird auf diese Sequenz der KIV-Simplifier angewendet, entsteht das Beispiel

Beispiel 4.3 (hoare1‴)

$\vdash put(mkarray(1),\ 0,\ ele) = mkarray(0)$

Dafür konnte der automatische Theorembeweiser SPASS ein Gegenbeispiel konstruieren. Da diese Ergebnisse sehr schlecht ausgefallen sind, wurden für das Beispiel *hoare2* (3.2) keine weiteren Tests durchgeführt.

4.3.2 Mergesort auf Listen

Diese Beispiele führen, durch die Verwendung von Parametersorten in den Listen, zu einem Problem. Für die Sorte **elem** existiert kein Erzeugtheitsprinzip. Deshalb kann mit dieser Spezifikation keine Aussage über die Werte der verschiedenen Variablen dieser Sorte gemacht werden. Für die Beispiele benötigt man aber, daß die Werte der Variablen *ele* und ele_0 verschieden sind. Aufgrund der Axiomatisierung kann nicht festgestellt werden, daß es zwei verschiedene Werte für Elemente der Sorte **elem** gibt, da die Sorte **elem** aber als Platzhalter für andere Sorten dient, kann es verschiedene Werte geben. Um diesen Sachverhalt den automatischen Theorembeweisern mitzuteilen, kann man ein Axiom $\exists\ ele,\ ele_0.\ ele \neq ele_0$ formulieren und dies zusätzlich in die Theorie mit aufnehmen. Damit wird für die Sorte **elem** garantiert, daß es verschiedene Elemente gibt. Eine andere Möglichkeit ist, diesen Sachverhalt gleich in die zu beweisende Formel mit einzubauen. Die Variablen *ele* und ele_0 werden für die Gegenbeispiel-Suche nicht durch den $\exists$-Quantor gebunden, sondern es wird als Voraussetzung die Gleichung $ele \neq ele_0$ in die Sequenz mit aufgenommen. Suchen wir für das Beispiel *merge1* (3.3)

$$ele_0 \neq ele,\ \neg\ isperm(ele_0 \oplus @,\ ele \oplus @),\ \neg\ isperm(ele \oplus ele_0 \oplus @,\ ele \oplus ele \oplus @) \vdash$$

ein Gegenbeispiel, wandeln wir es in die Form

$$ele_0 \neq ele \vdash \neg\ isperm(ele_0 \oplus @,\ ele \oplus @),\ \neg\ isperm(ele \oplus ele_0 \oplus @,\ ele \oplus ele \oplus @)$$

um, und übergeben es den automatischen Theorembeweisern zur Verifikation. Im Sukzedenten kommt kein $\exists$-Quantor mehr vor, da, außer *ele* und ele_0, keine weiteren freien Variablen in der Sequenz vorkommen. Nehmen wir stattdessen das Axiom $\exists\ ele,\ ele_0.\ ele \neq ele_0$ mit in die Theorie auf, muß die Sequenz in

$$\vdash \exists\ ele,\ ele_0.\ \neg\ isperm(ele_0 \oplus @,\ ele \oplus @),\ \neg\ isperm(ele \oplus ele_0 \oplus @,\ ele \oplus ele \oplus @)$$

	Fall 1			Fall 2		
Beispiel	*merge1*	*merge2*	*merge3*	*merge1*	*merge2*	*merge3*
PROTEIN	0.1 / 0.1	— / 4.8	— / —	0.2 / 0.1	— / 11.5	— / —
SPASS	— / 0.8	— / —	— / —	68 / 0.2	— / —	— / —
KIV	2^*	2^*	2^*	0	1^*	1^*

	Fall 3		
Beispiel	*merge1*	*merge2*	*merge3*
PROTEIN	0.2 / 0.1	— / —	— / —
SPASS	67.3 / 0.3	— / 28.2	— / —
KIV	0	0	0

Abbildung 4.2: Ergebnisse: Mergesort

umgewandelt werden. Damit haben wir pro Beispiel zwei Versuche für die automatischen Theorembeweiser. Einen dritten Versuch erhalten wir, wenn wir für das Beispiel *merge2* (3.4)

$$ele_0 \neq ele, \neg\ isperm(ele_0 \oplus ele_1 \oplus @, ele \oplus ele_1 \oplus @),$$
$$\neg\ isperm(ele \oplus ele_0 \oplus ele_1 \oplus @, ele \oplus ele \oplus ele_1 \oplus @) \vdash$$

und Beispiel *merge3* (3.5)

$$ele_0 \neq ele, ele_2 \in x_4 \oplus ele_1 \oplus x_8,$$
$$isperm(x_6, delonce(ele_2, x_4 \oplus ele_1 \oplus x_8)),$$
$$\neg\ isperm(ele_0 \oplus ele_1 \oplus ele_2 \oplus x_6, ele_1 \oplus x_4 \oplus ele \oplus ele_1 \oplus x_8),$$
$$\neg\ isperm(ele \oplus ele_0 \oplus ele_1 \oplus ele_2 \oplus x_6, ele \oplus ele_1 \oplus x_4 \oplus ele \oplus ele_1 \oplus x_8) \vdash$$

noch einen Test mit den konkret vorgegebenen Belegungen für das Gegenbeispiel machen. Für das Beispiel *merge1* ist die Testsequenz mit der konkreten Belegung dieselbe, wie in dem Fall, in dem die Verschiedenheit der Elemente *ele* und ele_0 in die Formel kodiert wurde, da nur diese Elemente frei in der Sequenz vorkommen. Der Vollständigkeit halber geben wie das Ergebnis nochmals mit an. Damit erhalten wir für Mergesort auf Listen insgesamt neun Versuche. Im Fall 1 wird das zusätzliche Axiom in die Theorie mit aufgenommen, das die Existenz verschiedener Elemente garantiert, im Fall 2 wird die Verschiedenheit der Elemente in der Formel selbst garantiert und im dritten Fall wird das konkrete Gegenbeispiel vorgegeben. Bei dieser Testreihe konnten zwar einige Beispiele gelöst werden, aber kein automatischer Theorembeweiser konnte das Beispiel *merge3* (3.5) lösen.

4.3.3 Reguläre Ausdrücke

Zusätzlich zu den Beispielen *match1* (3.6)

$$\vdash @_r{}^*\ matches\ @_s$$

und *match2* (3.7)

$$\vdash mkreg(b)^*\ matches\ @_s,$$

geben wir noch zwei weitere Beispiele für die Tests an. Diese entstehen aus den Beispielen von oben, indem die Definition von *matches* eingesetzt wird.

Beispiel 4.4 (match1′)

$$\vdash @_r{}^*\ matches\ @_s,\ \exists\ s_1,\ s_2.\ @_s = s_1 +\!+ s_2 \wedge @_r\ matches\ s_1 \wedge @_r{}^*\ matches\ s_2$$

Beispiel	*match1*	*match2*	*match*1′	*match*2′
PROTEIN	— / —	0 / 0	— / —	0 / 0
SPASS	— / 0.9	11.8 / 0.2	11.8 / 11.7	10.8 / 0.2
KIV	1	1	0	0

Abbildung 4.3: Ergebnisse: reguläre Ausdrücke

Beispiel 4.5 (match2′)

$$\vdash mkreg(b)^*\ matches\ @_s,$$
$$\exists\ s_1,\ s_2.\ @_s = s_1 +\!+ s_2 \wedge mkreg(b)\ matches\ s_1 \wedge mkreg(b)^*\ matches\ s_2$$

Für diese vier Beispiele wurde versucht, ein Gegenbeispiel zu finden (siehe Abbildung 4.3). Dabei schloß der automatische Theorembeweiser SPASS gut ab. PROTEIN konnte immerhin die Hälfte der Beispiele lösen.

4.4 Auswertung

Bei den Beispielen über die Binärsuche auf Arrays hat keiner der automatischen Theorembeweiser überzeugt. Nur SPASS konnte eines der Beispiele lösen. Sieht man sich die Beispiele an, erkennt man die große **Schachtelungstiefe** der Funktionen. Der Konstruktor 0, der in *mkarray* verwendet wird, liegt im Beispiel $<_{sor}\ (put(mkarray(1), 0, ele), 0, 0) \vdash$ in Schachtelungstiefe vier. Damit kommen die automatischen Theorembeweiser offensichtlich nicht zurecht.

Die Anzahl der Variablen, für die eine Belegung gesucht werden muß, spielen im Beispiel Mergesort eine große Rolle. Die Beispiele *match1* (3.3) und *match2* (3.4), in denen keine Variablen für Listen instantiiert werden mußten, konnten die automatischen Theorembeweiser noch lösen. Das Beispiel *match3* (3.5) konnte nicht mehr gelöst werden, obwohl es den beiden anderen sehr ähnlich ist.

Der KIV-Simplifier scheint sich von der Schachtelungstiefe der Funktionen und von der Anzahl der freien Variablen weniger beeindrucken zu lassen. In den Beispielen der Binärsuche auf Arrays und Mergesort konnte der Simplifier die Lösungen finden, sobald die konkreten Gegenbeispiele vorlagen.

Die Beispiele über reguläre Ausdrücke hat SPASS, zumindest mit der reduzierten Axiomenmenge, alle lösen können. Dagegen konnte PROTEIN nur zwei der vier Beispiele lösen. Auch insgesamt konnte SPASS mehr Beispiele lösen, als der automatische Theorembeweiser PROTEIN. Wenn wir SPASS für die Gegenbeispiel-Suche verwenden wollen, müssen wir ihn immer mit konkreten Belegungen aufrufen, damit dem Benutzer ein konkretes Gegenbeispiel geliefert werden kann, denn SPASS besitzt kein Antwortprädikat. Betrachten wir uns deshalb die Beispiele, in denen konkrete Werte vorlagen. Dies sind für die Binärsuche auf Arrays die Beispiele *hoare*1′ (4.1), *hoare*1″ (4.2) und *hoare*1‴ (4.3), für Mergesort alle Beispiele im dritten Fall und alle Beispiele über reguläre Ausdrücke. Diese Beispiele kann der KIV-Simplifier fast alle lösen. Nur für zwei Beispiele über reguläre Ausdrücke wird eine Interaktion benötigt. SPASS hat dagegen bei den Beispielen über die Binärsuche auf Arrays große Probleme. Auch bei den Beispielen zu Mergesort zeigt sich, daß, mit zunehmender Variablenanzahl, SPASS an seine Grenzen stößt.

Aus diesem Test erkennt man, daß sich beide automatischen Theorembeweiser nicht für die Gegenbeispiel-Suche eignen. PROTEIN, der ein Antwortprädikat besitzt, mit dem konkrete Gegenbeispiele geliefert werden können, löst nur wenige Beispiele. Der automatische Theorembeweiser SPASS schneidet besser als PROTEIN ab, da für ihn aber konkrete Werte vorgegeben werden müssen und für diese Fälle der KIV-Simplifier besser abschneidet, kann sich auch SPASS nicht für die Gegenbeispiel-Suche auszeichnen. Deshalb müssen wir eine eigene Lösung für die Gegenbeispiel-Suche finden. Im nächsten Kapitel stellen wir eine Lösung vor, die durch Beweisfortführung versucht, Gegenbeispiele zu findet.

Kapitel 5

Gegenbeispiele durch Beweisfortführung

Im vorigen Kapitel sahen wir, daß eine Lösung mit automatischen Theorembeweisern keine befriedigende Ergebnisse liefert. Das Beispiel *merge* (3.2) konnte nicht ohne weiteres gelöst werden, da es Bedingungen an eine Parametersorte stellt. Ist eine Spezifikation unterspezifiziert oder enthält nicht erzeugte Parametersorten, kann eine Formel φ in einem Modell der Spezifikation gültig sein, in einem anderem jedoch nicht. Da φ nicht in allen Modellen gültig ist, gibt es ein Gegenbeispiel. Die Spezifikation erfüllt jedoch nicht $Cl_{\exists} \neg \varphi$, da es auch Modelle gibt, in denen φ gültig ist.

Gegenbeispiele dieser Art können mit automatischen Theorembeweisern nicht gefunden werden, denn das System hat keine Informationen über Parametersorten oder unterspezifizierten Teilen einer Spezifikation. Für den Benutzer eines Beweissystems ist es meist einfach zu sehen, ob es Modelle in der Spezifikation gibt, die Bedingungen über Parametersorten oder unterspezifizierten Teilen der Spezifikation erfüllen. Deshalb sollte der Benutzer in einen Gegenbeispielbeweis eingreifen können, damit er diese Fälle entscheiden kann.

Wir entwickeln eine Methode, die eine Formel φ soweit vereinfacht, bis nur noch Bedingungen an Parametersorten und unterspezifizierten Teilen der Spezifikation übrigbleiben. Der Benutzer entscheidet dann für diese Bedingungen, ob ein Gegenbeispiel vorliegt, oder nicht. Damit sich der Benutzer nicht in einen neuen Beweis eindenken muß, führen wir den Originalbeweis weiter. Um dabei ein Gegenbeispiel zu erhalten, muß ein Widerspruch abgeleitet werden. Wurde **false** oder eine widersprüchliche Aussage über den Parametersorten abgeleitet, haben wir ein Gegenbeispiel gefunden. Damit der Benutzer Hinweise auf die Fehlerursache erhält, geben wir ihm die Bedingungen zurück, die zum Gegenbeispiel geführt haben.

Die Beweisfortführung bietet noch den Vorteil, daß für eine in allen Modellen gültige Formel φ, bei der wir ein Gegenbeispiel vermuten und die Gegenbeispiel-Suche aufrufen, eventuell ein Beweis für die Formel φ gefunden werden kann. Deshalb kann durch die Beweisfortführung manchmal eine erfolglose Gegenbeispiel-Suche verhindert werden.

Im nächsten Abschnitt beschreiben wir ausführlich das Vorgehen, ein Gegenbeispiel zu finden. Zuerst zeigen wir Bedingungen, die für die Beweisfortführung garantieren, daß, falls ein Gegenbeispiel gefunden wird, auch eines für die Originalaussage existiert. Danach sehen wir, wie man aus dem entstandenen Beweisbaum das konkrete Gegenbeispiel erhält. Für die Beweisfortführung werden heuristische Verfahren benutzt, die wir im Anschluß besprechen. Zum Schluß dieses Abschnittes sprechen wir noch Erweiterungen an, die die Gegenbeispiel-Suche verbessern können. Im darauffolgenden Abschnitt bewerten wir die entwickelte Methode, und gehen darauf ein, warum die Methode nicht vollständig ist und welche Vorteile die Beweisfortführung mit sich bringt. Im letzten Abschnitt dieses Kapitels vergleichen wir die Gegenbeispiel-Suche durch Beweisfortführung mit ähnlichen Ansätzen.

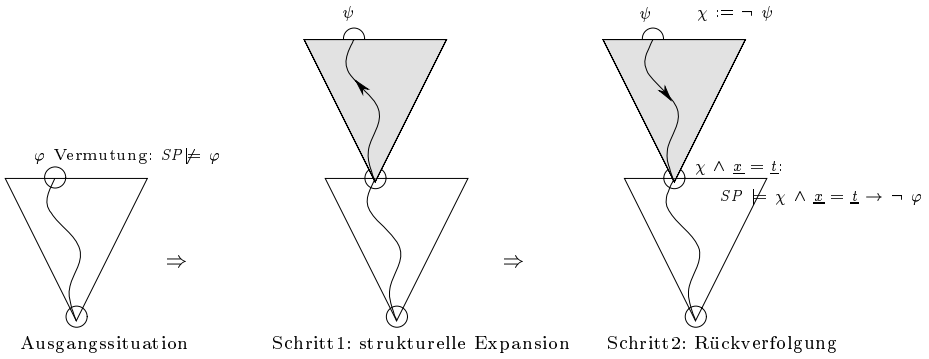

Abbildung 5.1: Ablauf der Gegenbeispiel-Suche

5.1 Vorgehensweise und Korrektheit

Um den Ablauf der Gegenbeispiel-Suche zu verdeutlichen sehen wir uns die Abbildung 5.1 an. Die Ausgangssituation ist eine Stelle in einem Beweisversuch, bei der der Benutzer vermutet, daß die Aussage φ nicht mehr bewiesen werden kann. Um die Vermutung $SP \not\models \varphi$ zu bestätigen, wird die Gegenbeispiel-Suche aufgerufen. Sie führt den Beweis mit einem eingeschränkten Regelsatz weiter, bis eine Aussage ψ erreicht wird, wobei $\neg\ \psi$ erfüllbar ist. Der Regelsatz muß eingeschränkt werden, damit aus der Erfüllbarkeit von $\neg\ \psi$ die Erfüllbarkeit von $\neg\ \varphi$ abgeleitet werden kann. Die Formel ψ darf nur Aussagen über Parametersorten oder unterspezifizierten Teilen der Spezifikation enthalten. Sonst könnte dem Benutzer sofort die Formel φ vorgelegt werden, und es bliebe ihm die Entscheidung überlassen, ob sie widersprüchlich ist oder nicht.

Sind wir bei der Formel ψ angelangt, wissen wir, daß ein Gegenbeispiel existiert. Der letzte Schritt ist nun, aus der Ableitung die Bedingungen wieder herauszurechnen, die zu dem Gegenbeispiel geführt haben. Diese Bedingung sollen dem Benutzer zur Verfügung gestellt werden, damit er Hinweise auf die Fehlerursache erhält. Für $\chi := \neg\ \psi$ erhalten wir aus der Rückberechnung eine Bedingung $\chi \wedge \underline{x} = \underline{t}$, mit der das Gegenbeispiel gezeigt werden kann. Es gilt dann $SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varphi$. Dabei entspricht χ den Bedingungen über den Parametersorten bzw. den unterspezifizierten Teilen der Spezifikation, für die der Benutzer entscheidet, ob sie erfüllbar sind oder nicht. Die Bedingung χ schränkt also die Modelle ein, in denen ein Gegenbeispiel existiert, und die Bedingung $\underline{x} = \underline{t}$ bindet die freien Variablen von φ an die Konstruktorterme, mit denen $\neg\ \varphi$ gültig ist. Nun können wir den Begriff **Gegenbeispiel** genau definieren.

Definition 5.1 (Gegenbeispiel)
Sei χ eine Formel, $\underline{x}$ der Variablenvektor $(x_1 \ldots x_n)$ und $\underline{t}$ ein Vektor von Konstruktortermen $(t_1 \ldots t_n)$ mit $\underline{x} = free(\varphi)/param(\varphi)$, $free(\chi) = param(\chi)$ und für alle i gilt: $x_i \notin free(t_i)$. Das Tupel $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ ist ein Gegenbeispiel für eine Formel φ genau dann, wenn ein Modell $\mathcal{A}$ existiert mit

$$\mathcal{A}\ \in Mod(SP \cup Cl_{\exists}\ \chi) \tag{5.1}$$

und

$$SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varphi \tag{5.2}$$

Durch 5.1 erhalten wir ein Modell $\mathcal{A} \in Mod(SP)$, in dem χ erfüllbar ist. In diesem Modell $\mathcal{A}$ ist auch $\neg\ \varphi^{\underline{t}}_{\underline{x}}$ gültig. Dies wird durch 5.2 garantiert. Damit ist sichergestellt, daß es ein Modell $\mathcal{A}$ und eine Belegung v gibt, mit $\mathcal{A}, v \models \neg\ \varphi$ und somit $SP \not\models \varphi$ gilt. Die Bedingungen, die für ein Gegenbeispiel an die freien Variablen gestellt werden, garantieren, daß der Benutzer nur Bedingungen über Parametervariablen entscheiden muß, und daß für jede freie Variable von φ eine Belegung berechnet wird.

Beispiel 5.1
Für eine Formel

$$\varphi = ord(ele_1 \oplus x) \wedge ord(ele_2 \oplus y) \rightarrow ord((ele_1 \oplus x) \odot (ele_2 \oplus y))$$

ist $\mathcal{G} = (ele_2 < ele_1,\ (x, y) = (@, @))$ *ein Gegenbeispiel.*
Dabei schränkt $\chi = e_1 < e_2$ *die Modelle der Spezifikation ein. Es schließt die Modelle aus, in denen die Trägermenge einelementig ist, denn dort gibt es keine zwei Elemente verschiedener Größe. Da es aber Modelle gibt, in denen* $ele_2 < ele_1$ *erfüllbar ist, z.B. in der Theorie der natürlichen Zahlen, und* $SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varphi$ *gültig ist, erhalten wir ein Gegenbeispiel.*

Die Sprechweise, eine Formel φ **hat ein Gegenbeispiel**, benutzen wir, wenn wir ausdrücken wollen, daß für eine Formeln φ ein Gegenbeispiel existiert. Dies gilt dann, wenn $\neg\ \varphi$ erfüllbar ist.

Korrolar 5.1
Ist eine Formel $\neg\ \varphi$ *erfüllbar, so hat* φ *ein Gegenbeispiel.*

Beweis 5.1
Wenn $\neg\ \varphi$ *erfüllbar ist, so gibt es ein Modell* $\mathcal{A}$ *und eine Belegung* v *mit* $\mathcal{A}, v \models \neg\ \varphi$. *Seien* $free(\varphi) = (\underline{x},\ \underline{y})$ *mit* $\underline{x} = gen(\varphi)$ *und* $\underline{y} = param(\varphi)$. *Für alle* i *sei* $v(x_i) = t_{i\mathcal{A},v}$. *O.B.d.A sind alle* t_i *Konstruktorterme. Dann ist* $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$, *mit* $\chi = \neg\ \varphi^{\underline{t}}_{\underline{x}}$ *und* $t = (t_1, \ldots, t_n)$, *ein Gegenbeispiel für* φ. *Nach Konstruktion ist* $\mathcal{A} \in Mod(SP \cup Cl_\exists\ \chi)$ *und* $SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varphi$ *und die Variablenbedingungen der Definition 5.1 sind erfüllt.*

Da wir jetzt wissen, wie ein Gegenbeispiel genau aussehen muß, sehen wir uns die Strategie an, mit der wir es erhalten wollen.

Der erste Schritt ist zu zeigen, daß die Vermutung $SP \not\models \varphi$ richtig ist. Um einen Beweis *für* φ zu erhalten, müssen alle Beweisäste geschlossen, d.h. zu **true** reduziert werden. Dies kann man nicht mehr erreichen, wenn ein Beweisast zu **false** reduziert wurde. Ist dies der Fall, wurde entweder ein Fehler in der Beweisführung begangen, indem eine beweisbaren Formel zu einer nicht beweisbare Formel reduziert wurde, oder φ ist nicht beweisbar. Benutzten wir zur Suche des Gegenbeispiels nur solche Regeln, die eine Formel φ nicht „abschwächen", schließen wir einen Fehler in der Beweisführung aus. Eine Regel ist abschwächend, wenn sie eine beweisbare Formel in eine nicht beweisbare Formel reduzieren kann.

Erhalten wir in einem Beweisast eine Formel χ über Parametersorten bzw. unterspezifizierten Teilen der Spezifikation und es gibt ein Modell $\mathcal{A}$ der Spezifikation, in dem $\mathcal{A} \models Cl_\exists\ \neg\ \chi$, so gilt $\mathcal{A} \not\models \chi$. Damit läßt sich zeigen, daß $SP \not\models \varphi$ gilt, wenn im Beweis keine abschwächenden Regeln verwendet wurden. Welche Regeln nicht abschwächend sind und die Korrektheit der Aussage zeigen wir im Abschnitt 5.1.1.

Der nächste Schritt wäre nun, die Belegungen für das Gegenbeispiel zu berechnen. Dies kann man aus den Regelanwendungen des Gegenbeispielbeweises.

Beispiel 5.2
Sehen wir uns die folgende Ableitung an:

$$\dfrac{\dfrac{\vdash}{ord(@) \vdash}}{x = @, ord(x) \vdash}$$

Es ist leicht zu sehen, daß x *mit* @ *belegt ein Gegenbeispiel für die Konklusio der Ableitung ist.*

Nicht jede Regelanwendung trägt etwas zur Berechnung der Belegung bei. Die zweite Regelanwendung der Ableitung hat keinen Einfluß auf die Belegung. Die erste Regelanwendung, die in einer Formel eine Gleichung zwischen einer Variablen und einem Term einsetzt, ist ein typisches Beispiel einer Regel, die bei der Berechnung der Belegung berücksichtigt werden muß. Für die Variable muß der Term gemerkt werden, der eingesetzt wird. Er repräsentiert die Belegung der

Variable in diesem Schritt. Bei welchen Regeln die Belegung wie angepaßt werden muß, sehen wir im Abschnitt 5.1.2.

Bis jetzt wissen wir, wann man sicher sein kann, daß eine Formel φ nicht mehr bewiesen werden kann und wie wir die Belegung für das Gegenbeispiel erhalten. Betrachten wir nun also die Beweisfortführung. Das Ziel ist es, einen Beweisast mit der Formel **false** zu erhalten. Dies erhalten wir, wenn wir für die freien Variablen von φ die Konstruktorterme einsetzen, die das Gegenbeispiel bilden. Dann kann φ durch Ausrechnen der Funktionswerte zu **false** reduziert werden.

Um die Konstruktorterme für ein Gegenbeispiel zu erhalten, erzeugen wir systematisch alle möglichen, indem wir eine freie Variable durch ihre Konstruktoren ersetzen. Die Parameter der Konstruktoren werden mit neuen, passenden Variablen instantiiert. Damit wir alle Konstruktorterme irgendwann erzeugen können, machen wir eine Fallunterscheidung über die verschiedenen Konstruktoren. In jedem Fall ersetzen wir die Variable durch einen ihrer Konstruktoren. Ein Term wird dadurch erzeugt, daß einer Variable, die *in* einer Konstruktorfunktion vorkommt, ein Wert zugewiesen wird. Werden Konstruktorkonstanten für die Variablen eingesetzt, erhalten wir konkrete Konstruktorterme, welche die Variablenbelegungen repräsentieren.

Beispiel 5.3
Sei die Variable x von der Sorte **list**. *Dann kann man für x die Konstruktoren $x = @$ und $x = n \oplus x'$ zuweisen. Wenn wir beide Fälle betrachten, ist es immer noch möglich, jeden Term für x einzusetzen. Wollen wir z.B. x mit $x = 1 \oplus 2 \oplus @$ instantiieren, so müssen wir $x' = 2 \oplus @$ und $n = 1$ setzen.*

In welchen der Fälle nach dem Gegenbeispiel weitergesucht wird, entscheiden heuristische Verfahren, die wir im Abschnitt 5.1.3 betrachten.

5.1.1 Existenz eines Gegenbeispiels

In diesem Abschnitt wollen wir den intuitiven Begriff „nicht abschwächende" Regel formal als **invertierbare** Regel definieren. Darauf aufbauend zeigen wir, daß durch ausschließliche Verwendung invertierbarer Regeln ein Kriterium angegeben werden kann, mit dem man die Vermutung $SP \not\models \varphi$ bestätigen kann.

Definition 5.2 (invertierbare Regel)
Eine Regel

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

heißt genau dann invertierbar, wenn für jedes $\mathcal{A}$ gilt:

$$\text{aus } \mathcal{A} \models \Gamma \vdash \Delta \text{ folgt } \mathcal{A} \models \Gamma_i \vdash \Delta_i \quad \text{für } 1 \leq i \leq n \tag{5.3}$$

Das bedeutet, wenn die Konklusio einer invertierbaren Regel gültig ist, dann sind auch die Prämissen gültig. Solche Regeln verlieren keine Information. Durch Kontraposition erhält man daraus, daß aus einer nicht gültigen Prämisse einer invertierbaren Regel eine nicht gültige Konklusio folgt. Gibt es also ein Gegenbeispiel für eine Prämisse, so auch für die Konklusio. Betrachten wir zuerst die nicht invertierbaren Regeln, die in KIV verwendet werden.

Nicht invertierbare Regeln

Nicht invertierbare Regeln sind solche, die aus einer beweisbaren Formel eine nicht mehr beweisbare Formel machen können. Wird eine solche Regel angewendet, kann es geschehen, daß für eine der Prämissen ein Gegenbeispiel gefunden werden kann, obwohl es für die Konklusio keines gibt. Nicht invertierbare Regeln dürfen nicht für die Suche eines Gegenbeispiels verwendet werden, denn für eine Aussage, die zu **false** reduzierte werden konnte, kann man dann nicht sagen, ob ein Gegenbeispiel existiert, oder nicht. Die nicht invertierbaren Regeln der DL-Logik sind

- weakening
- while right

und zusätzlich für nicht deterministische Programme

- execute call
- contract call (left/right)
- execute loop

***weakening*-Regel** Um zu zeigen, daß die *weakening*-Regel

$$\frac{\Gamma \vdash \Delta}{\Gamma,\ \varphi \vdash \Delta}\ \text{weakening}$$

nicht invertierbar ist, sehen wir uns ein Beispiel an.

Beispiel 5.4 (weakening-Regel)
Die folgende Ableitung führt zu einer Formel, die ein Gegenbeispiel hat, obwohl die Konklusio gültig ist.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ele_1 < ele_2}\ simplifier}{ord(ele_1 \oplus @) \wedge ord(ele_2 \oplus @) \vdash ord(ele_1 \oplus @ \odot ele_2 \oplus @)}\ \vdots\ cut\ y = ele_2 \oplus @}{ord(ele_1 \oplus @) \wedge ord(y) \vdash ord(ele_1 \oplus @ \odot y)}\ \vdots\ cut\ x = ele_1 \oplus @}{ord(x) \wedge ord(y) \vdash ord(x \odot y)}\ weakening\ x.last < y.first$$

Da es Modelle gibt (z.B. die Theorie der natürlichen Zahlen) die $\neg\ (ele_1 < ele_2)$ *erfüllen, hätten wir ein Gegenbeispiel gefunden. Wie man sich aber leicht überzeugen kann, ist die Aussage* $ord(x) \wedge ord(y) \wedge x.last < y.first \vdash ord(x \odot y)$ *gültig (siehe auch Abschnitt 2.2).*

Deshalb darf die *weakening*-Regel nicht bei der Suche eines Gegenbeispiels verwendet werden. Daß das Kriterium der invertierbaren Regel hinreichend für die Korrektheit der Suche ist, zeigen wir weiter unten.

***while*-Regel** Für die *while*-Regel ist es leicht einzusehen, daß sie nicht invertierbar ist. Die Regel lautet

$$\frac{\Gamma \vdash \text{INV}, \Delta \qquad \text{INV}, \varepsilon \vdash \langle\alpha\rangle\text{INV}, \Delta \qquad \text{INV}, \neg\ \varepsilon \vdash \varphi, \Delta}{\Gamma \vdash \langle \textbf{while}\ \varepsilon\ \textbf{do}\ \alpha\rangle\varphi, \Delta}\ \text{while}$$

wobei INV die Schleifeninvariante ist, die vom Benutzer eingegeben wird. Wird sie zu stark gewählt, kann sie nicht aus den Bedingung der Konklusio gefolgert werden, und die erste Prämisse kann nicht beweisen werden. Wird die Invariante zu schwach gewählt, können die zweite und dritte Prämisse nicht beweisen werden, da in den Prämissen die Vorbedingungen zu schwach sind. Obwohl die Konklusio beweisbar ist, kann deshalb durch eine falsche Invariante eine Sequenz unbeweisbar werden.

***execute call, contract call left/right, execute loop*-Regeln** (siehe Anhang B.1) Diese Regeln sind nur für nicht deterministische Programme nicht invertierbar, und zwar genau dann, wenn es der Beweis erfordert, daß zwei Programmausführungen des nicht deterministischen Programms

verschiedene Ergebnisse liefern. Denn bei den Regeln werden immer zwei Programmaufrufe gleicher Programme mit gleichen Parametern zu einem Programmaufruf zusammengefaßt. Sehen wir als Beispiel die *execute call*-Regel an.

$$\frac{\Gamma \vdash \underline{\sigma} = \underline{\tau} \quad \langle \mathbf{proc}\ \delta_1 \mathbf{in}\ f(\underline{\sigma};\ \underline{x})\rangle(\underline{x} = \underline{y}), \varphi^{\frac{y}{x}}_{\underline{x}}, \Gamma \vdash \psi^{\frac{y}{z}}_{\underline{z}}}{\langle \mathbf{proc}\ \delta_1 \mathbf{in}\ f(\underline{\sigma};\ \underline{x})\rangle\varphi, \Gamma \vdash \langle \mathbf{proc}\ \delta_2 \mathbf{in}\ f(\underline{\tau};\ \underline{z})\rangle\psi}\ \text{execute call}$$

Ein Beispiel zeigt, daß diese Regel nicht invertierbar ist.

Beispiel 5.5 (execute call)
Sei f eine Funktion, die für die Eingabe 5 sowohl 1 als auch 2 liefern kann. Damit kann die Behauptung

$$x' + 1 = z',\ \langle f(5;\ x)\rangle(x = x') \vdash \langle f(5;\ z)\rangle(z = z')$$

bewiesen werden. Wir führen beide Programmaufrufe aus. Da es für f einen Durchlauf gibt, der 1 ergibt und einen Durchlauf, für den 2 als Ergebnis herauskommt, erhalten wir $1 + 1 = z' \vdash z' = 2$ und damit einen Beweis. Wenden wir aber die execute call*-Regel auf die Sequenz an, erhalten wir*

$$x' + 1 = z',\ \langle f(5;\ x)\rangle(x = y),\ y = x' \vdash y = z'.$$

Führen wir nun das Programm aus, kann $y = 1$ das Ergebnis sein und wir erhalten $1 + 1 = z' \vdash 1 = z'$, was mit $z' = 2$ erfüllbar ist, oder wir erhalten $y = 2$ und somit $2 + 1 = z' \vdash 2 = z'$, und mit $z' = 3$ eine erfüllende Belegung. Damit wurde aus einer beweisbaren Formel eine unbeweisbare.

An diesem Beispiel sehen wir, daß die *execute call*-Regel nicht invertierbar ist. Das Beispiel läßt sich leicht für die Regeln *contract call left/right* und *execute loop* anpassen. Da nicht deterministische Programme in der Praxis der Programmverifikation selten vorkommen, werden diese Regeln für die Gegenbeispiel-Suche jedoch wie invertierbare Regeln behandelt. Die meisten Regeln, die in KIV verwendet werden, sind invertierbar.

Invertierbare Regeln

Für die restlichen Regeln, die in KIV verwendet werden, ist es leicht zu sehen, daß sie invertierbar sind. Deshalb zeigen wir nur exemplarisch ein Beweis für eine prädikatenlogische Regel, der *cut*-Regel, und ein Beweis für eine Programmregel, der *if right*-Regel. Als erstes sehen wir uns die *cut*-Regel an.

***cut*-Regel** Um den Beweis für die Invertierbarkeit übersichtlicher zu gestalten, formulieren wir zwei Lemmata.

Lemma 5.1

$$\models (\alpha \to \beta) \vee \gamma \quad \textit{genau dann, wenn} \models \alpha \to (\beta \vee \gamma)$$

Beweis 5.2 (Lemma 5.1)

$$\begin{aligned} & \models (\alpha \to \beta) \vee \gamma \\ \overset{def.\to}{\Leftrightarrow}\ & \models (\neg\, \alpha \vee \beta) \vee \gamma \\ \overset{asso.}{\Leftrightarrow}\ & \models \neg\, \alpha \vee (\beta \vee \gamma) \\ \overset{def.\to}{\Leftrightarrow}\ & \models \alpha \to (\beta \vee \gamma) \end{aligned}$$

□

Lemma 5.2

$\models (\alpha \to \beta) \vee \neg \gamma$ *genau dann, wenn* $\models (\alpha \wedge \gamma) \to \beta$

Beweis 5.3 (Lemma 5.2)

$$\begin{array}{rl} & \models (\alpha \to \beta) \vee \neg \gamma \\ \overset{def.\to}{\Leftrightarrow} & \models (\neg \alpha \vee \beta) \vee \neg \gamma \\ \overset{asso./com.}{\Leftrightarrow} & \models (\neg \alpha \vee \neg \gamma) \vee \beta \\ \overset{de\ Morgan/def.\to}{\Leftrightarrow} & \models \neg (\alpha \wedge \gamma) \to \beta \end{array}$$

□

Damit kann man nun zeigen, daß die *cut*-Regel

$$\frac{\Gamma , \varphi \vdash \Delta \quad \Gamma \vdash \Delta , \varphi}{\Gamma \vdash \Delta} \text{ cut}$$

invertierbar ist.

Beweis 5.4 (Invertierbarkeit der *cut*-Regel)
Fassen wir die Sequenzen als Formeln auf, ist zu zeigen:

wenn $\mathcal{A} \models \Gamma \to \Delta$ *, dann* $\mathcal{A} \models (\Gamma \wedge \varphi \to \Delta) \wedge (\Gamma \to \Delta \vee \varphi)$

$$\begin{array}{rl} & \mathcal{A} \models \Gamma \to \Delta \\ \Rightarrow & \mathcal{A} \models \Gamma \to \Delta \vee (\varphi \wedge \neg \varphi) \\ \overset{dist.\wedge}{\Rightarrow} & \mathcal{A} \models ((\Gamma \to \Delta) \vee \varphi) \wedge ((\Gamma \to \Delta) \vee \neg \varphi) \\ \overset{Lemma\ 5.1}{\Rightarrow} & \mathcal{A} \models (\Gamma \to (\Delta \vee \varphi)) \wedge ((\Gamma \to \Delta) \vee \neg \varphi) \\ \overset{Lemma\ 5.2}{\Rightarrow} & \mathcal{A} \models (\Gamma \to (\Delta \vee \varphi)) \wedge ((\Gamma \wedge \varphi) \to \Delta) \end{array}$$

□

***if right*-Regel** Die *if right*-Regel hat die folgende Gestalt:

$$\frac{\Gamma , \varepsilon \vdash \langle \alpha \rangle \varphi , \Delta \quad \Gamma , \neg \varepsilon \vdash \langle \beta \rangle \varphi , \Delta}{\Gamma \vdash \langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta \rangle \varphi , \Delta} \text{ if right}$$

Beweis 5.5 (Invertierbarkeit der *if right*-Regel)
Nach den Basisregeln der dynamischen Logik (siehe Anhang C) gilt

$$\models [\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta]\varphi \leftrightarrow ((\varepsilon \to [\alpha]\varphi) \wedge (\neg \varepsilon \to [\beta]\varphi))$$

und damit auch

$$\models \langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta \rangle \varphi \leftrightarrow ((\varepsilon \to \langle \alpha \rangle \varphi) \wedge (\neg \varepsilon \to \langle \beta \rangle \varphi)).$$

Erst recht gilt dann für alle $\mathcal{A}$*,*

aus $\mathcal{A} \models \langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta \rangle \varphi$ *folgt* $\mathcal{A} \models (\varepsilon \to \langle \alpha \rangle \varphi)$ *und* $\mathcal{A} \models (\neg \varepsilon \to \langle \beta \rangle \varphi))$

und damit die Invertierbarkeit der if right*-Regel.*

□

Für die Konstruktion eines Gegenbeispiels reicht eine Regelanwendung nicht aus. Durch wiederholtes Anwenden von Regeln, entsteht ein Ableitungsbaum. Die Invertierbarkeit von Regeln läßt sich leicht auf einen Ableitungsbaum übertragen.

Korrolar 5.2 (Transitivität der Invertierbarkeit)
Sei A ein Ableitungsbaum, der nur aus invertierbaren Regeln aufgebaut ist. Sei c die Konklusio und $p_1 \ldots p_n$ die Prämissen von A. Desweiteren ist $\varphi = formula(c)$ und $\psi_i = formula(p_i)$ für $1 \leq i \leq n$. Dann gilt:

$$wenn\ \mathcal{A} \models \varphi,\ dann\ \mathcal{A} \models \psi_i\ für\ 1 \leq i \leq n \tag{5.4}$$

Beweis 5.6
Folgt mit Induktion über die Anzahl der Ableitungsschritte.
□

Nun können wir den zentralen Satz für die Gegenbeispielkonstruktion durch Beweisfortführung zeigen. Haben wir durch Beweisfortführung aus der Formel φ eine Formel ψ ableiten können, für die es ein Modell der Spezifikation gibt, in der $\neg\ \psi$ erfüllbar ist, so existiert ein Gegenbeispiel für φ.

Satz 5.1 (Existenz eines Gegenbeispiels)
Sei A ein Ableitungsbaum, der nur aus invertierbaren Regeln aufgebaut ist, c die Konklusio und $\varphi = formula(c)$. Desweiteren seien $p_1 \ldots p_n$ die Prämissen von A und $\psi_i = formula(p_i)$ für $1 \leq i \leq n$.

Wenn ein Modell $\mathcal{A} \in Mod(SP)$ existiert, in dem ein $\neg\ \psi_i$ erfüllbar ist, dann existiert ein Gegenbeispiel für die Konklusio c.

Beweis 5.7
Wir führen einen Widerspruchsbeweis über die Gültigkeit der Konklusio in $\mathcal{A}$.
Annahme: $\mathcal{A} \models \neg\ Cl_\exists\ \neg\ \varphi$

$$\begin{aligned} \mathcal{A} \models \neg\ Cl_\exists\ \neg\ \varphi \quad &\Leftrightarrow \quad \mathcal{A} \models Cl_\forall\ \varphi \\ &\overset{Korrolar\ 5.2}{\Rightarrow} \quad \mathcal{A} \models Cl_\forall\ \psi_i\ für\ alle\ 1 \leq i \leq n \\ &\quad\ \lightning\ \neg\ \psi_i\ erfüllbar\ in\ \mathcal{A} \end{aligned}$$

Damit ist auch $\neg\ \varphi$ in $\mathcal{A}$ erfüllbar und nach Korrolar 5.1 existiert ein Gegenbeispiel für φ.
□

Als Spezialfall ergibt sich daraus, daß immer ein Gegenbeispiel für die Konklusio existiert, wenn eine Prämisse des Ableitungsbaumes eine leere Sequenz ist.

Korrolar 5.3
Sei A ein Ableitungsbaum mit der Konklusio c und den Prämissen $p_1 \ldots p_n$, der nur aus invertierbaren Regeln aufgebaut ist. Wenn eine Prämisse $p_i = \vdash$ (die leere Sequenz) ist, existiert in allen Modellen der Spezifikation ein Gegenbeispiel für die Konklusio c.

Beweis 5.8
Für $p_i = \vdash$ und $\psi_i = formula(p_i)$ ist $\psi_i =$ **false** *und damit $\neg\ \psi_i$ in allen Modellen erfüllbar. Die Behauptung folgt mit Satz 5.1.*
□

5.1.2 Die Rückverfolgung

Im vorigen Abschnitt sahen wir, wann ein Gegenbeispiel für eine Formel φ existiert, nämlich dann, wenn durch invertierbare Regeln eine Formel ψ abgeleitet werden kann, die ein Gegenbeispiel hat. Um ein konkretes Gegenbeispiel zu erhalten, müssen noch die Konstruktorterme $\underline{t}$ für die freien Variablen $\underline{x}$ bestimmt werden, mit denen $\neg\ \varphi^{\underline{t}}_{\underline{x}}$ erfüllbar ist.

Die Konstruktorterme können aus den Regelanwendungen berechnet werden, die auf dem Pfad von φ bis ψ vorgenommen wurden. Das Vorgehen dabei ist, aus einem Gegenbeispiel für eine Prämisse einer Regel, ein Gegenbeispiel für deren Konklusio zu berechnen.

Definition 5.3 (Rückverfolgungsschritt)
Sei

$$\frac{p_1 \ldots p_n}{c}$$

eine Regel und es existiert ein Gegenbeispiel $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ für ein p_i. Dann berechnet ein **Rückverfolgungsschritt** *ein Gegenbeispiel $\mathcal{G}' = (\chi',\ \underline{x}' = \underline{t}')$ für die Konklusio c.*

Damit läßt sich die Rückverfolgung für einen Pfad im Ableitungsbaum definieren.

Definition 5.4 (Rückverfolgung)
Sei $s_0 \ldots s_n$ ein Pfad in einem Ableitungsbaum, der aus invertierbaren Regeln besteht. So berechnet die **Rückverfolgung** *für $i = n-1 \ldots 0$ sukzessive aus einem Gegenbeispiel $\mathcal{G}_{i+1}$ für s_{i+1} ein Gegenbeispiel $\mathcal{G}_i$ für s_i. Das Ergebnis ist ein Gegenbeispiel $\mathcal{G}_0$ für s_0.*

Um mit der Rückverfolgung ein Gegenbeispiel für die Konklusio s_0 eines Ableitungsbaumes zu erhalten, muß für den Pfad $s_0 \ldots s_n$ ein Gegenbeispiel für s_n existieren. Der Benutzer entscheidet, ob es ein Modell $\mathcal{A}$ der Spezifikation gibt, in dem $\neg\ \psi$ für $\psi = \mathit{formula}(s_n)$ erfüllbar ist. Somit schränkt $\neg\ \psi$ die Modelle ein, in denen ein Gegenbeispiel existiert und $\mathcal{G} = (\neg\ \psi,\ \mathbf{true})$ ist ein Gegenbeispiel für s_n.

Korrolar 5.4
Sei s eine Sequenz und $\psi = \mathit{formula}(s)$ eine Formel mit $\mathit{free}(\psi) = \mathit{param}(\psi)$ und es existiert ein Modell $\mathcal{A} \in \mathit{Mod}(SP)$, das $\neg\ \psi$ erfüllt.
Dann ist $\mathcal{G} = (\chi,\ \mathbf{true})$ mit $\chi := \neg\ \psi$ ein Gegenbeispiel für s.

Beweis 5.9
Nach Voraussetzung existiert ein Modell $\mathcal{A} \in \mathit{Mod}(SP \cup Cl_{\exists}\ \chi)$ und die Variablenbedingungen von Definition 5.1 sind erfüllt. Wegen $\chi := \neg\ \psi$ gilt $SP \models \chi \rightarrow \neg\ \psi$.
□

Wurde für s_n die leere Sequenz abgeleitet, so ist $\chi = \mathbf{true}$ und man hat für alle Modelle der Spezifikation ein Gegenbeispiel gefunden. Die Korrektheit der Rückverfolgung läßt sich auf die Korrektheit eines einzelnen Rückverfolgungsschrittes zurückführen.

Satz 5.2 (Korrektheit der Rückverfolgung)
Sei A ein Ableitungsbaum, der aus invertierbaren Regeln besteht, und $s_0 \ldots s_n$ ein Pfad in A. Wenn für $\psi = \mathit{formula}(s_n)$ gilt, daß $\mathit{free}(\psi) = \mathit{param}(\psi)$ ist, und es existiert ein Modell $\mathcal{A} \in \mathit{Mod}(SP)$, das $\neg\ \psi$ erfüllt, dann berechnet die Rückverfolgung ein Gegenbeispiel $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ für $\varphi = \mathit{formula}(s_0)$.

Beweis 5.10
Der Beweis erfolgt mit Induktion über die Länge n des Pfades $s_0 \ldots s_n$.
$n = 0$: ok, mit Satz 5.4
$n \rightsquigarrow n+1$: folgt aus der Korrektheit der einzelnen Rückverfolgungsschritte.
□

Es bleibt noch die Korrektheit der einzelnen Rückverfolgungsschritte zu zeigen. Dazu geben wir für die einzelnen Regeln die Berechnungen der Rückverfolgung an und zeigen deren Korrektheit. Nach Definition schränkt die Formel χ eines Gegenbeispiels $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ die Wahl der Modelle ein, in denen ein Gegenbeispiel existiert. Liegt ein Gegenbeispiel im Modell $\mathcal{A}$ für eine Prämisse einer invertierbaren Regel vor, garantiert die Invertierbarkeit, daß auch ein Gegenbeispiel für die Konklusio der Regel in $\mathcal{A}$ existiert. D.h., das Modell, in dem das Gegenbeispiel existiert, ändert sich bei einem Rückverfolgungsschritt nicht. Deshalb ändert sich auch die Formel χ bei einem Rückverfolgungsschritt einer invertierbaren Regen nicht.

Es kann aber sein, daß sich die Belegung für das Gegenbeispiel ändern muß. Da es aber viele Regeln gibt, bei denen die Belegung nicht geändert werden muß, definieren wir ein Kriterium, welches diese Regeln charakterisiert.

Definition 5.5 (starke Invertierbarkeit)
Eine Regel

$$\frac{p_1 \ldots p_n}{c}$$

heißt genau dann **stark invertierbar**, *wenn*

$$\models formula(c) \rightarrow formula(p_1) \wedge \ldots \wedge formula(p_n) \tag{5.5}$$

gilt.

Korrolar 5.5
Eine Regel ist genau dann stark invertierbar, wenn für alle $\mathcal{A} \in Mod(SP)$ und alle Belegungen v gilt:

$$\text{wenn } \mathcal{A}, v \models formula(c), \text{ dann auch } \mathcal{A}, v \models formula(p_i) \text{ für alle } 1 \leq i \leq n \tag{5.6}$$

Beweis 5.11
Das Korrolar folgt nach der Definition von $\models$.
□

Satz 5.3 (Rückverfolgung über stark invertierbare Regeln)
Sei

$$\frac{p_1 \ldots p_n}{c}$$

eine stark invertierbare Regel und $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ ein Gegenbeispiel für ein p_i. Dann ist $\mathcal{G}$ ein Gegenbeispiel für c.

Beweis 5.12
Sei $\psi_i = formula(p_i)$ und $\varphi = formula(c)$. Mit Kontraposition der Gleichung 5.5 gilt für ein i:

$\models \neg\ \psi_i \rightarrow \neg\ \varphi$ *(*) und damit*

$$\begin{aligned} & SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \psi_i \\ \overset{(*)}{\Rightarrow} \quad & SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varphi \end{aligned}$$

□

Wenn eine Regel stark invertierbar ist, ist sie erst recht invertierbar. Wie man sich leicht überzeugen kann, sind auch die meisten invertierbaren Regeln stark invertierbar. Ausnahmen sind die Simplifier-Regeln *insert-eq-var-drop* und die *structural induction*-Regel.

***insert-eq-var-drop*-Regel** Die *insert-eq-var-drop*-Regel ist eigentlich eine zusammengesetzte Regel aus der *insert-equation*-Regel und der *weakening*-Regel. Sie hat folgende Form:

$$\frac{\Gamma_x^\tau \vdash \Delta_x^\tau}{x = \tau, \Gamma \vdash \Delta} \text{ insert-eq-var-drop}$$

oder

$$\frac{\dfrac{\Gamma_x^\tau \vdash \Delta_x^\tau}{x = \tau, \Gamma_x^\tau \vdash \Delta_x^\tau} \text{ weakening}}{x = \tau, \Gamma \vdash \Delta} \text{ insert-equation}$$

Die Regel wird nur angewendet, wenn $x \notin \mathit{free}(\tau)$. Die Prämisse $\Gamma_x^\tau \vdash \Delta_x^\tau$ enthält dann keine freie Variable x mehr. Deshalb ist die Regel trotz ihres *weakening*-Schrittes invertierbar, aber an einem Beispiel sehen wir, daß sie nicht stark invertierbar ist.

Beispiel 5.6
Sei $\mathcal{A}$ ein Modell der Spezifikation und v eine Belegung mit $v(x) = 3$. Dann gilt

$$\mathcal{A}, v \models x = 4 \vdash \mathit{prime}(x), \text{ aber nicht } \mathcal{A}, v \models \vdash \mathit{prime}(4),$$

denn die Belegung $v(x) = 3$ macht den Antezedenten der linken Sequenz falsch und es existiert kein Gegenbeispiel, obwohl mit der Belegung $v(x) = 3$ ein Gegenbeispiel für $\vdash \mathit{prime}(4)$ existiert.

Das Gegenbeispiel $\mathcal{G} = (\mathbf{true},\ \mathbf{true})$ für $\vdash \mathit{prime}(4)$ muß deshalb um die Belegung $x = 4$ erweitert werden, um für $x = 4 \vdash \mathit{prime}(x)$ ein Gegenbeispiel zu sein. Das Gegenbeispiel lautet dann $\mathcal{G}' = (\mathbf{true},\ x = 4)$. Für die *insert-eq-var-drop*-Regel müssen wir deshalb folgenden Rückverfolgungsschritt anwenden.

Satz 5.4 (Rückverfolgungsschritt für die *insert-eq-var-drop*-Regel)
Sei $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ ein Gegenbeispiel für die Prämisse der insert-eq-var-drop*-Regel. Dann ist $\mathcal{G}' = (\chi,\ (\underline{x} = \underline{t})_x^{x_0} \wedge x = \tau)$ das Gegenbeispiel für die Konklusio, wobei x_0 eine neue Variable ist.*

Beweis 5.13
Nach Voraussetzung gibt es ein Modell $\mathcal{A}$ und eine Belegung v mit $\mathcal{A}, v \models \chi \wedge \underline{x} = \underline{t}$ und mit

$$v' = v[x_0 \leftarrow x_{\mathcal{A},v},\ x \leftarrow \tau_{\mathcal{A},v}] \text{ gilt } \mathcal{A}, v' \models \chi \wedge \underline{x} = \underline{t} \wedge x = \tau.$$

Desweiteren gilt nach Voraussetzung: $SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \Gamma_x^\tau \vdash \Delta_x^\tau$

$$\begin{array}{rl}
SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ (\Gamma_x^\tau \rightarrow \Delta_x^\tau) & \\
\overset{x_0 \text{ neu Var.}}{\Rightarrow} & SP \models \chi \wedge (\underline{x} = \underline{t})_x^{x_0} \rightarrow \neg\ (\Gamma\ {}_x^\tau\!\rightarrow \Delta_x^\tau) \\
\overset{x \notin \mathit{free}(\Gamma) \cup \mathit{free}(\Delta)}{\Rightarrow} & SP \models \chi \wedge (\underline{x} = \underline{t})_x^{x_0} \wedge x = \tau \rightarrow \neg\ (\Gamma\ {}_x^\tau\!\rightarrow \Delta_x^\tau) \\
\Rightarrow & SP \models \chi \wedge (\underline{x} = \underline{t})_x^{x_0} \wedge x = \tau \rightarrow \neg\ (\Gamma \rightarrow \Delta) \\
\Rightarrow & SP \models \chi \wedge \underline{x} = \underline{t} \wedge x = \tau \rightarrow x \neq \tau \wedge \neg\ (\Gamma \rightarrow \Delta) \\
\Rightarrow & SP \models \chi \wedge \underline{x} = \underline{t} \wedge x = \tau \rightarrow \neg\ (x = \tau \wedge \Gamma \rightarrow \Delta)
\end{array}$$

Damit sind die Bedingungen für ein Gegenbeispiel nach Definition 5.1 erfüllt.
□

***structural induction*-Regel** Die *structural induction*-Regel ist nicht stark invertierbar. Sie hat für eine Variable x der Sorte s mit dem Erzeugtheitsprinzip s **generated by** c, f folgende Form:

$$\frac{\forall\ \underline{y}.\ \varphi(c, \underline{y}) \quad \forall\ \underline{y}.\ \varphi(x', \underline{y}) \vdash \varphi(f(x'),\ \underline{y})}{\Gamma \vdash \Delta} \text{ structural induction}$$

wobei $\varphi(\underline{y}) = \Gamma \rightarrow \Delta$ mit $\underline{y} = \mathit{free}(\Gamma \rightarrow \Delta) \backslash x$ und $\mathit{sort}(x) = s$. Dabei ist x die Induktionsvariable und x' eine neue Variable mit $\mathit{sort}(x') = s$.

Beispiel 5.7
Sei $\mathcal{A} \in Mod(SP)$ ein Modell und v eine Belegung mit $v(x') = @$ und $v(x) = @$. Die strukturelle Induktion liefert folgende Regelanwendung:

$$\frac{len(x') < 1 \vdash len(n \oplus x') < 1}{len(x) < 1 \vdash} \text{ structural induction}$$

Dann ist v ein Gegenbeispiel für die Prämisse, aber nicht für die Konklusio.

Der Rückverfolgungsschritt für die *structural induction*-Regel muß die Induktionsvariable mit dem entsprechenden Konstruktorterm belegen.

Satz 5.5 (Rückverfolgungsschritt für die *structural induction*-Regel)
Sei $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ ein Gegenbeispiel für die Prämisse und $sort(x) = s$ **generated by** *c, f, dann ist $\mathcal{G}' = (\chi,\ (\underline{x} = \underline{t})^{x_0}_x) \wedge x = c)$ ein Gegenbeispiel für die Konklusio, falls $\mathcal{G}$ eines für die linke Prämisse war und $\mathcal{G}' = (\chi,\ (\underline{x} = \underline{t})^{x_0}_x) \wedge x = f(x'))$ eines, falls $\mathcal{G}$ für die rechte Prämisse der* structural induction*-Regel ein Gegenbeispiel war.*

Beweis 5.14
Wir zeigen den Beweis für den Fall, wo ein Gegenbeispiel für die rechte Prämisse vorliegt, der Fall für die linke Prämisse läßt sich analog beweisen. Nach Voraussetzung gilt:

$$\begin{aligned}
SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ (\forall\ \underline{y}.\ \varphi(x',\ \underline{y}) \rightarrow \varphi(f(x'),\ \underline{y})) \\
\Rightarrow \quad SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow (\forall\ \underline{y}.\ \varphi(x',\ \underline{y}) \wedge \neg\ \varphi(f(x'),\ \underline{y})) \\
\Rightarrow \quad SP \models \chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varphi(f(x'),\ \underline{y}) \\
\overset{x \notin free(f(x')),\ x \notin \underline{y}}{\Rightarrow} \quad SP \models \chi \wedge (\underline{x} = \underline{t})^{x_0}_x \wedge x = f(x') \rightarrow \neg\ \varphi(f(x'),\ \underline{y}) \\
\Rightarrow \quad SP \models \chi \wedge (\underline{x} = \underline{t})^{x_0}_x \wedge x = f(x') \rightarrow \neg\ \varphi(x,\ \underline{y})
\end{aligned}$$

□

Rückverfolgung über nicht invertierbare Regeln

Bis jetzt haben wir die Rückverfolgung nur über invertierbare Regeln betrachtet. Für die Gegenbeispielkonstruktion ist dies auch vollkommen ausreichend, denn zur Suche eines Gegenbeispiels werden nur invertierbare Regeln verwendet. Hat man ein Gegenbeispiel für eine Prämisse in einem Beweisversuch gefunden, ist dies meist nicht der frühestmögliche Punkt, an dem ein Gegenbeispiel vorliegt. Für den Benutzer, der den interaktiven Beweis durchführt, ist ein wichtiger Hinweis auf die Ursache des Fehlers der Punkt, an dem zum ersten Mal ein Fehler aufgetreten ist. Deshalb ist es Ziel der Rückverfolgung, den Fehler nicht nur bis zu dem Punkt zurückzuverfolgen, an dem die Gegenbeispiel-Suche gestartet wurde, sondern soweit zurück, wie möglich.

Da wir keine nicht deterministischen Programme betrachten wollen, müssen nur die nicht invertierbaren Regeln *weakening* und *while right* betrachtet werden.

***weakening*-Regel** Sei $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ ein Gegenbeispiel für die Prämisse von

$$\frac{\Gamma \vdash \Delta}{\Gamma,\ \varphi \vdash \Delta} \text{ weakening}$$

Damit $\mathcal{G}$ noch ein Gegenbeispiel für die Konklusio ist, muß φ unter den Voraussetzungen von $\mathcal{G}$ erfüllbar sein. Wenn man $\chi \wedge \underline{x} = \underline{t} \rightarrow \varphi$ beweisen kann, hat man die Gültigkeit von φ unter den Voraussetzungen von $\mathcal{G}$ gezeigt, und damit ist φ erst recht erfüllbar. Sind $free(\varphi) \backslash \underline{x} \neq \varnothing$, kann es sein, daß φ für eine bestimmte Belegung der Variablen $free(\varphi) \backslash \underline{x}$ erfüllbar ist, obwohl

$\chi \wedge \underline{x} = \underline{t} \rightarrow \varphi$ nicht bewiesen werden konnte. Man muß die Erfüllbarkeit von $\chi \wedge \underline{x} = \underline{t} \wedge \varphi$ zeigen. Die Bedingungen $\chi \wedge \underline{x} = \underline{t}$ müssen für den Erfüllbarkeitstest hinzugenommen werden, da dort die Bedingungen für die Erfüllbarkeit von $\Gamma \wedge \neg \Delta$ enthalten sind. Würden wir die Bedingungen nicht beachten, könnte es sein, daß wir die Erfüllbarkeit von φ zeigen können, obwohl $\varphi \wedge \Gamma \wedge \neg \Delta$ nicht erfüllbar ist. Dies müssen wir jedoch zeigen, um das Gegenbeispiel für die Konklusio zu erhalten.

Beispiel 5.8
Sehen wir uns die Ableitung

$$\frac{x \geq 1 \vdash}{x < 1 \wedge x \geq 1 \vdash} \textit{ weakening}$$

an. Die Prämisse ist mit $x = 1$ erfüllbar, wogegen $x < 1$ mit $x = 0$ erfüllbar ist. Dagegen ist $x < 1 \wedge x \geq 1$ unerfüllbar.

Für den Erfüllbarkeitstest von $\chi \wedge (\underline{x} = \underline{t}) \wedge \varphi$ kann der gleiche Mechanismus wie für die Suche eines Gegenbeispiels eingesetzt werden. Wir suchen ein Gegenbeispiel für χ, $(\underline{x} = \underline{t})$, $\varphi \vdash$. Das Gegenbeispiel enthält dann die erfüllenden Belegungen für $\neg$ *formula*$(\chi,\ (\underline{x} = \underline{t}),\ \varphi\ \vdash) = \chi \wedge (\underline{x} = \underline{t}) \wedge \varphi$.

Diese beiden Mechanismen reichen nicht immer aus, um ein Gegenbeispiel für die Konklusio zu erhalten. Denn das Gegenbeispiel für die Prämisse kann schon Belegungen enthalten, die unverträglich mit der Formel φ sind, d.h. φ kann mit diesem Gegenbeispiel nicht gezeigt werden. Trotzdem kann es sein, daß ein anderes Gegenbeispiel die Konklusio widerlegt.

Beispiel 5.9
Bei der Anwendung von

$$\frac{x \leq 1 \vdash}{x \neq 1, x \leq 1 \vdash} \textit{ weakening}$$

haben wir $\mathcal{G} = (\mathbf{true},\ x = 1)$ als Gegenbeispiel für die Prämisse. Aber $x = 1 \wedge x \neq 1$ ist unerfüllbar. Trotzdem gibt es für die Konklusio mit $\mathcal{G}' = (\mathbf{true},\ x = 0)$ ein Gegenbeispiel.

Wenn die beiden oben beschriebenen Methoden keinen Erfolg gezeigt haben, kann man für die Konklusio eine neue Gegenbeispiel-Suche starten, um die Lösung zu erhalten. Findet die Suche kein Gegenbeispiel, muß man die Rückverfolgung an dieser Stelle abbrechen.

Diese drei Methoden unterscheiden sich in ihrer Mächtigkeit. Die letzte ist die mächtigste, aber auch die aufwendigste. Um für $\varphi \wedge \Gamma \wedge \neg \Delta$ eine erfüllende Belegung zu finden, muß man zum Großteil nochmals die gleichen Berechnungen durchführen, wie für $\Gamma \wedge \neg \Delta$. Die zweite Methode ist wegen der Suche von erfüllbaren Belegungen aufwendiger als die allererste Methode, kann aber auch Gegenbeispiele finden, wenn in φ Variablen auftreten, die in Γ und Δ nicht vorkommen.

***while right*-Regel** Die zweite nicht invertierbare Regel ist die *while right*-Regel. Sie soll garantieren, daß die *Schleifeninvariante* zu jedem Zeitpunkt der *while*-Schleife gilt. Mit ihrer Hilfe werden dann die Nachbedingungen der Schleife gezeigt. Die Invariante INV muß somit vor, während und nach der *while*-Schleife gelten. Dies garantiert die Regel

$$\frac{\Gamma \vdash \mathrm{INV}, \Delta \quad \mathrm{INV}, \varepsilon \vdash \langle\alpha\rangle \mathrm{INV}, \Delta \quad \mathrm{INV}, \neg\ \varepsilon \vdash \varphi, \Delta}{\Gamma \vdash \langle \mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha\rangle\varphi, \Delta}\ \text{while right}$$

Das Problem bei der *while*-Regel ist, daß sowohl die Invariante, die der Benutzer eingeben kann, als auch das *while*-Programm an sich, fehlerhaft sein kann. Wenn die Rückverfolgung erkennen würde, wo der Fehler liegt, könnte sie entweder stoppen, und dem Benutzer die falsche Invariante vorlegen, oder mit der Rückverfolgung des Gegenbeispiels für das falsche Programm fortfahren.

Jedoch ist es nicht einfach Kriterien zu finden, die eine Aussage darüber machen, welche Art von Fehler vorliegt. Kann man z.B. die linke Prämisse nicht zeigen, ist nicht ersichtlich, ob man eine zu starke Invariante gewählt hat, oder ob die Konklusio falsch ist.

Einige Kriterien gibt es trotzdem. Hat man ein Gegenbeispiel für die linke Prämisse und kann damit $\neg\ \varepsilon$ erfüllen, hat man eine Belegung gefunden, mit der die Schleife nicht durchlaufen wird. Ist dann $\neg\ \varphi$ erfüllbar, kann man das Gegenbeispiel auch für die Konklusio verwenden. Diese ist gleich $\Gamma \vdash \varphi,\ \Delta$, da die Schleife übersprungen wird und nach Voraussetzung ist $\Gamma \wedge \neg\ \Delta$ erfüllbar. Man muß $\chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ \varepsilon \wedge \neg\ \varphi$ zeigen, um aus der linken Prämisse ein Gegenbeispiel für die Konklusio herzuleiten.

Wenn die rechte Prämisse ein Gegenbeispiel liefert, kann man auf eine ähnliche Weise ein Gegenbeispiel für die Konklusio finden. Wenn man nachweisen kann, daß $\chi \wedge \underline{x} = \underline{t} \rightarrow \Gamma$ gültig ist, hat man wiederum eine Belegung gefunden, in der die *while*-Schleife nicht durchlaufen wird und mit der man $\models \chi \wedge \underline{x} = \underline{t} \rightarrow \Gamma,\ \neg\ \varphi,\ \neg\ \Delta$ zeigen kann. Als Verbesserung dieser Methode kann man auch ein neues Gegenbeispiel für $\Gamma \wedge \neg\ \varphi \wedge \Delta \wedge \varepsilon$ suchen.

Alle vorgestellten Vorgehensweisen finden aber keine Fehler, die in der *while*-Schleife enthalten sind, da die Schleife in keinem der Fälle betrachtet wird. Um solche Fehler zu entdecken, müßte man die Schleife ausführen. Können dann alle anfallenden Bedingungen gezeigt werden, kann das Gegenbeispiel an die Konklusio weiterpropagiert werden. Dieses Vorgehen wird aber wegen seiner Komplexität nicht betrachtet.

Kann man an einer nicht invertierbaren Regel kein Gegenbeispiel für die Konklusio berechnen, so muß man die Rückverfolgung abbrechen. Zu diesem Zeitpunkt weiß man, daß die Prämisse fehlerhaft ist, und wie für sie das Gegenbeispiel aussieht.

5.1.3 Strukturelle Expansion

Das Erfüllbarkeitsproblem läßt sich durch eine Suche lösen. Sucht man eine Belegung für eine Formel φ und hat man für $\underline{x} = \mathit{free}(\varphi)$ eine erfüllende Belegung $\underline{t}$ (durch erraten) gefunden, so läßt sich $\varphi^{\underline{t}}_{\underline{x}}$ bzw. $\underline{x} = \underline{t} \rightarrow \varphi$ beweisen. Ein Lösungsweg wäre nun, alle möglichen Belegungen $\underline{t}_i$ aufzuzählen und $\underline{x} = \underline{t}_i \rightarrow \varphi$ versuchen, zu beweisen. Gelingt dies, hat man eine Lösung gefunden. Dies ist ein *generate & test*-Algorithmus, der im allgemeinen nicht sehr effizient ist [Kum92].

Ein effizienteres Verfahren ist *backtracking*. Dabei werden nicht alle Variablen auf einmal instantiiert, sondern eine nach der anderen. Nach jedem Instantiierungsschritt wird die entstandene Formel vereinfacht. Hat man einen Punkt erreicht, an dem keine Lösung mehr gefunden werden kann, nimmt man die zuletzt gemachten Belegungen soweit zurück, bis für eine Variable eine alternative Instantiierung möglich ist, und setzt dort die Suche fort. Wenn ein Punkt erreicht ist, der eine Lösung ausschließt und die Variablen sind erst teilweise belegt, besteht gegenüber *generate & test* der Vorteil, daß man alle potentiellen Lösungen, die diese teilweise Instantiierung beinhaltet, ausschließen kann. Man kann also einen ganzen Unterbereich des Lösungsraumes ausschließen.

Für diese Methode benötigt man jedoch einen **Simplifier**, der die Vereinfachung der Formel nach jedem Instantiierungs-Schritt vornimmt. Eine Formel gilt als ,,einfacher", wenn sie weniger Funktionen und/oder weniger Variablen enthält. Aus einer Formel sollte deshalb möglichst redundante Information entfernt werden. Ein Simplifier versucht dies durch prädikatenlogische Umformungen und durch Einsetzen von Funktionsdefinitionen zu erreichen. Hat der Simplifier eine Formel zu **true** vereinfacht, ist sie bewiesen. Wurde eine Formel vom Simplifier jedoch zu **false** vereinfacht, hat man eine falsche Instantiierung gewählt und muß sie zurücknehmen.

Kommen in den Formeln Variablen vor, deren Sorten einen unendlichen Wertebereich besitzen, können wir nicht immer nur die zuletzt gemachte Instantiierung zurücknehmen. Sonst kann es geschehen, daß eine Belegung einer Variablen, die keine Lösung mehr zuläßt, nicht mehr zurückgenommen werden kann, da für eine andere Variable, die später instantiiert wurde, alle (unendlich vielen) alternativen Belegungen durchprobiert werden. Man kommt nicht an die Stelle zurück, an der die fehlerhafte Belegung vorgenommen wurde. Damit wir für alle Variablen die Instantiierungen parallel betrachten können, benutzen wir eine Suche [RN95].

Definition 5.6 (Suche)
Bei einer **Suche** *wird, ausgehend vom Ausgangszustand, ein Zustand expandiert, d.h. für ihn werden alle Nachfolgezustände erzeugt. Jeder Zustand kann nur einmal expandiert werden. Dadurch wird ein* **Suchbaum** *erzeugt.*

Da wir ein Gegenbeispiel durch Beweisfortführung erhalten wollen, ist unser Ausgangszustand die Formel φ, für die wir $SP \not\models \varphi$ vermuten. Das Ziel ist **false** oder eine Formel ψ abzuleiten, die keine freien Variablen mehr enthält. Für ψ muß dann entschieden werden, ob $\neg\ \psi$ erfüllbar ist.

Doch wie erhalten wir die Nachfolgezustände für eine Formel. Für eine Variable können wir nicht alle möglichen Belegungen einsetzen, da es unendlich viele geben kann. Deshalb setzen wir die (endlich vielen) Konstruktoren für eine Variable ein. Für erzeugte Sorten gilt, daß jeder Wert durch einen Konstruktorterm repräsentiert werden kann. Lassen wir für die Parameter der Konstruktorfunktionen noch alle möglichen Belegungen offen, d.h. für die Parameter werden neue Variablen eingesetzt, können wir, wenn wir alle Nachfolgezustände gemeinsam betrachten, immer noch Repräsentanten für alle Werte erzeugen. Damit ist nach der Expansion immer noch der gesamte Suchraum abgedeckt.

Beispiel 5.10
Sei $\varphi(n)$ eine Formel, für die eine Belegung gesucht wird. Dabei ist n von der Sorte **nat**, *die durch die Konstruktoren 0 und $+1$ erzeugt wird. Dann sind die Nachfolgezustände $\varphi' := (n = 0) \wedge \varphi(n)$ und $\varphi'' := (n = n' + 1) \wedge \varphi(n)$.*

An diesem Beispiel sieht man, daß für n immer noch alle Belegungen möglich sind, da für n' noch jeder Term eingesetzt werden kann. D.h. mit φ'' werden alle Formeln abgedeckt, für die n Werte > 0 annimmt und mit φ' wird der Fall abgedeckt, in dem $n = 0$ ist.

Zerlegt man den Suchraum mit Hilfe der Konstruktoren $c_1 \ldots c_n$, so teilt man ihn nach der Struktur der Terme, die in den einzelnen Nachfolgeknoten auftreten können, auf. Eine Lösung für den Nachfolgeknoten i kann nur einen Wert ergeben, der durch den Konstruktor c_i erzeugt werden kann. Deshalb nennen wir dieses Vorgehen **strukturelle Expansion**.

Definition 5.7 (strukturelle Expansion)
Sei φ eine Formel, x eine Variable mit $x \in free(\varphi)$ und $c_1 \ldots c_n$ die Konstruktoren der Sorte $s = sort(x)$. Seien $\underline{x}_1 \ldots \underline{x}_n$ passende Variablen(-vektoren) zu $c_1 \ldots c_n$. Erzeugt man für φ die Nachfolger

$$(x = c_1(\underline{x}_1) \wedge \varphi(x)) \ \ldots \ (x = c_n(\underline{x}_n) \wedge \varphi(x))$$

so nennt man dies **strukturelle Expansion**.

Durch einen Expansionsschritt zerlegt man den Suchraum. Damit wir eine vorhandene Lösung finden können, muß ein Expansionsschritt den Suchraum so zerlegen, daß die Gesamtheit aller Zerlegungen wieder den gesamten Suchraum aufspannen. Wie wir gesehen haben, erfüllt die strukturelle Expansion diese Bedingung.

Damit eine vorhandene Lösung wirklich gefunden wird, muß der Suchraum so durchsucht werden, daß jeder Punkt irgendwann erreicht wird. Das bedeutet, daß jeder Zustand (Formel) irgendwann einmal für die Expansion ausgewählt und jede Variable der Formel irgendwann einmal expandiert werden muß. Wenn jeder Zustand für die Expansion einmal ausgewählt wird, erreichen wir, daß jede (teilweise) Belegung der Variablen für die Lösung auch tatsächlich in Betracht gezogen wird. Damit wir auch jeder Variablen einer Formel eine Belegung zuordnen, muß sie irgendwann expandiert werden. Eine **faire Suche** garantiert, daß jeder Zustand ausgewählt wird. Analog dazu muß die Auswahl der Variablen fair sein.

Definition 5.8 (faire Suche)
Wird in einem Suchbaum jeder Zustand (irgendwann) für die Expansion ausgewählt, so nennt man die Suche **fair**.

Eine faire Suche kann man erreichen, indem man alle vorhandenen Zustände expandiert, bevor die erzeugten Nachfolgezustände expandiert werden. Man erhält eine *Breitensuche*. Diese Methode ist meist nicht sehr effizient, weshalb wir heuristische Verfahren zur Auswahl des zu expandierenden Zustandes benutzen. Die Heuristik muß dann dafür sorgen, daß jeder Zustand des Suchbaumes irgendwann einmal betrachtet wird. Wir benutzen eine A^*-Suche, da diese sehr flexibel eingesetzt werden kann.

Definition 5.9 (A^*-Suche)
Sei n ein Zustand, $g(n)$ eine Funktion, die die Kosten bis zu diesem Zustand berechnet und $h(n)$ eine Funktion, die die Kosten vom Zustand n bis zum Ziel abschätzt. N sei die Menge aller Zustände. Die Funktion

$$f(n) = g(n) + h(n)$$

ist die Bewertungsfunktion für die A^*-Suche. Die A^*-Suche setzt an dem Zustand $n' \in N$ die Suche fort, für den gilt:

$$\forall\, n.n \in N \rightarrow f(n') \leq f(n)$$

Die von den Funktionen h und g berechneten Kosten sind Maßzahlen, die den Aufwand für einen Zustand abschätzen sollen. Die A^*-Suche versucht den Zustand als nächstes zu betrachten, der die Lösung mit dem (wahrscheinlich) geringsten Aufwand findet. Je besser die Funktionen h und g den tatsächlichen Aufwand wiederspiegeln, desto schneller findet die Suche die richtige Lösung. Diese Strategie ist flexibel, da durch entsprechende Wahl der Funktionen g und h die Auswahl des nächsten Zustandes leicht modifiziert werden kann, ohne die Strategie ändern zu müssen.

Korrolar 5.6
Die A^*-Suche in einem Baum mit endlichem Verzweigungsfaktor ist fair, wenn es eine Konstante $\delta > 0$ gibt und die Kosten für jede Expansion mindestens δ betragen.

Beweis 5.15 (Skizze)
Durch die Expansion des Zustandes n steigen die Kosten der Nachfolgezustände gegenüber n mindestens um δ. Die Kosten von Zuständen, die nicht expandiert werden, bleiben gleich. Da zu jedem Zeitpunkt nur endlich viele Zustände existierten, bekommen Zustände die (noch) nicht expandiert wurden, irgendwann die minimalen Kosten und werden dann zur Expansion ausgewählt. Für einen ausführlichen Beweis siehe [RN95].

Durch die strukturelle Expansion erreichen wir, daß der Suchbaum nur einen endlichen Verzweigungsfaktor erhält. Die Mindestkosten für die Expansion müssen durch die Funktion g gewährleistet werden.

Im folgenden geben wir Kriterien an, wie die Funktionen g und h gewählt werden können, um die Suche eines Gegenbeispiels fair und effizient zu gestalten. Danach werden Kriterien für eine faire Auswahl der Variablen zur Expansion angegeben.

Die Funktion g

Betrachten wir als erstes die Funktion g. Sie berechnet die Kosten, die für die Suche auf dem Weg bis zum aktuellen Zustand entstanden sind. Damit die Suche fair wird, muß der Funktionswert von g bei jedem Expansionsschritt größer werden (δ Kosten). Für einen Zustand n bedeutet dies, daß für alle Nachfolgezustände n' $g(n) \leq g(n') + \delta$ sein muß.

Dies erreicht man, indem die Funktion g die Summe der Expansionsschritte auf dem Pfad vom Ausgangspunkt bis zum aktuellen Zustand berechnet. Die δ-Kosten erhalten damit den Wert 1. Alternativ dazu kann man auch die Anzahl der Vereinfachungsschritte betrachten. Nach jedem Expansionsschritt wird versucht, die entstandene Formel zu vereinfachen. Dazu wird der Simplifier und weitere Heuristiken aufgerufen. Wenn bei diesem Aufruf mehrere Schritte anfallen, können sich die beiden Maßzahlen durchaus unterscheiden.

Durch die Expansion möchte man erreichen, daß rekursiv definierte Funktionen angewendet werden können. Für diese Funktionen werden die Definitionen eingesetzt und dadurch die Formel vereinfacht. Da eine Variable in einer Formel öfters vorkommen kann, kann *ein* Expansionsschritt *mehrere* Einsetzungsschritte nach sich ziehen. Deshalb sind die Summen der Expansionen und der Einsetzungsschritte nicht identisch. Man könnte für die Funktion g auch die Anzahl der Einsetzungsschritte von Funktionsdefinitionen zählen.

Da alle drei Maßzahlen stark korrelieren, dürfte es keinen großen Unterschied machen, welche wir zur Bestimmung von g heranziehen. Nach einem Expansionsschritt muß nicht immer ein Einsetzungs- bzw. ein Vereinfachungsschritt folgen. Deshalb kann es theoretisch passieren, daß in einem Zweig die Funktion g für die Nachfolgezustände nicht mehr anwächst, wenn nur die Vereinfachungs- bzw. Einsetzungsschritte gezählt werden. Deshalb sollte die Anzahl der Expansionen, eventuell gemischt mit den anderen Kenngrößen, immer berücksichtigt werden, um eine faire Suche zu garantieren. Fassen wir die vorgestellten Kriterien für die Funktion g nochmals zusammen. Es sind die

- Anzahl der Expansionsschritte,
- Anzahl der Vereinfachungsschritte,
- und Anzahl der Einsetzungsschritte für Funktionsdefinitionen.

Die Funktion h

Das Ziel der strukturellen Expansion ist es, alle Variablen erzeugter Sorten aus einer Formel zu entfernen, damit die Formel zu **true**, **false** oder Bedingungen über Parametersorten ausgewertet werden kann und der Such-Algorithmus terminiert. Deshalb könnte man die Anzahl der freien, erzeugten Variablen, die in der Formel vorkommen, in der Funktion h berücksichtigen. Je mehr Variablen in einer Formel vorhanden sind, desto weiter ist man noch vom Ziel entfernt.

Bei einem Expansionsschritt werden für eine Variable ihre Konstruktoren eingesetzt. Für jede Konstruktorfunktion erhält man also neue Variablen. Enthält eine Konstruktorfunktion der Sorte s_1 einen Parameter der Sorte s_2, so führt ein Expansionsschritt für eine Variable der Sorte s_1 eine neue Variable der Sorte s_2 ein. Eine Formel, die Variablen der Sorte s_1 enthält, ist deshalb vermutlich aufwendiger zu lösen, als eine Formel, die nur Variablen der Sorte s_2 enthält, denn bei einer Expansion wird eine Variable der Sorte s_2 eingeführt, die später wieder eliminiert werden muß. Diese Abhängigkeiten sind aus der Spezifikationsstruktur ersichtlich und darauf aufbauend kann man eine Sortenhierarchie erstellen.

Definition 5.10 (Sortenhierarchie)
Eine **Sortenhierarchie** *ist eine Anordnung der Sorten, mit der Bedeutung, daß Formeln, die Variablen enthalten, deren Sorten hoch in der Sortenhierarchie stehen, schwieriger zu lösen sind, als solche, die nur Variablen von Sorten enthalten, die niedriger in der Sortenhierarchie stehen.*

Eine Anordnung der Sorten könnte, wie oben beschrieben, aus der Spezifikationsstruktur folgen. Das Gewicht einer Formel würde sich dann als die mit Werten aus der Sortenhierarchie gewichtete Summe über alle Variablen ergeben. Ähnlich wie bei den Sorten, könnte man auch eine Hierarchie auf Funktionen definieren.

Definition 5.11 (Funktionenhierarchie)
Eine **Funktionenhierarchie** *ist eine Anordnung der Funktionen mit der Bedeutung, daß eine Funktion, die höher in der Hierarchie steht, ,,schwieriger" zu lösen ist, als eine Funktion, die weiter unten in der Hierarchie auftritt.*

Eine Funktionenhierarchie kann man dadurch erhalten, daß die Funktion g nicht weiter oben in der Funktionenhierarchie stehen darf, als die Funktion f, wenn in der Funktionsdefinition von f die Funktion g vorkommt, denn für die Berechnung von f muß man die Definition einsetzen, in der g enthalten ist. Man muß also auch die Funktion g berechnen, um ein Ergebnis für f zu erhalten. Deshalb ist die Berechnung von f mindestens so schwierig wie die von g.

Wie bei den Sorten, gewichtet man eine Formel über die Summe der Gewichte der Funktionen. Dabei kann man unterscheiden, ob ein Formel der Form $f(g(x)) = y$ das Gewicht der Funktion f oder die Summe der Gewichte von f und g zuordnet wird.

Wenn man den Aufwand der Erstellung einer Funktionenhierarchie nicht treiben möchte, so kann man auch nur die Anzahl der Funktionen einer Formel als Gewicht heranziehen. Damit

gewichtet man alle Funktionen gleich stark. Nochmals zusammengefaßt sind die Kriterien für die Funktion h

- Anzahl freier, generierter Variablen,
- Variablen gewichtet mit der Sortenhierarchie,
- Anzahl der Funktionen,
- und Funktionen gewichtet mit der Funktionenhierarchie.

Für die Berechnung der Funktion h wird wohl eine Kombination der verschiedenen Kriterien sinnvoll sein. Welche Kombination gut geeignet ist, untersuchen wir in Abschnitt 7.2.1.

Auswahl der Variablen für die Expansion

In einem Zustand gibt es meist mehrere freie, erzeugte Variablen, die für die Expansion in Frage kommen. Für die Effizienz der Suche ist nicht nur die Suchstrategie wichtig, sondern auch die Auswahl der zu expandierenden Variablen. Wird immer dieselbe Variable expandiert, kann die Gegenbeispiel-Suche unter Umständen überhaupt kein Gegenbeispiel finden, da nie eine Belegung für eine andere Variable erzeugt wird, die für das Gegenbeispiel notwendig ist. Die Suche ist dann nicht fair. Eine nochmalige Expansion einer Variablen $x = c(\underline{y})$ bedeutet, eine Variable y_i aus $\underline{y}$ zu expandieren, denn damit wird implizit auch x ein neuer Wert zugewiesen.

Ist man in einem Zustand angelangt, in dem eine Variable schon öfters expandiert wurde und hat die Lösung noch nicht erreicht, ist es unwahrscheinlich, daß eine nochmalige Expansion dieser Variable Erfolg bringt. Es ist wahrscheinlicher, daß die Expansion einer Variable, die noch nicht expandiert wurde, mehr zur Lösung beiträgt. Deshalb ist die Anzahl der Expansionen, die mit einer Variablen schon gemacht wurden, ein Kriterium für die Auswahl. Eine Variable, die noch nicht so oft expandiert wurde, ist zu bevorzugen.

Hat die Sorte einer Variable nur einen Konstruktor, wird durch Expansion dieser Variable keine weitere Fallunterscheidung eingeführt. Um in den Terminierungsfall der Gegenbeispiel-Suche zu gelangen, müssen alle erzeugten Variablen durch Konstruktorterme ersetzt worden sein. Für diese Variable muß also auf jeden Fall irgendwann einmal der entsprechende Konstruktor eingesetzt werden. Werden zuerst andere Expansionen durchgeführt, muß diese Variable später, in allen durch die Expansion entstandenen Fällen, expandiert werden. Da sie bei der Expansion keine verschiedenen Fälle erzeugen, ist es günstig, Variablen bei der Expansion zu bevorzugen, deren Sorte nur einen Konstruktor besitzt.

Das Ziel der Expansion ist es, die Formel soweit wie möglich zu vereinfachen. Eine Formel kann durch Einsetzen von Funktionsdefinitionen vereinfacht werden. Kommt eine Variable öfters in einer Formel vor, kann die Formel durch Expansion dieser Variable wahrscheinlich stärker vereinfacht werden, als durch Expansion einer Variable, die in der Formel nur einmal auftritt, denn es besteht die Möglichkeit, daß durch diese Expansion mehrere Einsetzungsschritte möglich werden. Außerdem wird, für die Fälle, in denen Konstruktorkonstanten eingesetzt werden, durch die Expansion einer Variable, die öfters in einer Formel auftritt, die Gesamtzahl der freien Variablen einer Formel stark verringert. Dies ist, wie wir oben gesehen haben, ein Kriterium für das Formelgewicht, welches wir ja durch Expansion und Vereinfachung verringern wollen.

Auch die Sorte einer Variable spielt bei der Gewichtung einer Formel eine Rolle. Um das Formelgewicht zu verringern wäre es gut, Variablen zu expandieren, deren Sorten hoch in der Sortenhierarchie stehen. Aus dem gleichen Grund könnte es auch sinnvoll sein, solche Variablen bei der Expansion zu bevorzugen, die in Funktionen auftauchen, die weit oben in der Funktionenhierarchie stehen.

Ziel ist es für die Variablen Konstruktorterme einzusetzen, die dafür sorgen, daß Funktionsdefinitionen eingesetzt werden können, um die Formel zu vereinfachen. Ist die Formel rekursiv über Konstruktoren definiert, so muß man dafür Sorge tragen, daß die entsprechenden Konstruktoren an die richtigen Positionen der Formel gelangen.

Definition 5.12 (rekursive Position)
Ist eine Funktion rekursiv über Konstruktoren der Sorte s definiert, so ist die **rekursive Position** *der Funktion die Position in der Parameterliste, an der Konstruktoren der Sorte s auftreten.*

Durch Expansion einer Variable, die auf einer rekursiven Position steht, wird es ermöglicht, die Definition dieser Funktion einzusetzen und die Formel zu vereinfachen. Deshalb ist es günstig, solche Variablen bei der Expansion zu bevorzugen. Gibt es verschiedene Variablen auf rekursiven Positionen, so ist die Variable zu bevorzugen, die auf der rekursiven Position einer Funktion vorkommt, die hoch in der Funktionenhierarchie steht, denn dadurch wird das Formelgewicht stärker verringert. Fassen wir nochmals alle Kriterien zusammen, die zur Auswahl einer Variablen herangezogen werden können. Es sind die Kriterien

- Anzahl der Expansionen,
- bevorzuge eine Variable, deren Sorte nur einen Konstruktor hat,
- Anzahl der Vorkommen in der Formel,
- Sorte der Variablen,
- und Vorkommen auf rekursiven Positionen.

Auch für die Auswahl der Variablen wird wohl eine Kombination der Kriterien nötig sein. Wie diese Kombination aussehen kann, zeigen die Versuche in Abschnitt 7.2.2.

5.1.4 Erweiterungen der Methode

In diesem Abschnitt werden zwei Erweiterungen der Gegenbeispiel-Suche durch Beweisfortführung vorgestellt. Die eine führt zu einer effizienteren Suche, da nicht alle Formeln der Sequenz betrachtet werden, und somit weniger Bedingungen zu erfüllen sind. Die zweite Erweiterung erhöht die Lesbarkeit der Sequenz, die dem Benutzer zur Entscheidung vorgelegt wird, damit er entscheiden kann, ob ein Gegenbeispiel existiert oder nicht.

Die Induktionshypothese

Die Gegenbeispiel-Suche sollte ohne Induktionshypothese durchgeführt werden. Dadurch sind weniger Bedingungen in der Sequenz zu erfüllen und die Suche findet schneller ein Gegenbeispiel. Die Induktionshypothese für eine Sequenz $\Gamma \vdash \Delta$ bei Induktion über die Variable x hat die Form

$$\text{IND-HYP} = \forall\, x'.\ x' \ll x \rightarrow \forall\, \underline{y}.\ (\Gamma_x^{x'} \rightarrow \Delta_x^{x'})$$

wobei x' eine neue Variable und $\underline{x} = \mathit{free}(\Gamma) \cup \mathit{free}(\Delta) \backslash \{x\}$. Soll ein Gegenbeispiel $\mathcal{G}$ für eine Sequenz auch IND-HYP erfüllen, muß für alle $x' \ll x$ die Formel $\Gamma_x^{x'} \rightarrow \Delta_x^{x'}$ unter $\mathcal{G}$ gelten. Es darf also kein Gegenbeispiel $\mathcal{G}'$ existieren, in dem x einen kleineren Wert bezüglich der $\ll$-Ordnung besitzt. Sonst wäre die Induktionshypothese nicht erfüllt, da mit den Belegungen von $\mathcal{G}'$ die Formel $\exists\ \underline{x},\ x'.\ \neg\ (\Gamma \rightarrow \Delta)$ gelten würde. Beachtet man die Induktionshypothese bei die Gegenbeispiel-Suche, so würde die Suche nur **minimale** Gegenbeispiele im Sinne der $\ll$-Ordnung finden. Dies ist aber nicht nötig und deshalb möchte man die Induktionshypothese für die Gegenbeispiel-Suche nicht beachten.

Daraus ergibt sich aber, daß das Gegenbeispiel $\mathcal{G}$ nie für die ganze Sequenz gilt, sondern immer nur für den Teil ohne der Induktionshypothese. Wenn wir $\mathcal{G}$ bis zur Konklusio c des Beweisbaumes zurückverfolgen können, haben wir kein Problem, denn c enthält keine Induktionshypothese. Nur wenn die Rückverfolgung an einer nicht invertierbaren Regel stoppt, können wir keine genau Aussage machen, da $\mathcal{G}$ nicht für die ganze, darüberliegende Sequenz gilt.

An einer nicht invertierbaren Regel müssen wir bei der Rückverfolgung nur dann stoppen, wenn durch diesen Schritt neue Gegenbeispiele eingeführt wurden, d.h. nach der Regelanwendung ist die Menge der Gegenbeispiele größer, wie vor der Regelanwendung. Stoppt die Rückverfolgung, hat die

Gegenbeispiel-Suche ein Gegenbeispiel geliefert, das in der Menge liegt, die die nicht invertierbare Regel eingeführt hat. Ansosten hätte man das Gegenbeispiel ja über die Regel propagieren können. Dann ist die Erfüllbarkeit der Induktionshypothese unerheblich, da der Grund für den Abbruch der Rückverfolgung die nicht invertierbare Regel ist.

Existiert für die Konklusio kein Gegenbeispiel, kann auch für die Prämisse, die wir ohne Induktionshypothese betrachten, keines gefunden werden, solange nur invertierbare Regeln benutzt werden. Um dies zu verdeutlichen, betrachten wir einen modifizierten Beweisbau, in dem keine Induktion angewendet wird. Durch die *apply induction*-Regel wird für bestimmte Variablenbelegungen die Induktionshypothese in eine Sequenz eingeführt. Um die *apply induction*-Regel zu simulieren, wenden wir an dieser Stelle die *cut*-Regel an, die genau die Bedingung einführt, die im Originalbeweis durch die *apply induction*-Regel eingeführt werden. Dadurch können wir dieselbe Prämisse erzeugen, die auch im Originalbeweis erzeugt wurde, jedoch ohne die Induktionshypothese einzuführen. Da der modifizierte Beweisbaum nur aus invertierbaren Regeln besteht und für die Konklusio kein Gegenbeispiel existiert, kann für die Prämisse kein Gegenbeispiel gefunden werden.

Wenn die Konklusio kein Gegenbeispiel hat, kann für eine Prämisse, bei der die Induktionshypothese nicht betrachtet wird, nur dann ein Gegenbeispiel gefunden werden, wenn eine nicht invertierbare Regel verwendet wurde. Wenn die Rückverfolgung dann an dieser Regel stoppt, ist dies wiederum unabhängig von der Induktionshypothese.

Wir sehen also, daß die Gegenbeispiel-Suche und die Rückverfolgung die Induktionshypothese nicht beachten brauchen. Warum die Darstellung dieses Sachverhaltes so schwierig ist, liegt daran, daß wir bei der Rückverfolgung immer nur ein Gegenbeispiel für einen Teil der Sequenz haben und nie für die ganze Sequenz. Wie wir gezeigt haben, sind die Gründe, warum die Rückverfolgung anhalten muß, aber unabhängig von der Induktionshypothese und da wir keine minimalen Gegenbeispiele verlangen, müssen wir die Induktionshypothese weder bei der Gegenbeispiel-Suche noch bei der Rückverfolgung berücksichtigen.

Elimination von Quantoren

Wenn die Beweisfortführung bei einer Sequenz angekommen ist, die keine freien Variablen mehr besitzt, ist es wünschenswert, daß sie keine Quantoren mehr enthält, denn für den Benutzer ist es schwierig zu entscheiden, ob eine Formel, die Quantoren enthält, Gegenbeispiele hat oder nicht. Deshalb möchten wir die Quantoren möglichst entfernen. Dies kann man, wenn aus den restlichen Bedingungen der Sequenz die Aussage des Quantors abgeleitet werden kann.

Korrolar 5.7 (Quantoren Elimination 1)
Sei $\forall\, x.\ \varphi(x),\ \Gamma \vdash \Delta$ *eine Sequenz und* $\Gamma \vdash \forall\, x.\ \varphi(x),\ \Delta$ *beweisbar. Dann ist* $\forall\, x.\ \varphi(x),\ \Gamma \vdash \Delta$ *äquivalent zu* $\Gamma \vdash \Delta$.

Beweis 5.16
Zeige dazu, daß die Regel

$$\frac{\Gamma \vdash \forall\, x.\ \varphi(x)\ \Delta \qquad \Gamma \vdash \Delta}{\forall\, x.\ \varphi(x),\ \Gamma \vdash \Delta}$$

korrekt und invertierbar ist.

Korrektheit: Wenn $\Gamma \vdash \Delta$ *gilt, dann gilt* $\forall\, x.\ \varphi(x),\ \Gamma \vdash \Delta$ *erst recht.*

Invertierbarkeit: Es sei $\forall\, x.\ \varphi(x),\ \Gamma \vdash \Delta$ *mit dem Ableitungsbaum A beweisbar, dann ist auch* $\Gamma \vdash \Delta$ *beweisbar, denn es gilt:*

$$\dfrac{\dfrac{A}{\forall\, x.\ \varphi(x),\ \Gamma \vdash \Delta} \qquad \dfrac{}{\Gamma \vdash \Delta, \forall\, x.\ \varphi(x)}\ \textit{nach Voraussetzung}}{\Gamma \vdash \Delta}\ \textit{cut}$$

□

Kann bei einem Beweisversuch für $\Gamma \vdash \forall\ x.\ \varphi(x)$ die Ableitung nicht geschlossen werden, sondern es bleibt eine Bedingung $\Gamma' \vdash$ übrig, kann die Formel $\forall\ x.\ \varphi(x)$ durch Γ' ersetzt werden, wenn für die Ableitung nur stark invertierbare Regeln verwendet wurden.

Korrolar 5.8 (Quantoren Elimination 2)
Sei $\Gamma,\ \forall\ x.\ \varphi(x) \vdash$ *eine Sequenz und es existiert ein Beweisversuch*

$$\dfrac{\begin{array}{c}\Gamma' \vdash \\ \vdots\end{array}}{\Gamma \vdash \forall\ x.\ \varphi(x)}\ (*)$$

der nur stark invertierbaren Regeln verwendet. Dann kann man in der Sequenz $\forall\ x.\ \varphi(x)$ *durch* $\neg\ \Gamma'$ *ersetzen.*

Beweis 5.17
Wegen (*) gilt: Aus $\Gamma \rightarrow \forall\ x.\ \varphi(x)$ *folgt* $\neg\ \Gamma'$ *und damit* $(\Gamma \rightarrow \forall\ x.\ \varphi(x)) \rightarrow \neg\ \Gamma'$ *und erst recht*

$$(\Gamma \wedge \forall\ x.\ \varphi(x)) \rightarrow \neg\ \Gamma' \tag{5.7}$$

Für die Sequenz $\neg\ \Gamma',\ \Gamma \vdash \forall\ x.\ \varphi(x)$ *liefert der Ableitungsbaum (*) einen Beweis.*

$$\dfrac{\begin{array}{c}\dfrac{}{\neg\ \Gamma',\ \Gamma' \vdash} \\ \vdots\end{array}}{\neg\ \Gamma',\ \Gamma \vdash \forall\ x.\ \varphi(x)}\ (*)$$

Mit Hilfe von Satz 5.7 kann man folgende Ableitung erstellen:

$$\dfrac{\dfrac{\neg\ \Gamma', \Gamma \vdash}{\forall\ x.\ \varphi(x), \neg\ \Gamma', \Gamma \vdash}\ \textit{Satz 5.7} \qquad \dfrac{\dfrac{}{\neg\ \Gamma',\ \forall\ x.\ \varphi(x),\ \Gamma \vdash \neg\ \Gamma'}}{\forall\ x.\ \varphi(x), \Gamma \vdash \neg\ \Gamma}\ \textit{Gleichung 5.7}}{\forall\ x.\ \varphi(x), \Gamma \vdash}\ \textit{cut}$$

□

Wenden wir diese Regeln auf die Sequenz an, die dem Benutzer vorgelegt wird, kann sie eventuell Quantoren entfernen und dem Benutzer fällt es einfacher zu entscheiden, ob ein Gegenbeispiel existiert, oder nicht.

5.2 Bewertung der Methode

In diesem Abschnitt betrachten wir die Vor- und Nachteile der vorgestellten Methode zur Gegenbeispielkonstruktion. Als erstes gehen wir auf den größten Nachteil, der Unvollständigkeit ein. Danach sehen wir, welche Vorteile die Beweisfortführung bringen kann.

5.2.1 Die Gegenbeispiel-Suche ist nicht vollständig

Sehen wir uns dazu ein Beispiel an. Gegeben ist eine Spezifikation über geordnete Listen, wie in Abbildung 2.2, in der die Axiomatisierung von *merge* und *ord* nicht zusammenpaßt. Nach der Spezifikation von *ord* dürfen keine gleichen Elemente in einer geordneten Liste vorkommen. Dies kann aber mit der Definition

ax-08: $merge(x, @) = x$
ax-09: $ele < ele_0 \rightarrow merge(ele \oplus x, ele_0 \oplus y) = ele \oplus merge(x, ele_0 \oplus y)$
ax-10: $\neg\ ele < ele_0 \rightarrow merge(ele \oplus x, ele_0 \oplus y) = ele \oplus merge(ele \oplus x, y)$.

von *merge* geschehen. Sehen wir uns dazu ein Beispiel an.

Beispiel 5.11

$$merge(ele \oplus @, ele \oplus @) \overset{ax\text{-}10}{=} ele \oplus merge(ele \oplus @, @) \overset{ax\text{-}08}{=} ele \oplus ele \oplus @$$

Da *sort* rekursiv über *merge* definiert ist, gibt es für die Aussage $ord(sort(x))$ ein Gegenbeispiel, nämlich $\mathcal{G}(ele = ele',\ x = ele \oplus ele' \oplus @)$. Dieses Gegenbeispiel kann aber der vorgestellte Algorithmus mit der Definition

ax-11: $sort(@) = @$
ax-12: $sort(x \odot y) = merge(sort(x), sort(y))$

von *sort* nicht finden.

Beispiel 5.12
Sehen wir uns dazu die Ableitungsschritte an, die die Gegenbeispiel-Suche machen würde.

$$\dfrac{\dfrac{\dfrac{}{ord(@)}}{ord(sort(@))} \qquad \dfrac{\dfrac{?}{ord(sort(ele \oplus @))} \qquad \dfrac{\dfrac{?}{ord(sort(ele \oplus ele' \oplus @))} \qquad \dfrac{\vdots}{ord(sort(ele \oplus ele' \oplus ele'' \oplus x'''))}}{ord(sort(ele \oplus ele' \oplus x''))}}{ord(sort(ele \oplus x'))}}{ord(sort(x))}$$

An den Stellen **?** wird eine Anfrage an den Benutzer gestellt, ob die jeweilige Formel erfüllbar ist, oder nicht, den die Formeln enthalten keine freien Variablen mehr. Es muß also der Benutzer selber das Gegenbeispiel berechnen, das doch eigentlich der Simplifier lösen sollte. Der kann aber die Formel $sort(ele \oplus @)$ nicht weiter vereinfachen, da er für die Funktion *sort* nur Axiome für den Terminierungsfall und eine Listenkonkatenation mit $\odot$ hat. Die strukturelle Expansion liefert für die Variablen aber nur Konstruktorterme. Die Beziehung zwischen $\odot$ und $\oplus$ ist in den Axiomen ax-05 und ax-06 beschrieben.

ax-05: $@ \odot x = x$
ax-06: $ele \oplus x \odot y = ele \oplus (x \odot y)$

Die Definition von $\odot$ steht auf der rechten Seite der Gleichung und enthält den Listenkonstruktor $\oplus$. Deshalb kann ein Term, der $\odot$ enthält, durch den Simplifier in einen Term umgeschrieben werden, der nur noch aus $\oplus$ besteht, nicht aber umgekehrt. Für einen Konstruktorterm, der in der Ableitung erzeugt wird, kann deshalb nie ein Term mit $\odot$ entstehen. Damit kann aber auch nicht die Bedingung von Axiom ax-12, welches die Funktion *sort* spezifiziert, erfüllt werden. Da es, außer für die leere Liste, keine weiteren Axiome für *sort* gibt, kann die Funktion nicht vereinfacht werden und das Gegenbeispiel wird nicht automatisch gefunden. Dieses Phänomen tritt immer dann auf, wenn die Spezifikation rekursiv definierte Axiome enthält, die nicht rekursiv über Konstruktoren, sondern über andere Funktionen, definiert sind.

Ein anderer Fall, in dem ein ähnliches Problem auch auftreten kann, sieht man an den Axiomen ax-09 und ax-10 für *merge*. In diesem Fall ist die Definition zwar rekursiv über Konstruktoren, aber sie enthält eine Bedingung über eine Parametersorte. Für eine Parametersorte existieren aber keine Erzeugtheitskriterien und wir können Behauptungen, wie $merge(ele \oplus ele' \oplus @)$, nicht weiter vereinfachen, da nicht bekannt ist, ob $ele < ele'$ oder nicht. Deshalb kann nicht entschieden werden, ob das Axiom ax-09 oder das Axiom ax-10 angewendet werden muß. Der Simplifier kann das Problem nicht vereinfachen.

Wie wir gesehen haben, ist die Gegenbeispiel-Suche nicht vollständig, aber man kann garantieren, daß der Algorithmus immer terminiert, wenn ein Gegenbeispiel existiert. Wie man im Beispiel 5.12 an der linken offenen Prämisse $ord(sort(ele \oplus @))$ aber sehen kann, kann der Algorithmus auch an einer Stelle terminieren, an der kein Gegenbeispiel vorliegt.

Betrachten wir nun die Terminierung im positiven Fall. Wenn ein Gegenbeispiel vorliegt, gibt es in einem Modell eine Belegung, unter der die Behauptung falsch ist. Jeder Wert dieser Belegung

kann durch einen Konstruktorterm repräsentiert werden. Durch die vollständige Such-Strategie, werden alle Konstruktorterme irgendwann einmal erzeugt. Gelangt man in den Zweig, in dem die Konstruktorterme erzeugt wurden, die die Werte der Belegung des Gegenbeispiels repräsentieren, muß der Algorithmus terminieren. Da für alle Variablen entsprechende Konstruktorterme eingesetzt wurden, sind in der verbleibenden Formel keine Variablen mehr vorhanden. Da dies das Abbruchkriterium für den Algorithmus ist, kann er nur dann weiterlaufen, wenn er mit dieser Belegung die Formel beweisen kann. Da diese Belegung aber das Gegenbeispiel repräsentiert, kann sie die Formel nicht erfüllen und der Algorithmus terminiert.

5.2.2 Die Beweisfortführung

Die Regeln, die für die Gegenbeispiel-Suche verwendet werden, sind Beweisregeln, d.h. sie sind eigentlich dafür ausgelegt, Beweise zu finden. Wird während einer Gegenbeispiel-Suche ein Beweisast geschlossen, so konnte durch Regelanwendungen ein Teil der Behauptung bewiesen werden. Da bei der Gegenbeispiel-Suche der Beweis für eine Formel φ fortgeführt wird, bedeutet dies, daß φ mit der bis dahin gefundenen Belegung erfüllbar ist. Deshalb kann in diesem Beweisast kein Gegenbeispiel mehr liegen und jede weitere Suche wäre unnütz. Deshalb kann dieser Ast von der Suche für ein Gegenbeispiel ausgeschlossen werden, und der Suchraum wird eingeschränkt. Diesen Effekt erhalten wir alleine dadurch, daß wir den Originalbeweis fortführen. Es ist nicht notwendig, spezielle Regeln für die Suchraumeinschränkung zu entwickeln.

In günstigen Fällen kann es geschehen, daß eine Formel φ sogar bewiesen werden kann. Da für die Gegenbeispiel-Suche keine Induktion benutzt wird, kann dies nur für solche Formeln geschehen, für deren Beweis keine Induktion notwendig ist. Die strukturelle Expansion zerlegt eine Variable in die Konstruktoren ihrer Sorte. Dieses Vorgehen wurde gewählt, da Funktionen meist rekursiv über Konstruktoren definiert sind. Durch die Zerlegung gelangen die entsprechenden Konstruktoren an die rekursive Position der Funktion und die Funktionsdefinition kann eingesetzt werden. Dadurch können natürlich nicht nur Gegenbeispiele gefunden werden, sondern auch Beweise, denn eine häufig angewandte Strategie bei der Beweissuche ist die Fallunterscheidung. Sie vereinfacht die Formel, indem sie die Formel aufteilt. Diese Aufteilung kann unter Umständen genügen, damit der Simplifier den Rest des Beweises automatisch lösen kann.

5.3 Abgrenzung zu anderen Verfahren

Für automatische Beweissysteme hat Protzen [Pro92] einen Mechanismus entwickelt, um falsche Annahmen, die von Theorembeweisern automatisch generiert werden, zu entdecken. Dabei berechnet er eine Substitution σ, mit der, für eine (falsche) Annahme φ, die negierte Aussage $\neg\ \varphi$ erfüllt wird.

Sein Vorgehen ist dabei, die Definition für eine Funktion f einzusetzen, um damit die Ausgangsformel $\neg\ \varphi$ zu vereinfachen. Bedingungen, die in $\neg\ \varphi$ für f gelten, müssen dabei an die Definition von f weiterpropagiert werden. Für quantorenfreie Formeln φ, die implizit $\forall$-abgeschlossen sind, ist sein Verfahren vollständig und korrekt.

Das von Protzen verwendete Kalkül verwendet eine Semantik, die nur ein Modell für eine Spezifikation besitzt, wohingegen in KIV eine Spezifikation mehrere Modelle besitzen kann. Dadurch kann es bei Protzen nicht vorkommen, daß in einem Modell der Spezifikation eine Formel φ gilt, in einem anderem aber $\neg\ \varphi$.

Ein weiterer Unterschied ist, daß in den Spezifikationen, die Protzen betrachtet, nur erzeugte Sorten auftreten können. Damit gibt es bei ihm keine Parametersorten. Die Funktionsdefinitionen sind algorithmisch und müssen vollständig sein, d.h. für jede Eingabe besitzt die Funktion einen definierten Ergebniswert. Außerdem muß die Funktion f für jede Eingabe x bei der Berechnung von $f(x)$ terminieren.

Mit diesen Einschränkungen kann Protzen die Probleme umgehen, die im vorgestellten Ansatz zu Anfragen an den Benutzer geführt haben. Es kann keine Unterspezifikation geben und, da es keine Parametersorten gibt, auch keine Aussagen über Parametersorten.

Da für eine Funktion meist mehrere Definitionsklauseln zur Verfügung stehen, muß man, um einen vollständigen Kalkül zu bekommen, alle Definitionsklauseln parallel betrachten. Dazu benutzt Protzen einen Oder-Baum und verlangt darauf eine faire Such-Strategie. Um die Suche nach der Lösung zu beschleunigen, markiert er Wege, die trivial unerfüllbar sind, als geschlossen.

Unser vorgestellter Ansatz zur Gegenbeispiel-Suche durch Beweisfortführung ist allgemeiner, wie der Ansatz von Protzen. Er setzt weder eine vollständige, algorithmische Funktionsdefinition noch die ausschließliche Verwendung von erzeugten Sorten voraus. Der allgemeinere Ansatz wird deshalb benötigt, weil in den Fallstudien, die mit KIV bearbeitet werden, selten die Bedingungen von Protzen erfüllt sind. In den Beispielen aus Kapitel 3 erfüllt nur das Beispiel *compiler* die Bedingungen von Protzens Ansatz. Die anderen Beispiele benutzen entweder Parametersorten oder die Spezifikation der natürlichen Zahlen, in der der Vorgänger von 0 nicht spezifiziert ist.

Die grundsätzlichen Ideen für die Gegenbeispiel-Suche sind in beiden Ansätzen gleich. Es wird eine Substitution (Belegung der Variablen) σ berechnet, mit der die negierte Aussage erfüllbar ist. Protzen geht bei der Berechnung von der negierten Aussage aus. Der vorgestellte Ansatz führt den Beweis der Originalaussage fort, und liefert zusätzlich zur Substitution σ noch eine Bedingung χ, die Einschränkungen über die Modelle der Spezifikation enthält. Dies ist bei Protzen, wegen des eingeschränkten Kalküls, nicht notwendig. Um die Substitution zu berechnen, werden die Aussagen vereinfacht. Protzen setzt die Definitionen einer Funktion ein und propagiert die Einschränkungen für die Funktion an deren Definitionen weiter. Im vorgestellten Ansatz wird versucht, die Bedingungen für die Anwendung der Definition durch strukturelle Expansion zur Verfügung zu stellen, und dann die Definition einzusetzen. Dies ist der wesentliche Unterschied der beiden Ansätze. Für beide Ansätze wird eine faire Such-Strategie benötigt und beide Ansätze bieten die Möglichkeit, unerfüllbare Formeln als geschlossen zu markieren.

Kapitel 6

Integration in KIV

Auf den theoretischen Grundlagen von Kapitel 5 aufbauend, wurde eine Gegenbeispiel-Suche in KIV integriert. Für die Integration ist es wichtig, daß sich die Gegenbeispiel-Suche in den gewohnten Arbeitsprozeß einbinden läßt. Die Suche muß aus einem aktuellen Beweisversuch heraus aufgerufen werden können, denn stößt der Benutzer bei einem Beweis auf eine Formel, für die er ein Gegenbeispiel suchen lassen möchte, wird er nicht den aktuellen Beweis abbrechen wollen. Eine Nebenrechnung sollte das Gegenbeispiel liefern. Zusätzlich muß noch die frühestmögliche Stelle im Originalbeweis berechnet werden, an der ein Gegenbeispiel gefunden werden kann.

Wie die Gegenbeispiel-Suche in den Ablauf eines Beweisversuchs integriert ist, sehen wir in Abschnitt 6.1. Danach gehen wir auf die Implementierung der einzelnen Schritte der Gegenbeispiel-Suche ein. Abschließend sehen wir noch ein paar Eckdaten, die den Aufwand für die Implementierung abschätzen sollen.

6.1 Integration der Gegenbeispiel-Suche in einen Beweisversuch

Die Gegenbeispiel-Suche wird aufgerufen, wenn sich der aktuelle Beweisversuch bei einer Formel φ befindet, von der der Benutzer vermutet, daß sie falsch ist. Der Benutzer ruft dann die Gegenbeispiel-Suche auf, die für das aktuelle Beweisziel ein Gegenbeispiel liefern soll. Wenn kein Gegenbeispiel gefunden werden kann, muß der Benutzer versuchen, den Originalbeweis zu Ende zu führen. Deshalb darf der Beweisbaum für den Beweisversuch nicht verändert werden. Um dies gewährleisten zu können, wird für die Gegenbeispiel-Suche ein Unterbeweis gestartet, für den eine neue Beweisumgebung geöffnet wird. Die Informationen des Originalbeweises bleiben dabei vollständig erhalten. Bei Beendigung des Unterbeweises gibt dieser seine ganzen Informationen an den Originalbeweis zurück. Der Originalbeweis kann die Informationen dann für den eigenen Beweis weiterverwenden.

Der Unterbeweis wird mit einer Formel φ gestartet. Mittels struktureller Expansion wird dann versucht, ein Gegenbeispiel zu finden. Ist ein Gegenbeispiel gefunden worden, wird der Unterbeweis verlassen und das Gegenbeispiel an den Originalbeweis gegeben. Damit die frühestmögliche Stelle, an der ein Fehler auftritt, im Beweisversuch gefunden wird, muß nun das Gegenbeispiel im Originalbeweis weiter zurückverfolgt werden.

Wie in Abschnitt 5.1.2 beschrieben, können bei der Rückverfolgung Beweisverpflichtungen entstehen, wenn nicht invertierbare Regeln verwendet wurden. Um diese Beweisverpflichtungen zu zeigen, kann erneut ein Unterbeweis aufgerufen werden. Wenn der Benutzer diese Beweisverpflichtung übergeht, d.h. er propagiert ein Gegenbeispiel über eine nicht invertierbare Regel, kann es geschehen, daß das berechnete Gegenbeispiel schlußendlich kein Gegenbeispiel für die Konklusio des Originalbeweises mehr ist. Um zu verifizieren, daß das berechnete Gegenbeispiel auch wirklich für die Konklusio gilt, kann erneut ein Beweisversuch gestartet werden.

Sehen wir uns im nächsten Abschnitt an, welche Aktionen in einem Unterbeweis durchgeführt werden können und danach, wie die Rückverfolgung abläuft. Wurde ein Gegenbeispiel gefunden, liegt ein Fehler vor, der behoben werden muß. Danach muß die (modifizierte) Aussage erneut bewiesen werden. Wie man dazu den aktuellen Beweisbaum wiederverwenden kann, sehen wir im letzten Teil dieses Abschnittes.

6.1.1 Der Unterbeweis für die Gegenbeispiel-Suche

Wenn man die Gegenbeispiel-Suche startet, gelangt man in einen Unterbeweis, der für eine Formel φ ein Gegenbeispiel sucht. Ist φ jedoch korrekt, kann keines gefunden werden. Da das Erfüllbarkeitsproblem unentscheidbar ist, kann man nicht wissen, ob φ korrekt ist, oder ob das entsprechende Gegenbeispiel noch nicht gefunden wurde. Deshalb kann die Gegenbeispiel-Suche unendlich lange laufen. Da KIV ein interaktives Beweissystem ist, bleibt es dem Benutzer überlassen, die Gegenbeispiel-Suche abzubrechen.

Damit der Benutzer die Gegenbeispiel-Suche nicht gerade dann abbricht, wenn die Suche im nächsten Schritt ein Gegenbeispiel finden würde, benötigt der Benutzer Informationen, in welchem Zustand sich die Suche gerade befindet. Dazu wird dem Benutzer die aktuelle Formel angezeigt, an der die Suche fortsetzt, damit er sehen kann, ob die bearbeiteten Formeln einfacher werden, dann ist eine Lösung in Sicht, oder nicht. Werden jedoch immer nur Instantiierungen gemacht, und die Formel kann nicht vereinfacht werden, wird wahrscheinlich keine Lösung mehr gefunden.

Da die Suche den ganzen Suchraum abdecken muß, kann es sein, daß sie gerade in einem Teil sucht, der keine Lösung enthält, obwohl eine Lösung existiert. Damit sich der Benutzer ein Bild von dem Gesamtzustand der Suche machen kann, wird zusätzlich der Suchbaum ausgegeben. Darin kann der Benutzer alle Zustände betrachten und sehen, ob es einen Zustand gibt, der zu einer Lösung führen könnte. Dadurch erhält der Benutzer die Möglichkeit, sich einen Überblick über den Stand der Suche zu verschaffen, damit er nicht zum falschen Zeitpunkt abbricht.

Hat die Suche eine erfüllbare Formel φ abgeleitet, kann der Pfad im Suchbaum zu dieser Formel Hinweise geben, warum die Ausgangsformel falsch ist. Sehen wir uns dazu ein Beispiel über sortierte Listen an. In der Spezifikation (siehe Abbildung 2.2) ist die Funktion *merge* falsch spezifiziert, da sie gleiche Elemente in der Liste zuläßt, wogegen *ord* nur gilt, wenn alle Elemente aufsteigend bezüglich der Ordnung $<$ angeordnet sind.

Beispiel 6.1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash}{\vdash\ ele < ele}\ \textit{simplifier}}{\vdash\ ele < ele \wedge ord(ele \oplus @)}\ \textit{ax-14}}{\vdash\ ord(ele \oplus ele \oplus @)}\ \textit{ax-15}}{\vdash\ ord(ele \oplus merge(ele \oplus @, @))}\ \textit{ax-08}}{\vdash\ ord(merge(ele \oplus @, ele \oplus @))}\ \textit{ax-09}}{\begin{array}{c}\vdots\\ \vdash\ ord(merge(x, y))\end{array}}$$

Als ein Gegenbeispiel würden wir $\mathcal{G} = (\textbf{true},\ (x, y) = (ele \oplus @, ele \oplus @))$ erhalten. Damit haben wir aber noch keinen Hinweis auf den Fehler, denn wir wissen nur, daß $ord(merge(x, y))$ in der Spezifikation nicht gültig ist. Wenn wir aber den Suchbaum der Gegenbeispiel-Suche betrachten, erkennen wir, daß in dessen Verlauf die Formel $ord(ele \oplus ele \oplus @)$ entsteht, die nicht gültig ist. Sie wird aus der Formel $ord(ele \oplus merge(ele \oplus @, @))$ abgeleitet, die wiederum aus $ord(merge(ele \oplus @, ele \oplus @))$ entsteht. Dieser Schritt, der durch die Anwendung des Axioms

$$\text{ax-09: } \neg\ ele_1 < ele_2 \rightarrow merge(ele_1 \oplus x, ele_2 \oplus y) = ele_2 \oplus merge(ele_1 \oplus x, y)$$

entstanden ist, kann nicht korrekt sein, da durch ihn gleiche Elemente in die Ergebnisliste gelangen können. Deshalb kann dieses Axiom nicht richtig sein, bzw. entspricht nicht der Absicht, eine <-geordnete Liste zu erhalten. Die korrekte Axiomatisierung wäre

ax-09.i: $ele_2 < ele_1 \rightarrow merge(ele_1 \oplus x, ele_2 \oplus y) = ele_2 \oplus merge(ele_1 \oplus x, y)$ und
ax-09.ii: $merge(ele \oplus x, ele \oplus y) = ele \oplus merge(x, y)$.

Ist eine Suche erfolgreich, d.h. es konnte eine Formel ψ abgeleitet werden, die keine freien, erzeugten Variablen mehr besitzt, bekommt der Benutzer die Meldung, wie er den Unterbeweis verlassen kann, um wieder in der Originalbeweis zurückzugelangen. Er hat nun die Möglichkeit, den Suchbaum zu inspizieren oder ein anderes Gegenbeispiel zu suchen, indem er im Suchbaum eine andere Formel ψ' sucht, die ein Gegenbeispiel hat. Das konkrete Gegenbeispiel wird deshalb erst beim Verlassen des Unterbeweises berechnet. Um sicher zu sein, daß die aktuelle Formel des Unterbeweises, von der das Gegenbeispiel berechnet wird, auch tatsächlich ein Gegenbeispiel hat, wird dies beim Verlassen nochmals nachgefragt. Ist die aktuelle Formel **false**, man befindet sich also an einer leeren Sequenz, so entfällt die Nachfrage, denn diese Formel hat immer ein Gegenbeispiel.

Im nächsten Abschnitt besprechen wir, wie das konkrete Gegenbeispiel berechnet und die Rückverfolgung gestartet wird.

6.1.2 Die Rückverfolgung

Beim Verlassen des Unterbeweises wird das konkrete Gegenbeispiel für die Konklusio des Suchbaumes, und damit für die Prämisse des Originalbeweises, von der aus die Gegenbeispiel-Suche aufgerufen wurde, berechnet. Da für die Suche des Gegenbeispiels nur Regeln verwendet wurden, die eine Rückberechnung zulassen, erhalten wir immer ein Gegenbeispiel für φ, wenn wir bei der Suche auf eine Formel ψ gestoßen sind, die ein Gegenbeispiel hat (siehe Abschnitt 5.1.2).

Die Berechnung des Gegenbeispiels für φ wird automatisch veranlaßt, wenn der Unterbeweis verlassen wird. Das berechnete Gegenbeispiel wird dem Benutzer präsentiert. Er kann dann entscheiden, ob das Gegenbeispiel im Originalbeweis noch weiter zurückverfolgt werden soll. Wird dies verlangt, versucht das System nach und nach ein Gegenbeispiel für Formeln auf dem Weg von der falschen Prämisse zur Konklusio des Beweisbaumes zu berechnen. Wenn in diesem Beweis auch Regeln verwendet wurden, die nicht invertierbar sind, können Beweisverpflichtungen oder Erfüllbarkeitsprobleme entstehen. Um diese zu lösen, werden erneut Unterbeweise gestartet.

Bevor die Unterbeweise gestartet werden, erfolgt eine Anfrage an den Benutzer, bei der er entscheiden kann, ob der Unterbeweis wirklich gestartet, die Rückverfolgung abgebrochen oder die Rückverfolgung ohne Beweis fortgesetzt wird. Wird erneut ein Unterbeweis für ein Erfüllbarkeitsproblem gestartet, gelangt man in den selben Modus wie bei der Gegenbeispiel-Suche. Es wird ein neues Gegenbeispiel geliefert, diesmal jedoch für einen Knoten mitten im Beweisbaum und nicht für eine Prämisse. Wenn eine Beweisverpflichtung gezeigt werden kann, bedeutet dies, daß das Gegenbeispiel von der Prämisse der Regeln auch für die Konklusio gilt. Ist der Benutzer der Meinung, daß der Beweisversuch bzw. Erfüllbarkeitsnachweis nicht gelingen wird oder er hat den Fehler schon entdeckt, kann er die Rückverfolgung abbrechen. Ist der Benutzer sicher, daß der Beweisversuch gelingen wird, kann er ihn auch übergehen. Dann wird einfach das bisherige Gegenbeispiel ohne Änderung einen Schritt weiter zurückgereicht.

Damit der Benutzer weiß, an welcher Stelle im Beweisbaum die Rückverfolgung sich gerade befindet, wird bei jeder Aktion, an der der Benutzer beteiligt ist, eine Markierung am Beweisbaum angebracht, die auf den aktuellen Knoten zeigt. Ist man schlußendlich an der Konklusio des Beweisbaumes angelangt, kann das berechnete Gegenbeispiel noch verifiziert werden. Dies ist deshalb notwendig, da Beweisverpflichtungen vom Benutzer übergangen werden können. Übergeht der Benutzer eine Beweisverpflichtung, die eine Anpassung des Gegenbeispiels ergeben würde, bzw. die überhaupt nicht beweisen werden kann, paßt das berechnete Gegenbeispiel nicht zur Konklusio.

Für die Verifikation wird erneut ein Unterbeweis gestartet. Ist $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ für die Konklusio $\Gamma \vdash \Delta$ ein Gegenbeispiel, so wird versucht $\chi \wedge \underline{x} = \underline{t} \rightarrow \neg\ (\Gamma \rightarrow \Delta\ )$ zu beweisen. Gelingt dies, ist die Konklusio nicht korrekt, es wurde also ein Gegenbeispiel gefunden. Gelingt dies nicht, so

wurde bei der Rückverfolgung ein Fehler gemacht, der durch Übergehen einer Beweisverpflichtung entstanden ist.

Im folgenden Abschnitt betrachten wir, wie das Vorgehen nach der Gegenbeispiel-Suche aussehen kann, wenn der Benutzer den gefundenen Fehler beseitigt hat.

6.1.3 Vorgehen nach der Gegenbeispiel-Suche

Liegt an der Prämisse φ kein Gegenbeispiel vor und die Gegenbeispiel-Suche hat nicht zufällig φ beweisen können, so muß der Benutzer den Beweis vollenden.

Ist die Prämisse, für die ein Gegenbeispiel gesucht wurde, falsch, gelangt die Rückverfolgung entweder an die Konklusio des Beweisbaumes oder sie muß abgebrochen werden. Ist man mit der Rückverfolgung an der Konklusio angelangt, ist die Behauptung des Lemmas oder die Spezifikation falsch. Mit Hilfe der Belegung des Gegenbeispiels muß man nun versuchen, den Fehler zu finden und zu beheben. Ist dies geschehen, so beginnt man den Beweis mit der korrigierten Version erneut. Da bei einem Fehler z.B. die Aussage zu wenige Vorbedingungen hat, bzw. in einem Programm nur ein bestimmter Programmzweig fehlerhaft ist, wird der neue Beweisversuch ähnlich ablaufen, wie der Beweis für die fehlerhafte Aussage. Damit man Beweisversuche für fehlerhafte Programme wiederverwenden kann, ist in KIV ein **Reuse** Mechanismus eingebaut [RS93], der die Unterschiede des alten (fehlerhaften) und des neuen Programms analysiert und mit Hilfe dieser Information versucht, Beweisäste des alten Beweises der neuen Aussage zuzuordnen und wiederzuverwenden.

Für prädikatenlogische Formeln gibt es einen **Replay** Mechanismus. Dieser versucht, die Beweisschritte des alten Beweises in der selben Reihenfolge nachzuspielen. Ein ähnlicher Mechanismus wie für Programmformeln wurde auch für prädikatenlogische Formeln entwickelt, ist aber noch nicht in KIV integriert [Bal97].

Mit diesen Mechanismen kann man nun versuchen, den alten Beweis für die neue Behauptung wiederzuverwenden, damit, für die im alten Beweis schon geschlossenen Äste, nicht nochmals der Beweisaufwand anfällt.

Stoppt die Rückverfolgung jedoch mitten im Beweisbaum, liegt meist ein Fehler in der Beweisführung vor, den der Benutzer durch eine andere Beweisentscheidung an dieser Stelle beheben kann. Dann muß der Benutzer versuchen, den Beweis fortzuführen.

6.2 Implementierung in KIV

Die Erweiterung für die Konstruktion von Gegenbeispielen ist, wie das KIV-System, in PPL geschrieben. PPL (proof programming language) ist eine typisierte, funktionale Sprache, die in Lisp implementiert ist. Die Syntax ist weitgehend aus Lisp übernommen worden.

Um Beweise zu führen, werden **Regeln** angewendet, die die Struktur einer Formel ändern. Um oft wiederkehrende Regelanwendungen zu automatisieren, gibt es in KIV **Heuristiken**, die bestimmte Regelanwendungen selbstständig durchführen. Damit die Heuristiken arbeiten können, benötigen sie Informationen über den Beweis. Diese können in allgemeine Informationen, die während des ganzen Beweises gleich bleiben, und Informationen, die an einen Beweisknoten gebunden sind, eingeteilt werden. Allgemeine Informationen sind z.B die Axiome einer Spezifikation. Die Knoten-Informationen können für jeden Beweisknoten verschieden sein. Dazu gehören z.B die Anzahl bestimmter Regelanwendungen oder der Pfad zum aktuellen Knoten.

Die Gegenbeispiel-Suche ist vollständig über Heuristiken gesteuert. Welche Heuristiken dazu verwendet werden und wie diese arbeiten, sehen wir im nächsten Abschnitt. Danach wird die Rückverfolgung, und damit die Berechnung des konkreten Gegenbeispiels, beschrieben. Schließlich betrachten wir noch, welche Suchstrategien tatsächlich für die Suche verwendet werden.

6.2.1 Die Beweisfortführung in KIV

Wie oben erwähnt, ist die Beweisfortführung durch Heuristiken implementiert. Um einzelne Heuristiken überschaubar zu halten, kann in KIV ein **Heuristik-Satz** zusammengestellt werden. Dies

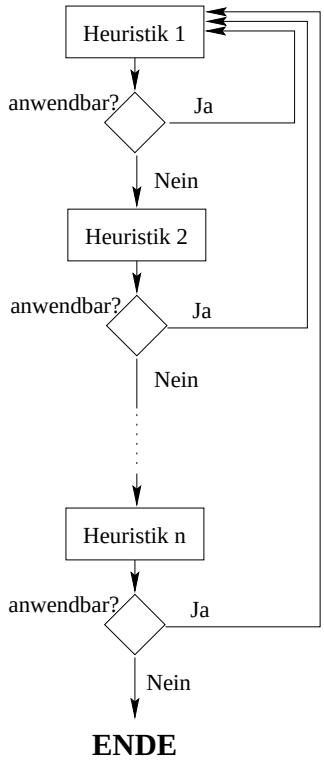

Abbildung 6.1: Reihenfolge der Heuristikauswahl

ist eine Liste von Heuristiken, die nacheinander abgearbeitet werden. Die Reihenfolge der Abarbeitung verdeutlicht die Abbildung 6.1. Die Liste wird vom Anfang an abgearbeitet. Sobald eine Heuristik erfolgreich angewendet werden konnte, wird wieder an den Anfang der Liste gesprungen, und die Abarbeitung beginnt von neuem. Die Bearbeitung des Heuristik-Satzes ist beendet, wenn keine der Heuristiken mehr anwendbar ist.

Bei einem Beweis können Formeln mit und ohne Programmformeln entstehen. Wenn in einer Formel keine Programmformeln enthalten sind, benötigen wir auch keine Heuristiken, die Programmformeln bearbeiten. Deshalb werden für die Gegenbeispiel-Suche zwei Heuristik-Sätze angeboten. Der PL-Heuristik-Satz wird verwendet, wenn die Formel keine Programmformeln enthält. Ansonsten wird der DL-Heuristik-Satz für die Gegenbeispiel-Suche verwendet.

Heuristiken für PL-Formeln

Die Beweisfortführung für PL-Formeln wird mit dem folgenden Heuristik-Satz durchgeführt.

- *select goal*-Heuristik
- *strong simplifier*-Heuristik
- *Quantifier closing*-Heuristik
- *elimination*-Heuristik
- *axiom cut*-Heuristik
- *rewrite*-Heuristik
- *pl case distinction*-Heuristik

- *give value*-Heuristik
- *constructor cut*-Heuristik
- *quantifier elimination*-Heuristik

Nun folgen kurze Erläuterungen, welche Aufgaben die einzelnen Heuristiken durchführen.

Select Goal Die *select goal*-Heuristik sorgt dafür, daß im nächsten Schritt der „richtige" Knoten des Beweisbaumes ausgewählt wird. Da bei der Gegenbeispiel-Suche die einfachen Formeln zuerst ausgesucht werden, sollten alle Prämissen des Beweisbaumes vereinfacht werden. Die *select goal*-Heuristik sorgt dafür, daß dies vor allen anderen Aktionen geschieht. Dazu wechselt sie zu noch nicht vereinfachten Prämissen, die die folgende *strong simplifier*-Heuristik dann vereinfacht.

Für die Gegenbeispiel-Suche wollen wir eine A^*-Suche implementieren. Deshalb muß die Heuristik die Prämisse mit dem geringsten Gewicht auswählen und zu ihr wechseln. Dazu müssen alle Prämissen bewertet sein. Ist eine Prämisse noch nicht bewertet, wechselt die *select goal*-Heuristik zu dieser Prämisse und die *goal value*-Heuristik bewertet sie. Sind alle Prämissen bewertet, wechselt die *select goal*-Heuristik zu der Prämisse, mit dem geringsten Gewicht, welche die *constructor cut*-Heuristik dann expandiert.

Strong Simplifier Die *strong simplifier*-Heuristik wendet die *simplifier*-Regel an, die prädikatenlogische Formeln vereinfacht. Dazu wendet sie Simplifier-Lemmas an, die ihr der Benutzer zur Verfügung stellt. Je mehr Lemmas der Benutzer zur Verfügung stellt, desto leistungsfähiger wird der Simplifier. Um ein Lemma als Simplifier-Lemma benutzen zu können, muß es die folgende Form haben:

$$\Gamma \vdash \psi_1 \wedge \ldots \wedge \psi_n \rightarrow \varphi$$

Die Formel φ kann $\phi = \chi$, $\phi \leftrightarrow \chi$, $\phi \rightarrow \chi$ oder eine atomare Formel sein. Eine Simplifier-Regel kann auf eine Sequenz angewendet werden, wenn eine Substitution σ existiert, so daß die Vorbedingung $\sigma(\Gamma)$ gezeigt werden kann und alle $\sigma(\psi_i)$ in der Sequenz vorkommen. Dann werden die Vorkommen von $\sigma(\phi)$ durch $\sigma(\chi)$ ersetzt. Wenn φ atomar ist, wird die Formel entfernt bzw., wenn $\neg\ \varphi$ in der Sequenz vorhanden ist, der Knoten geschlossen.

Sind die Simplifier-Lemmas bewiesen, ist ein Simplifier-Schritt invertierbar, wenn nicht eine Implikationsregel, also eine Regel, in der φ die Form $\phi \rightarrow \chi$ hat, angewendet wird, denn sie kann aus einer beweisbaren Aussage eine unbeweisbare machen [Lö94].

Beispiel 6.2
Sei $\vdash x + y < z \rightarrow y < z$ *ein Simplifier-Lemma. Die Aussage ist korrekt und kann bewiesen werden. Die Sequenz* $1 + y < z,\ y + 1 = z \vdash$ *ist auch beweisbar. Mit der definierten Simplifier-Regel erhalten wir daraus die Sequenz* $y < z,\ y + 1 = z \vdash$ *, die mit* $\mathcal{G} = (\mathbf{true},\ (y, z) = (0, 1))$ *ein Gegenbeispiel besitzt.*

Deshalb werden für die Suche von Gegenbeispielen alle Implikationsregeln aus dem ursprünglichen Simplifier-Satz entfernt. Daraus resultiert der **strong Simplifier**-Satz. Mit diesem Regelsatz ruft die *strong simplifier*-Heuristik den Simplifier auf. Damit ist die Anwendung der *simplifier*-Regel invertierbar und kann für die Gegenbeispiel-Suche eingesetzt werden.

Quantifier Closing Diese Heuristik versucht, Instanzen für Variablen zu finden, die durch einen Quantor gebunden sind. Wird eine Instanz gefunden, mit der die Formel bewiesen werden kann, so wird sie angewendet und der Knoten geschlossen. Damit wird verhindert, daß die Gegenbeispiel-Suche in einem beweisbaren Zweig ein Gegenbeispiel sucht und die Suche wird effizienter.

Elimination Die *elimination*-Heuristik versucht, auf eine Formel in der Sequenz die *elimination*-Regel (siehe B.2) anzuwenden. Dies ist eine spezielle Simplifier-Regel, die „bösartige" Funktionen durch „gutartige" Funktionen ersetzt. Gutartige Funktionen sind beispielsweise $+1$, $+$, $\oplus$ und bösartige -1, $-$, *.first*, *.rest*. Der Benutzer legt, wie für den Simplifier, Elimination-Lemmas fest und entscheidet damit, welche Funktionen er für gutartig bzw. bösartig hält. Meist werden Selektoren durch entsprechende Konstruktoren ersetzt. Die allgemeine Form eines Elimination-Lemmas ist

$$\Gamma \vdash \varphi \rightarrow (x_1 = t_1 \wedge \ldots \wedge x_n = t_n \leftrightarrow \exists\ \underline{y}.v = t \wedge \psi).$$

Die *elimination*-Heuristik versucht eine Substitution σ zu finden, so daß $\sigma(t_i)$ in der aktuellen Sequenz vorkommt und $\sigma(v)$ eine Variable ist. Wenn $\sigma(\Gamma)$ beweisbar ist und $\sigma(\varphi)$ in der Sequenz vorkommt, dann wird in der Sequenz $\sigma(t_i)$ durch $\sigma(x_i)$ und $\sigma(v)$ durch $\sigma(t)$ ersetzt und die Formel $\sigma(\psi)$ addiert.

Beispiel 6.3
Sei $x \neq @ \vdash (a = x.first \wedge y = x.rest \leftrightarrow x = a \oplus x)$ *ein Elimination-Lemma und* $x \neq @ \vdash \varphi(x.first,\ x.rest,\ x)$ *eine Sequenz. Die* elimination*-Regel wandelt die Sequenz in* $a \oplus y \neq @ \vdash \varphi(a,\ y,\ a \oplus y)$ *um.*

Wenn die entsprechenden Elimination-Lemmas vorhanden sind, sorgt die Heuristik dafür, daß Konstruktoren erzeugt werden. Dadurch entsteht, wie bei einem Expansionsschritt, die Möglichkeit, daß sich rekursive Funktionsdefinitionen einsetzten lassen und die Formel vereinfacht wird.

Axiom Cut Die *axiom cut*-Heuristik führt über die *cut*-Regel (siehe B.2) neue Bedingungen in eine Formel ein, um Regeln anwendbar zu machen. Die *rewrite*-Heuristik ersetzt in einer Sequenz alle Vorkommen von $\sigma(\rho)$ durch $\sigma(\tau)$, falls ein Axiom der Form $\varphi \rightarrow \rho = \tau$ und eine Substitution σ für die freien Variablen des Axioms existiert. Damit die *rewrite*-Heuristik anwendbar wird, muß die Bedingung $\sigma(\varphi)$ in der aktuellen Sequenz vorhanden sein. Die *axiom cut*-Heuristik führt diese Bedingung ein.

Beispiel 6.4
Ist das Axiom $\vdash ele = ele_0 \rightarrow del(ele, ele_0 \oplus x) = x$ *vorhanden, wobei die Sorte von ele eine Parametersorte ist, kann es die* rewrite*-Heuristik nicht auf eine Sequenz der Form* $\vdash @ = del(ele_1, ele_2 \oplus @)$ *anwenden, da über die Gleichheit der Parametervariablen* ele_1 *und* ele_2 *keine Aussage gemacht werden kann. Durch die* cut*-Regel kann, mit der* cut*-Formel* $ele_1 = ele_2$ *die folgende Ableitung erzeugt werden:*

$$\dfrac{\dfrac{\dfrac{ele_1 = ele_2 \vdash}{ele_1 = ele_2 \vdash @ = @}}{ele_1 = ele_2 \vdash @ = del(ele_1, ele_2 \oplus @)} \quad \dfrac{\vdots}{ele_1 \neq ele_2 \vdash @ = del(ele_1, ele_2 \oplus @)}}{\vdash @ = del(ele_1, ele_2 \oplus @)}$$

Die *axiom cut*-Heuristik soll für ein Axiom der Form $\varphi \rightarrow \rho = \tau$ die Bedingung $\sigma(\varphi)$ einführen, wenn eine Substitution σ existiert, so daß in der aktuellen Sequenz $\sigma(\rho)$ vorhanden ist. Die Heuristik versucht nun für die aktuelle Sequenz eine Substitution und ein Axiom zu finden, um die Bedingung $\sigma(\rho)$ einzuführen. Damit die gleiche Bedingung nicht immer wieder angewendet wird, speichert sich die Heuristik das Axiom und die Substitution mit der sie angewendet wurde. Dies ist deshalb notwendig, da in dem Fall, in dem die Vorbedingung nicht gilt, das Axiom und die Substitution immer wieder paßt. Im Beispiel sieht man dies im Fall $e_1 \neq e_2 \vdash @ = del(e_1, e_2 \oplus @)$. Da die Vorbedingung nicht erfüllt ist, kann keine Simplifikation stattfinden und die Sequenz bleibt durch die *cut*-Regel (bis auf $e_1 \neq e_2$) unverändert. Auf diese Sequenz paßt das Axiom und die Substitution natürlich wieder und die Regel könnte immer wieder angewendet werden, ohne einen Fortschritt zu bringen.

Der Benutzer kann in KIV auch selbst Lemmas benennen, die die *cut*-Regel bei ihrer Anwendung berücksichtigt.

Rewrite Der Simplifier ist dazu gedacht, Formeln zu vereinfachen und sie dem Benutzer verständlicher zu machen. Deshalb definiert der Benutzer, welche Lemmas der Simplifier anwenden soll. Für die Gegenbeispiel-Suche sind diese Regeln nicht immer ausreichend, denn es sollten alle Axiome angewendet werden können, da sie die Theorie der Spezifikation bilden. Meist sind aber nicht alle Axiome als Simplifier-Lemmas eingetragen, da sie zu unerwünschten Fallunterscheidungen in Beweisen führen können, und damit der Beweisaufwand wächst.

Beispiel 6.5
Sei $ord(ele_1 \oplus ele_2 \oplus x) \leftrightarrow ele_1 < ele_2 \wedge ord(ele_2 \oplus x)$ *als Simplifier-Lemma definiert. Wenn ord im Sukzedent einer Sequenz vorkommt, würde die Konjunktion zu einer Fallunterscheidung führen. Wenn aber ein Gegenbeispiel für* $\vdash ord(ele_1 \oplus ele_2 \oplus x)$ *gesucht wird, liefert genau diese Regel ein Gegenbeispiel* $\mathcal{G}(\neg\ ele_1 < ele_2,\ \mathbf{true})$*, indem sie die widersprüchliche Prämisse* $\vdash ele_1 < ele_2$ *erzeugt.*

Es ist aber nicht möglich, einfach alle Axiome als Simplifier-Lemmas einzutragen. Dadurch könnte ein Zyklus während der Simplifieranwendung entstehen. Dies geschieht dann, wenn der Simplifier-Satz Lemmas enthält, die Sequenzen immer wieder so umformen, daß dieselben Simplifier-Lemmas immer wieder anwendbar werden.

Beispiel 6.6
Sei das Lemma lem-01: $\vdash ele_1 < ele_2 \rightarrow ord(ele_2 \oplus x) \leftrightarrow ord(ele_1 \oplus ele_2 \oplus x)$ *als Simplifier-Lemma eingetragen, so würde durch Hinzunahme des Axioms ax-15:* $ord(ele_1 \oplus ele_2 \oplus x) \leftrightarrow ele_1 < ele_2 \wedge ord(ele_2 \oplus x)$ *als Simplifier-Lemma ein Zyklus entstehen, denn der Simplifier könnte folgende Ableitung erstellen:*

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash ele_1 < ele_2 \wedge ord(ele_2 \oplus x)}}{\vdash ord(ele_1 \oplus ele_2 \oplus x)}\ \textit{ax-15}}{\vdash ele_1 < ele_2 \wedge ord(ele_2 \oplus x)}\ \textit{lem-01}}{\vdash ord(ele_1 \oplus ele_2 \oplus x)}\ \textit{ax-15}$$

Da der aktuelle Simplifier in KIV keine Zyklen erkennt, werden die Axiome, die zusätzlich zu den Simplifier-Lemmas aufgenommen werden, als *rewrite*-Lemmata angeboten. Die *rewrite*-Heuristik wendet mit der *insert rewrite lemma*-Regel (siehe B.2) diese Lemmas an. Dabei sorgt die Heuristik dafür, daß ein Rewrite-Lemma mit derselben Substitution nur einmal angewendet wird. Deshalb können keine Zyklen entstehen.

Die *rewrite*-Heuristik ist vor allem bei nicht frei erzeugten Sorten von Bedeutung, um die Extensionalitäts-Axiome anwenden zu können. Extensionalitäts-Axiome haben die Form

$$a_1 = a2 \leftrightarrow \varphi_1 \wedge \ldots \wedge \varphi_n$$

und definiert die Gleichheit von Konstruktortermen einer Sorte. Sie werden nicht als Simplifier-Lemmas eingetragen, da sie im Sukzedenten zu Fallunterscheidungen führen würden. Durch die *constructor cut*-Heuristik (siehe unten) gelangen eben solche Gleichungen über erzeugte Sorten in den Sukzedenten. Die Gegenbeispiel-Suche muß nun zeigen, daß es verschiedene Elemente a_1 und a_2 gibt. Dazu setzt sie das Extensionalitäts-Axiom ein und versucht, eines der Konjunktions-Glieder zu widerlegen.

Pl Case Distinction Durch Anwendung der *rewrite*-Regel oder der *simplifier*-Regel können im Antezedenten einer Sequenz Disjunktionen bzw. im Sukzedenten Konjunktionen entstehen. Eine Sequenz der Form $\varphi \vee \psi \vdash \chi$ bedeutet, daß χ sowohl aus φ als auch aus ψ folgt. Für die Gegenbeispiel-Suche genügt es, in einem der Fälle ein Gegenbeispiel zu finden. Deshalb werden Fallunterscheidungen über prädikatenlogische Formeln eingeführt.

Die *pl case distinction*-Heuristik führt für jede Disjunktion im Antezedenten bzw. Konjunktion im Sukzedenten die Fallunterscheidungen ein. Dadurch wird der Suchraum wie bei der strukturellen Expansion aufgeteilt. Der Unterschied ist, daß die strukturelle Expansion den Suchraum anhand der Struktur der Konstruktorterme aufteilt, wohingegen die *pl case distinction*-Heuristik den Suchraum nach der Struktur der Sequenz, für die das Gegenbeispiel gesucht wird, aufteilt.

Constructor Cut Die *constructor cut*-Heuristik wählt sich aus den freien, erzeugten Variablen der Sequenz eine aus und führt für diese Variable die strukturelle Expansion durch. Mit welchen Kriterien die Variable für die Expansion ausgewählt wird, sehen wir in Abschnitt 6.2.3. Für die Expansion einer Variablen von frei erzeugten Sorten wird die Regel

$$\frac{x = c,\ \Gamma \vdash \Delta \qquad x = f(x'),\ \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \text{constructor cut (1)}$$

angewendet, wobei $x \in \mathit{free}(\Gamma) \cup \mathit{free}(\Delta)$, $s = \mathit{sort}(x)$ und s **freely generated by** c, f. Die Variable x' ist eine neue Variable der Sorte s.

Für nicht frei erzeugte Variablen wird zusätzlich dafür gesorgt, daß sich der Wert mit jeder Expansion ändern muß. Dies ist notwendig, da bei erzeugten Sorten verschiedene Konstruktorterme den gleichen Wert repräsentieren können. Für Variablen erzeugter Sorten sieht die Expansions-Regel folgendermaßen aus:

$$\frac{x = c, \Gamma \vdash \Delta \qquad x = f(x'), x \neq x', \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \text{constructor cut (2)}$$

Damit wird erzwungen, daß sich der Wert des Konstruktorterms ändert. Damit die Regeln auch in Beweisen verwendet werden können, müssen sie korrekt sein. Für die Rückverfolgung zeigen wir, daß sie auch stark invertierbar sind.

Satz 6.1 (Korrektheit der Expansionsregeln)
Die constructor cut*-Regeln sind korrekt.*

Beweis 6.1 (Korrektheit der Expansionsregeln)
Wir zeigen die Korrektheit der Regeln, indem wir eine Ableitung, die das gewünschte Resultat ergibt, mit existierenden Regeln angeben. Dazu benötigen wir eine Hilfsableitung. Diese Ableitung zeigt, daß anstatt einer Substitution auch eine entsprechende Gleichung in den Antezedenten addiert werden kann.

*Ableitung**

$$\dfrac{\dfrac{\dfrac{\dfrac{x = f(x'), \Gamma \vdash \Delta}{x = f(x'), \Gamma_x^x \vdash \Delta_x^x}}{x = f(x'), (\Gamma_x^{f(x')})^x_{f(x')} \vdash (\Delta_x^{f(x')})^x_{f(x')}}\ \mathit{ins.\ eq}}{\dfrac{x = f(x'), \Gamma_x^{f(x')} \vdash \Delta_x^{f(x')}}{\exists\ x.\ x = f(x'), \Gamma_x^{f(x')} \vdash \Delta_x^{f(x')}}\ \mathit{exists\ left}} \qquad \dfrac{\dfrac{}{\Gamma_x^{f(x')} \vdash \Delta_x^{f(x')}, \exists\ x.\ x = f(x'), f(x') = f(x')}}{\Gamma_x^{f(x')} \vdash \Delta_x^{f(x')}, \exists\ x.\ x = f(x')}$$

$$\Gamma_x^{f(x')} \vdash \Delta_x^{f(x')}\ \mathit{cut}$$

Mit Hilfe dieser Ableitung läßt sich nun der Beweis für die constructor cut (1)*-Regel führen. Wir betrachten nur den Fall einer Konstruktorfunktion f. Der Fall für die Konstruktorkonstante c folgt analog.*

$$\dfrac{\vdots\ \vdash \Gamma_x^c \to \Delta_x^c \qquad \dfrac{\dfrac{\dfrac{x = f(x'), \Gamma \vdash \Delta}{\Gamma_x^{f(x')} \vdash \Delta_x^{f(x')}}\ \mathit{Ableitung^*}}{\vdash \Gamma_x^{f(x')} \to \Delta_x^{f(x')}}\ \mathit{impL\text{-}r}}{\forall\ \underline{x}.\ \Gamma_x^{x'} \to \Delta_x^{x'} \vdash \Gamma_x^{f(x')} \to \Delta_x^{f(x')}}\ \mathit{weakening}}{\Gamma \vdash \Delta}\ \mathit{structural\ induction}$$

Für die *constructor cut (2)*-Regel läßt sich auf ähnliche Weise eine Ableitung aus den bestehenden Regeln ableiten. zusätzlich muß die Ungleichung $x \neq x'$ eingefügt werden.

$$\dfrac{\begin{array}{c}\vdots \\ \vdash \Gamma_x^c \rightarrow \Delta_x^c\end{array} \qquad \dfrac{x' = f(x') \ldots \qquad \dfrac{\dfrac{\dfrac{\dfrac{x = f(x'), x' \neq x, \Gamma \vdash \Delta}{x = f(x'), x' \neq f(x'), \Gamma \vdash \Delta}\ \text{insert eq}}{x = f(x'), x' \neq f(x'), \Gamma_x^x \vdash \Delta_x^x}\ \text{Ableitung 2}}{x' \neq f(x'), \vdash \Gamma_x^{f(x')} \rightarrow \Delta_x^{f(x')}}\ \text{weakening}}{x' \neq f(x'), \forall\ \underline{x}.\ \Gamma_x^{x'} \rightarrow \Delta_x^{x'} \vdash \Gamma_x^{f(x')} \rightarrow \Delta_x^{f(x')}}\ \text{cut}}{\forall\ \underline{x}.\ \Gamma_x^{x'} \rightarrow \Delta_x^{x'} \vdash \Gamma_x^{f(x')} \rightarrow \Delta_x^{f(x')}}}{\Gamma \vdash \Delta}\ \text{structural induction}$$

Die linke Seite der *cut*-Regel schließt die folgende Ableitung.

$$\dfrac{\dfrac{\dfrac{}{x' = f(x'), \Gamma_x^{x'} \rightarrow \Delta_x^{x'}, \forall\ \underline{x}.\ \Gamma_x^{x'} \rightarrow \Delta_x^{x'} \vdash \Gamma_x^{x'} \rightarrow \Delta_x^{x'}}\ \text{all right}}{x' = f(x'), \forall\ \underline{x}.\ \Gamma_x^{x'} \rightarrow \Delta_x^{x'} \vdash \Gamma_x^{x'} \rightarrow \Delta_x^{x'}}\ \text{insert equation}}{x' = f(x'), \forall\ \underline{x}.\ \Gamma_x^{x'} \rightarrow \Delta_x^{x'} \vdash \Gamma_x^{f(x')} \rightarrow \Delta_x^{f(x')}}$$

Wir zeigen jetzt noch die Invertierbarkeit der Regeln.

Satz 6.2 (Starke Invertierbarkeit der *constructor cut*-Regeln)
Die constructor cut*-Regeln sind stark invertierbar.*

Beweis 6.2
Fassen wir die Regeln als Sequenzen auf, ist

$$(\Gamma \rightarrow \Delta) \rightarrow (x = c \wedge \Gamma \rightarrow \Delta) \wedge (x = f(x') \wedge \Gamma \rightarrow \Delta)\ \textit{constructor cut 1}$$

bzw.

$$(\Gamma \rightarrow \Delta) \rightarrow (x = c \wedge \Gamma \rightarrow \Delta) \wedge (x = f(x') \wedge x \neq x' \wedge \Gamma \rightarrow \Delta)\ \textit{constructor cut 2}$$

zu zeigen. Dies ist trivial gültig, da die Formeln auf der rechten Seite der Implikation mehr Voraussetzungen haben, als die Formeln im Vordersatz der Implikation.
□

Damit sind die *constructor cut*-Regeln korrekt und (stark) invertierbar und können sowohl für die Beweise, als auch für die Gegenbeispiel-Suche benutzt werden. In KIV selbst wird nur eine *constructor cut*-Regel angeboten. Sie bietet dem Benutzer alle freien, erzeugten Variablen der Sequenz zur Auswahl an. Für die gewählte Variable entscheidet die Regel dann, ob die Sorte der Variablen erzeugt oder frei erzeugt ist und welche der Variante der *constructor cut*-Regel angewendet werden muß. Die *constructor cut*-Heuristik wählt selbstständig eine Variable aus der Sequenz aus und wendet die *constructor cut*-Regel an.

Quantifier Elimination Die *quantifier weakening*-Heuristik versucht, eine quantifizierte Formel aus der Sequenz zu entfernen, die redundante Information enthält, d.h. die quantifizierte Formel kann aus dem Rest der Sequenz abgeleitet werden.

Für den Benutzer ist es schwierig zu entscheiden, ob quantifizierte Formeln widersprüchlich sind. Da in dem vorgestellten Verfahren der Gegenbeispiel-Suche der Benutzer die Entscheidung über Formeln mit Parametervariablen treffen soll, ob sie widersprüchlich sind oder nicht, sollten ihm möglichst einfache Formeln vorgelegt werden. Deshalb wird versucht, noch vorhandene, quantifizierte Formeln aus der Sequenz zu entfernen. Dies übernimmt die *quantifier weakening*-Heuristik. Sie kommt erst dann an die Reihe, wenn die *constructor cut*-Heuristik keine Variable

mehr findet, über die sie expandieren kann. Dies ist dann der Fall, wenn keine erzeugten Variablen mehr in der Sequenz vorkommen.

Die *quantifier weakening*-Heuristik benutzt die *weakening*-Regel (siehe B.2), um quantifizierte Formeln zu entfernen. Da während der Gegenbeispiel-Suche nur invertierbare Regeln verwendet werden dürfen, muß die Heuristik dafür sorgen, daß nur solche Formeln entfernt werden, die aus der restlichen Sequenz abgeleitet werden können. Damit die quantifizierte Formel $\forall \underline{x}.\ \varphi$ aus der Sequenz $\forall \underline{x}.\ \varphi, \Gamma \vdash \Delta$ entfernt werden kann, muß $\vdash (\Gamma \to \Delta) \to \forall \underline{x}.\ \varphi$ gelten (siehe 5.1.4). Um diese Aussage zu zeigen, könnte man erneut einen Unterbeweis starten und mit verschiedenen Heuristiken versuchen, sie zu beweisen. Für die Implementierung wurde aber ein einfacherer Weg gewählt, für den der Aufwand für einen Unterbeweis entfällt.

Der Simplifier kann intern in KIV als Funktion aufgerufen werden, dem die zu vereinfachende Sequenz als Argument übergeben wird. Als Ergebnis wird ein Beweisbaum geliefert, der als Konklusio die übergebene Sequenz besitzt und als Prämisse die vereinfachte Ergebnissequenz. Konnte der Simplifier die Konklusio zu **true** vereinfachen, ist die Prämisse geschlossen.

Um nun $\vdash (\Gamma \to \Delta) \to \forall \underline{x}.\ \varphi$ zu beweisen, wird der Simplifier mit dieser Sequenz aufgerufen. Wird als Ergebnis ein Beweisbaum ohne offene Prämisse geliefert, ist die Aussage gültig. Ist die Prämisse nicht geschlossen, so kann die Aussage zwar gültig sein, doch der Simplifier kann sie nicht zeigen. Deshalb wird die Formel $\forall \underline{x}.\ \varphi$ nicht aus der Ausgangssequenz entfernt.

Durch dieses Vorgehen können nicht alle Formeln, die man entfernen könnte, erkannt werden, aber es ist wesentlich effizienter, als einen Unterbeweis zu starten. Die zweite Variante, Quantoren zu entfernen, die auch im Abschnitt 5.1.4 vorgestellt wurde, ist nicht in KIV integriert. Dazu müßte aus dem Simplifier die *insert-eq-var-drop*-Regel entfernt werden, da sie nicht stark invertierbar ist. Da diese Regel fest in den Simplifier integriert ist und nicht, wie die Implikationsregeln, über einen Simplifier-Satz addiert wird, ist dies nicht einfach möglich.

Heuristiken für DL-Formeln

Bisher wurden immer nur prädikatenlogische Formeln betrachtet. KIV ist ein System, mit dem man auch die Korrektheit von Programmen zeigen kann. Dazu werden Programme als Formeln der Dynamischen Logik aufgefaßt. Die Regeln der Dynamischen Logik überführen die DL-Formeln in PL-Formeln. Über die PL-Formel werde die Beweisäste dann geschlossen. Um Gegenbeispiele auch für Sequenzen mit DL-Formeln erzeugen zu können, müssen entsprechende Heuristiken DL-Regeln anwenden. Aus den entstehenden PL-Formeln versuchen die oben genannten Heuristiken dann, das Gegenbeispiel zu konstruieren. Dabei entstehen Sequenzen, die sowohl DL- als auch PL-Formeln enthalten. Die Heuristik-Liste für Sequenzen mit DL-Formeln enthält deshalb neben den Heuristiken für DL-Regeln, auch die oben beschriebenen Heuristiken für PL-Regeln. Die

- *select goal*-Heuristik,
- *symbolic execution*-Heuristik,
- *contract and execute,*-Heuristik
- *module specific*-Heuristik,
- *split*-Heuristik,
- *unfold*-Heuristik,
- *calls nonrecursive*-Heuristik,
- *calls concrete*-Heuristik,
- *strong simplifier*-Heuristik,
- *conditional right split*-Heuristik,
- *conditional left split*-Heuristik,

- *Quantifier closing*-Heuristik,
- *elimination*-Heuristik,
- *axiom cut*-Heuristik,
- *rewrite*-Heuristik,
- *pl case distinction*-Heuristik,
- *dl case distinction*-Heuristik,
- *give value*-Heuristik,
- und die *quantifier elimination*-Heuristik

werden zur Gegenbeispiel-Suche für DL-Formel verwendet. Nun beschreiben wir noch die Heuristiken, die Regeln auf DL-Formeln anwenden.

Symbolic Execution Die *symbolic execution*-Heuristik führt Programme aus, indem sie DL-Regeln anwendet. Sie benutzt dabei aber nur solche Regeln, die nicht zu Verzweigungen oder zu Endlosschleifen im Beweis führen können. Im einzelnen sind dies folgende Regeln (siehe B.1):

> *abort right, abort left, skip right, skip left, assign right, assign left, vardecls right, vardecls left, if positive right, if negative right, if positive left, if negative left*

Contract and Execute Die *contract and execute*-Heuristik versucht, die *execute call*-Regel, die *contract call left*-Regel und die *contract call right*-Regel auf die Sequenz anzuwenden (siehe B.1).

Module Specific In Programmbeweisen gibt es immer wieder ähnliche Situationen, in denen immer dieselbe Regel angewendet werden soll. Um diesen Vorgang zu automatisieren, kann der Benutzer einem Muster eine Regel zuordnen. Das Muster repräsentiert die Sequenzen, bei denen die zugeordnete Regel angewendet wird. Die *module specific*-Heuristik versucht, für eine Sequenz ein passendes Muster zu finden und die entsprechende Regel auf die Sequenz anzuwenden. Ähnlich wie die *strong simplifier*-Heuristik, die PL-Formeln vereinfacht, werden durch die *module specific*-Heuristik DL-Formeln vereinfacht.

split Die *split*-Heuristik wendet die *split left*-Regel (siehe B.1) auf den Antezedenten einer Sequenz an.

Calls Nonrecursive Die *calls nonrecursive*-Heuristik führt Prozeduren aus, die nicht rekursiv sind.

Calls Concrete Die *calls concrete*-Heuristik führt Prozeduren aus, deren Eingabeparameter konkrete Werte haben. Ausnahmen bilden Eingabeparameter, deren Sorten Parametersorten der Spezifikation sind, denn diese können noch variabel sein. Wenn die Prozedur terminiert, können, da alle Eingaben konkret sind, keine Endlosschleifen entstehen.

Conditional Left Split Die *conditional left split*-Heuristik wendet auf den Antezedenten der Sequenz die *if left*-Regel (siehe B.1) an.

Conditional Right Split Die *conditional right split*-Heuristik wendet *if right*-Regel (siehe B.1) auf den Sukzedenten der Sequenz an.

Dl Case Distinction Die *dl case distinction*-Heuristik löst, wie die *pl case distinction*-Heuristik für PL-Formeln, Konjunktionen im Antezedenten bzw. Disjunktionen im Sukzedenten zwischen DL-Formeln in der Sequenz auf.

Bemerkungen

Die meisten Heuristiken für die Gegenbeispiel-Suche werden auch für Programmbeweise benutzt und standen für die Implementierung schon zur Verfügung. Die *axiom cut*-Heuristik ist eine Erweiterung der *cut*-Heuristik, die schon vorhanden war. Für diese Heuristik gibt der Benutzer Lemmas an und die Heuristik versucht, die Vorbedingungen dieser Lemmas über die *cut*-Regel in eine Sequenz einzuführen. Die *axiom cut*-Heuristik modifiziert die *cut*-Heuristik, indem sie zusätzlich noch alle Axiome betrachtet, um deren Vorbedingungen in eine Sequenz einzuführen. Dasselbe gilt für die *strong simplifier*-Heuristik, die aus der *simplifier*-Heuristik abgeleitet wurde. Es werden aus dem Simplifier-Satz lediglich die Implikationsregeln entfernt, die nicht invertierbar sind. Die Heuristiken

- select goal,
- rewrite,
- give value,
- constructor cut,
- und quantifier weakening

sind speziell für die Gegenbeispiel-Konstruktion entwickelt worden. Die *select goal*-Heuristik und *give value*-Heuristik implementieren die A^*-Suche, wobei die *constructor cut*-Heuristik die Expansionen vornimmt. Die *rewrite*-Heuristik ist für zusätzliche Vereinfachungsschritte verantwortlich, die aus den Axiomen, die der Benutzer nicht in den Simplifier-Satz eingetragen hat, folgen. Die *quantifier weakening*-Heuristik ist letztendlich dafür verantwortlich, daß die Sequenz, die dem Benutzer überlassen wird, um zu entscheiden, ob ein Gegenbeispiel existiert, möglichst einfach repräsentiert wird.

Korrektheit der implementierten Suche

Alle Regeln, die durch die Heuristiken angewendet werden, müssen invertierbar sein, damit eine widersprüchliche Sequenz wirklich zu einem Gegenbeispiel zurückverfolgt werden kann.

Da wir keine nicht deterministische Programme betrachten wollen, sehen wir die Regeln *contract call* und *execute call left/right* als invertierbar an (siehe 5.1.1).

Die *rewrite*-Regel und die *(strong) simplifier*-Regel sind jedoch nur dann korrekt und invertierbar, wenn die verwendeten Rewrite bzw. Simplifier-Lemmas gültig sind. Die rewrite-Lemmas machen dabei keine Probleme, da nur Axiome verwendet werden. In KIV ist es aber nicht notwendig, daß alle Lemmas bewiesen sind, bevor sie als Simplifier-Lemmas eingetragen werden. Das Korrektheitsmanagement sorgt dafür, daß eine Spezifikation bzw. Modul erst dann als bewiesen gilt, wenn alle benutzten Lemmas auch bewiesen sind. Die Korrektheit wird dadurch im Nachhinein garantiert.

Für die Gegenbeispiel-Suche bedeutet dies, daß sie auch nicht invertierbare Regelanwendungen mit der *strong simplifier*-Heuristik durchführen kann. Dadurch kann für eine Aussage ein Gegenbeispiel abgeleitet werden, obwohl tatsächlich keines existieren muß.

Das Problem könnte dadurch behoben werden, daß für die *strong simplifier*-Regel nur Lemmas verwendet werden dürfen, die bewiesen sind. Zusätzlich müssen aber auch all die Lemmas bewiesen sein, die für die Beweise der Simplifier-Lemmas benutzt wurden. Dies nachzuprüfen ist aufwendig. Wir gehen deshalb davon aus, daß der Benutzer nur solche Lemmas in den Simplifier-Satz einträgt, die bewiesen oder sicher gültig sind. Diese Annahme ist deswegen gerechtfertigt, da alle Beweise ungültig werden, wenn die verwendeten Lemmas später nicht beweisen werden können. Deshalb wird der Benutzer bei der Wahl von Simplifier-Lemmas, die nicht bewiesen sind, sehr vorsichtig sein.

6.2.2 Implementierung der Rückverfolgung

Die Rückverfolgung gliedert sich eigentlich in zwei Teile. Zum Einen muß von der widersprüchlichen Aussage, die die Gegenbeispiel-Suche geliefert hat, das konkrete Gegenbeispiel berechnet werden und zum Anderen soll im Originalbeweis das Gegenbeispiel soweit wie möglich zurückverfolgt werden. Da für die Gegenbeispiel-Suche die gleichen Regeln wie für die Beweisführung benutzt werden, kann zum Berechnen des konkreten Gegenbeispiels auch die Funktion benutzt werden, die im Originalbeweis das Gegenbeispiel zurückrechnet. Der Unterschied ist, daß für die Gegenbeispiel-Suche nur invertierbare Regeln benutzt werden. Deshalb erhält man immer ein konkretes Gegenbeispiel für die Konklusio des Suchbaumes. Dies kann man für die Rückverfolgung im Originalbeweis nicht garantieren.

Für die Rückverfolgung werden die Regeln in der Reihenfolge, wie sie auf dem Weg von der Prämisse zur Konklusio stehen, in eine Liste eingetragen. Diese Information erhält man aus dem Beweis- bzw. Suchbaum. Arbeitet man die Liste ab, entspricht dies der Rückverfolgung von der Prämisse zur Konklusio. Je nach Regel, die als nächstes zu bearbeiten ist, wird das Gegenbeispiel entsprechend angepaßt. Da für die stark invertierbaren Regeln keine Anpassungen notwendig sind, müssen nur für die invertierbaren Regeln *insert-eq-var-drop* und *structural induction* und die nicht invertierbaren Regeln *weakening* und *while right* Anpassungen vorgenommen werden. In KIV kann bei den Regeln *all left*, *exists right* und *insert equation* die benutzte Formel entfernt werden. Dies entspricht einer impliziten Anwendung der *weakening*-Regel. Deshalb können bei diesen drei Regeln, wie bei der *weakening*-Regel, Beweisverpflichtungen entstehen.

Die Anpassung des Gegenbeispiels

Bei stark invertierbaren Regeln kann das Gegenbeispiel der Prämisse für die Konklusio übernommen werden. Wir zeigen nun für die nicht stark invertierbaren Regeln, wie die Anpassungen in der Implementierung vorgenommen werden. Dabei orientieren wir uns an den Vorgaben aus Abschnitt 5.1.2.

insert-eq-var-drop Bei der *insert-eq-var-drop*-Regel muß die Variable, die ersetzt wurde, mit dem Term, der für sie eingesetzt wurde, in das Gegenbeispiel mit aufgenommen werden. Falls der Variablenbezeichner schon im aktuellen Gegenbeispiel vorkommt, müssen diese Vorkommen umbenannt werden, damit es keine Konflikte gibt. Diese Anpassungen werden vom System entsprechend der Theorie durchgeführt.

Um die Variable und den zugehörigen Term zu erhalten, speichert man bei der Anwendung der *insert-eq-var-drop*-Regel diese Information in der lokalen Knoten-Information ab und kann sie dann bei der Rückverfolgung einfach ausgelesen.

***structural induction*-Regel** Die *structural induction*-Regel ersetzt die Induktionsvariable x in den Prämissen durch den Term $f(x')$, wobei f ein Konstruktor der entsprechenden Sorte ist. Wie bei der *insert-eq-var-drop*-Regel wird die Gleichung $x = f(x')$ mit in das Gegenbeispiel aufgenommen, wobei eventuell Umbenennungen stattfinden.

Aus technischen Gründen ist es bei der *structural induction*-Regel nicht möglich, die Gleichung $x = f(x')$ in einer Knoten-Information abzulegen. Deshalb wird bei der Rückberechnung diese Gleichung rekonstruiert.

***weakening*-Regel** Für die *weakening*-Regel haben wir drei Möglichkeiten besprochen, ein Gegenbeispiel zurückzuberechnen (siehe Seite 40). Die Möglichkeit, einen erneuten Unterbeweis zu starten, wurde in KIV nicht integriert, da diese Methode zu (zeit)aufwendig ist. Stattdessen wird versucht, die entfernte Formel φ unter den Bedingungen des Gegenbeispiels $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ zu beweisen. Dazu wird ein interner Aufruf des Simplifiers gemacht. Findet er den Beweis, kann das Gegenbeispiel weitergegeben werden. Ansonsten wird der Benutzer gefragt, ob ein Erfüllbarkeitsbeweis für die Formel φ durchgeführt werden soll. Ist dies gewünscht, wird ein Gegenbeispiel für $\chi \wedge (\underline{x} = \underline{t}) \wedge \varphi \vdash$ gesucht. Diese Sequenz entspricht der Aussage $\neg\ (\chi \wedge (\underline{x} = \underline{t}) \wedge \varphi\ )$. Wird

dafür ein Gegenbeispiel $\mathcal{G}'$ gefunden, haben wir Bedingungen, die $\neg\neg(\chi \wedge (\underline{x} = \underline{t}) \wedge \varphi) = \chi \wedge (\underline{x} = \underline{t}) \wedge \varphi$ erfüllen. Die Bedingungen in $\mathcal{G}'$ bilden also das Gegenbeispiel für die Konklusio.

Der Benutzer kann an dieser Stelle die Rückverfolgung auch abbrechen oder die Regel übergehen. Übergeht er die Regel, wird das Gegenbeispiel der Prämisse für die Konklusio der *weakening*-Regel übernommen. Dadurch können Fehler entstehen, die der Benutzer zu verantworten hat.

Da es aufwendig ist, einen Unterbeweis zu starten, wird zuvor noch ein weiterer Test durchgeführt. Kann die Formel $\chi \wedge (\underline{x} = \underline{t}) \vdash \neg\varphi$ bewiesen werden, kann keine erfüllende Belegung für $\chi \wedge (\underline{x} = \underline{t}) \wedge \varphi \vdash$ gefunden werden. Deshalb wird versucht, diese Formel vom Simplifier beweisen zu lassen. Gelingt dies, wird die Rückverfolgung abgebrochen, denn das Gegenbeispiel kann dann nicht weiter zurückgerechnet werden.

In Abschnitt 5.1.4 haben wir gesehen, daß für die Gegenbeispiel-Suche und die Rückberechnung die Induktionshypothese vernachlässigt werden kann. Deshalb entsteht beim Entfernen einer Induktionshypothese keine Beweisverpflichtung. In KIV kann derzeit nur die Induktionshypothese für die noethersche Induktion identifiziert werden. Wird diese entfernt, entsteht keine Beweisverpflichtung. Die Induktionshypothese für strukturelle Induktion findet sich in der Sequenz als $\forall$ -Quantor wieder und da sie nicht identifizierbar ist, werden bei der Rückverfolgung für sie Beweisverpflichtungen erstellt.

Bei der Anwendung der *weakening*-Regel ist es in KIV außerdem möglich, mehrere Formeln auf einmal aus der Sequenz zu entfernen. Dabei können auch Formeln aus dem Sukzedenten entfernt werden, die eigentlich, mit der *neg_r*-Regel (siehe C.8), negiert in den Antezedenten gebracht werden müßten. Für die Rückverfolgung werden alle entfernten Formeln parallel betrachtet, indem sie konjunktiv verknüpft und die Formeln aus dem Sukzedenten negiert werden.

***while right*-Regel** Da ein Gegenbeispiel über die *while right*-Regel meist nicht zurückberechnet werden kann, werden nur einfache Tests durchgeführt. Gibt es ein Gegenbeispiel $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ für die linke Prämisse der Regel (siehe B.3), kann es für die Konklusio verwendet werden, wenn man $\chi \wedge \underline{x} = \underline{t} \rightarrow \neg\varepsilon \wedge \neg\varphi$ beweisen kann (siehe Seite 41). Ein Gegenbeispiel für die rechte Prämisse gilt weiterhin, wenn $\chi \wedge \underline{x} = \underline{t} \rightarrow \Gamma$ gezeigt wird. Es wird mit Hilfe des Simplifiers versucht, die jeweilige Beweisverpflichtung zu zeigen. Gelingt dies, wird das Gegenbeispiel an die Konklusio der *while*-Regel zurückgereicht. Ansonsten wird die Rückverfolgung an dieser Stelle abgebrochen.

***all left, exists right, insert equation*-Regeln** Bei der Regel

$$\frac{\varphi(x'),\ \forall\ x.\ \varphi(x),\ \Gamma \vdash \Delta}{\forall\ x.\ \varphi(x),\ \Gamma \vdash \Delta}\ \text{all left}$$

kann in KIV mit der Regelnanwendung die Formel $\forall\ x.\ \varphi(x)$ entfernt werden. Entsprechend können auch für die Regeln *exists right* und *insert equation* Formeln entfernt werden (siehe B.2). Dies entspricht einem zusätzlichen *weakening*-Schritt nach der eigentlichen Regelanwendung.

Bei der Rückverfolgung wird nachgesehen, ob die Formel $\forall\ x.\varphi(x)$ entfernt wurde, um dann die gleichen Schritte wie bei der *weakening*-Regel anzuwenden, und ein Gegenbeispiel für die Konklusio zu berechnen.

Normierung

Bei der Rückverfolgung werden immer nur Bedingungen für das Gegenbeispiel dazugenommen. Deshalb kann es geschehen, daß in einem Gegenbeispiel $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ in $\underline{x}$ Variablen auftreten, die in der Sequenz nicht mehr vorkommen.

Beispiel 6.7

$$\dfrac{\begin{array}{cc}\vdots & \dfrac{\begin{array}{c}\vdots\\ ord(e \oplus x') \vdash\end{array}}{x = e \oplus x',\ ord(x)}\\ x = @,\ ord(x) \vdash & \end{array}}{ord(x)\vdash}$$

Für $ord(e \oplus x')$ *sei* $\mathcal{G} = (\chi,\ (x') = (t'))$ *ein Gegenbeispiel. Die Rückverfolgung liefert das Gegenbeispiel* $\mathcal{G}' = (\chi,\ (x', x) = (t', e \oplus x'))$ *für die Sequenzen* $x = e \oplus x',\ ord(x) \vdash$ *und* $ord(x) \vdash$

In $ord(x) \vdash$ kommt aber keine Variable x' mehr vor. Deshalb muß das Gegenbeispiel $\mathcal{G}'$ auch keine Bedingungen für x' mehr enthalten. Es würde $\mathcal{G} = (\chi,\ (x) = (a \oplus t'))$ als Gegenbeispiel genügen. Nach jedem Rückverfolgungsschritt kann deshalb eine Normierung durchgeführt werden. Dabei werden alle Variablen aus $\underline{x}$ entfernt, die nicht in der Sequenz enthalten sind. Kommen die Variablen in $\underline{t}$ vor, müssen sie dort durch den zugeordneten Term ersetzt werden.

Desweiteren kann durch eine Fallunterscheidung in der Konklusio eine Variable vorkommen, die in der Prämisse, für die ein Gegenbeispiel existiert, nicht vorkommt.

Beispiel 6.8

$$\dfrac{\varphi(x) \vdash \quad \psi(y) \vdash}{\varphi(x) \vee \psi(y) \vdash}\ dis\_l$$

Sei $\mathcal{G} = (\chi,\ (x) = (t))$ *ein Gegenbeispiel für* $\varphi(x) \vdash$ *, so ist es auch eines für* $\varphi(x) \vee \psi(y) \vdash$ *, denn die* dis_l*-Regel (siehe C.8) ist stark invertierbar.*

Dies bedeutet, daß für alle Belegungen von y ein Gegenbeispiel existiert. Nach der Definition 5.1 für ein Gegenbeispiel muß aber jeder Variable in der Sequenz eine Belegung zugeordnet werden. Dies könnte man erreichen, indem man der Variablen einfach eine Konstruktorkonstante zuweist. Der Benutzer erhält aber mehr Information, wenn er sieht, daß für alle Werte von y ein Gegenbeispiel existiert. Damit kein Induktionsbeweis notwendig ist, wird erst für die Verifikation des Gegenbeispiels den Variablen Werte zugewiesen, für die Ausgabe des Gegenbeispiels an den Benutzer werden jedoch keine Werte zugeordnet.

Das errechnete Gegenbeispiel wird dem Benutzer vorgelegt. Deshalb sollte es so verständlich wie möglich sein. Bei der Rückverfolgung werden für die Variablen der Sequenz die entsprechenden Belegungen berechnet. Dabei wird bei der *insert-eq-var-drop*-Regel die Gleichung $x = \tau$ mit in das Gegenbeispiel aufgenommen. Der Term τ kann dabei alle möglichen Formen annehmen. Für eine Liste x von natürlichen Zahlen kann für die Variable n über den natürlichen Zahlen z.B. die Gleichung $n = x.first$ eingesetzt werden. Diese Gleichung wird bei der Rückverfolgung dann in das Gegenbeispiel mit aufgenommen. Ist für ein Gegenbeispiel die Liste $x = 1 \oplus 2 \oplus @$, so lautet das komplette Gegenbeispiel $\mathcal{G} = (\mathbf{true},\ (n,\ x) = (x.first,\ 1 \oplus 2 \oplus @))$. Für den Benutzer wäre das Gegenbeispiel einfacher zu lesen, wenn es $\mathcal{G}' = (\mathbf{true},\ (n,\ x) = (1,\ 1 \oplus 2 \oplus @))$ lauten würde. Deshalb führen wir zusätzlich zu den oben genannten Normierungen noch eine Vereinfachung des Gegenbeispiels durch.

Verifikation des Gegenbeispiels

Wenn die Rückverfolgung bis zur Konklusio des Originalbeweises vorgedrungen ist, hat sie ein Gegenbeispiel gefunden. Sind bei der Rückverfolgung Beweisverpflichtungen entstanden, die der Benutzter übergangen hat, so kann es sein, daß das Gegenbeispiel nicht zur Konklusio paßt. Deshalb wird vom System die Möglichkeit gegeben, das gefundene Gegenbeispiel $\mathcal{G} = (\chi,\ \underline{x} = \underline{t})$ für die Konklusio $\Gamma \vdash \Delta$ zu verifizieren. Dazu wird mit der Beweisverpflichtung $\vdash \chi \wedge \underline{x} =$

$\underline{t} \rightarrow \neg\ (\Gamma \rightarrow \Delta)$ ein Unterbeweis aufgerufen. Die Normierung sorgt dafür, daß allen freien, erzeugten Variablen der Konklusio durch $\underline{x} = \underline{t}$ ein konkreter Wert zugewiesen wird. Da alle Variablen einen Wert besitzen, können die Programm-Formeln ausgeführt werden und die entstehenden PL-Formeln werden dann durch **rewriting** gelöst, d.h. für Funktionen werden die Definitionen eingesetzt, damit die PL-Formeln ausgerechnet werden können. Dies ist nun möglich, da alle Werte konkret sind. Die sich ergebenden Heuristiken für den Verifikationsbeweis sind deshalb einfach. Für eine Aussage mit DL-Formeln werden die Regeln

- *symbolic execution*-Heuristik,
- *calls concrete*-Heuristik,
- *strong unfold*-Heuristik,
- *calls nonrecursive*-Heuristik,
- *module specific*-Heuristik,
- *simplifier*-Heuristik*,
- *rewrite*-Heuristik*,
- *elimination*-Heuristik*,
- und *dl case distinction*-Heuristik

benutzt. Die mit * gekennzeichneten Heuristiken sind solche, die nur Regeln für PL-Formeln anwenden. Liegt nur eine prädikatenlogische Aussage vor, genügen sie für die Verifikation.

6.2.3 Implementierung der Suchstrategien

Die A^*-Suche wird durch die Heuristiken *select goal* und *give value* gesteuert. Bisher wurde aber noch nicht betrachtet, wie eine Formel bewertet wird. Diese Bewertung beeinflußt erheblich die Suche. Aus Abschnitt 5.1.3 des vorigen Kapitels erhalten wir die Kriterien

- Anzahl der Expansionsschritte,
- Anzahl der Vereinfachungsschritte,
- Anzahl der Einsetzungsschritte von Funktionsdefinitionen,
- Anzahl freier, generierter Variablen,
- Variablen gewichtet mit der Sortenhierarchie,
- Anzahl der Funktionen,
- und Funktionen gewichtet mit der Funktionenhierarchie

für die Bewertung einer Formel. Bevor wir genauer auf die endgültige Bewertung eingehen, sehen wir uns an, wie die Funktionenhierarchie und die Sortenhierarchie, die für die Bewertung einer Formel benutzt werden, berechnet werden.

Funktionenhierarchie

Die Funktionenhierarchie soll die Funktionen einer Spezifikation so anordnen, daß Funktionen, die schwieriger zu lösen sind, weiter oben in der Hierarchie stehen, als Funktionen, die nicht so schwierig zu lösen sind. Für die Funktionenhierarchie wurde eine bestehende Implementierung gewählt. Um den Automatisierungsgrad der Beweise zu erhöhen, wird in KIV versucht, automatische Theorembeweiser zu integrieren. Einigen automatischen Theorembeweiser kann eine Anordnung der Funktionen übergeben werden, die ihm Hinweise gibt, welche Funktionen als ,,schwierig" gelten. Mit dieser Information steuert der automatische Theorembeweiser dann die Beweissuche.

Den gleichen Mechanismus wollen wir auch für die Gegenbeispiel-Suche verwenden, weshalb wir diese Anordnung als Funktionenhierarchie betrachten können. Bei der Berechnung der Anordnung werden Paare $(f,\ g)$ von Funktionen, mit der Bedeutung, daß die Funktion f schwieriger als die Funktion g einzuschätzen ist, gebildet. Mit Hilfe dieser Paare wird dann eine topologische Sortierung durchgeführt, deren Ergebnis die Funktionenhierarchie ist.

Um die Paare zu erzeugen, werden verschiedene Kriterien berücksichtigt. In den Simplifier-Regeln hat der Benutzer festgelegt, welche Funktion er für schwieriger erachtet, denn durch eine Simplifier-Regel $\vdash \psi_1 \wedge \ldots \wedge \psi_n \rightarrow f(x) = g(y)$ weist er das System an, die Funktion f durch die Funktion g zu ersetzen. Der Benutzer schätzt also die Funktion g einfacher als die Funktion f ein. Aus den Simplifier-Regeln werden deshalb Paare $(f,\ g)$ gebildet, die bei der topologischen Sortierung berücksichtigt werden.

Desweiteren werden die Funktionsdefinitionen für die Funktionenhierarchie berücksichtigt. Falls zur Definition der Funktion f die Funktion g benötigt wird, aber für die Definition der Funktion g die Funktion f nicht, wird ebenfalls ein Paar $(f,\ g)$ gebildet. Wenn nach diesen beiden Schritten zwischen zwei Funktionen f und g noch keine Beziehung besteht, so wird eine Paar $(f,\ g)$ erzeugt, wenn die Zielsorte der Funktion f höher in der Sortenhierarchie steht, als die Zielsorte der Funktion g.

Bei der Paarbildung kann es geschehen, daß Zyklen entstehen, d.h es existieren Paare $(f,\ g)$ und $(g,\ f)$, bzw. $(f_1,\ f_2)\ \ldots\ (f_{n-1},\ f_n)$ mit $f_n = f_1$. Dann kann die Menge der Paare nicht topologisch sortiert werden. Wenn dies der Fall ist, wird eines der Paare entfernt und somit der Zyklus aufgebrochen. Da die Funktionenhierarchie nur zur Unterstützung einer heuristischen Berechnung dient, ist diese Art Zyklen zu beseitigen, ausreichend.

Sortenhierarchie

Die Sortenhierarchie soll, wie die Funktionenhierarchie, die Sorten, in schwieriger und weniger schwierig zu lösen, einteilen. Eine Sorte, die für ihre Definition eine andere Sorte als Parameter benutzt, steht höher in der Hierarchie, wie die in der Definition verwendete Sorte. Um die Sortenhierarchie zu berechnen, betrachten wir die Spezifikationsstruktur. Alle Sorten einer Spezifikation sind auf der selben Hierarchiestufe. Da in einer Spezifikation meist nur eine Sorte definiert wird, verliert man dadurch keine Information. Die implementierte Sortenhierarchie entspricht dann nicht der erwarteten Sortenhierarchie, wenn in einer Spezifikation zwei Sorten definiert werden und eine davon in der Definition der anderen Sorte vorkommt. Meist werden nur dann in einer Spezifikation zwei Sorten definiert werden, wenn es rekursive Sortendefinitionen sind, d. h. die Sorte s_1 benutzt die Sorte s_2 in der Definition und die Sorte s_2 benutzt die Sorte s_1 in ihrer Definition. In diesem Fall sollen die beiden Sorten s_1 und s_2 aber auf der gleichen Stufe stehen.

Die Sortenhierarchie wird aus der Spezifikationsstruktur abgeleitet. In der Sortenhierarchie stehen die Sorten, die in der Spezifikation Sp_1 definiert wurden über den Sorten, die in der Spezifikation Sp_2 definiert wurden, wenn in der Spezifikation Sp_1 Funktionen oder Sorten verwendet werden, die in der Spezifikation Sp_2 definiert wurden. Die Sorten der Spezifikation Sp_1 müssen zwar keine Sorten der Spezifikation Sp_2 benützen, aber da die Sorten der Spezifikation Sp_2 keine Sorten der Spezifikation Sp_1 benützen können, widerspricht die Anordnung nicht der beabsichtigten Sortenhierarchie, sondern bildet höchstens mehr Restriktionen, als nötig. Dafür müssen wir uns nicht die einzelnen Sortendefinitionen ansehen, um die Sortenhierarchie zu erhalten, sondern nur noch die Anordnung der Spezifikationen.

Für die Implementierung ist es einfacher, die Sortenhierarchie als Liste darzustellen. Bei rekursiven Sortendefinitionen müßten die Sorten jedoch auf der selben Stufe stehen. Diese Bedingung wird bei der Implementierung nicht beachtet, sondern eine der Sorten wird vor der anderen eingeordnet. Da dies aber nur seltene Fälle sind und die Sortenhierarchie nur für die heuristische Auswahl von Variablen verwendet wird, ist dies keine große Einschränkung.

Bewertung der Formeln

Für die Bewertung einer Formel wurden einige verschiedenen Methoden getestet (siehe 7.2.1). Für die Implementierung wurde eine Bewertung ausgewählt, die in allen Beispielen einen guten Eindruck hinterließ. Da für die Suche eine faire Strategie benötigt wird, muß die Entfernung vom Ausgangspunkt der Suche berücksichtigt werden. Dazu wird die Anzahl der Beweisschritte von der Konklusio bis zur aktuellen Prämisse gewählt. Für die Entfernung zum Ziel, also einer variablenfreien, widersprüchlichen Formel, wurde die Anzahl der freien, generierten Variablen der Sequenz und die Summe der Funktionsgewichte gewählt.

Für die Funktionsgewichte werden alle Funktionen einer Formel berücksichtigt, d.h. nicht nur Funktionen, die auf oberster Formelebene stehen, sondern auch Funktionen, die als Parameter auftauchen. Für die Formel $p(f(g(x), y)) \rightarrow q(z)$ werden die Funktionen f, g, p und q berücksichtigt und mit der Position in der Funktionenhierarchie bewertet.

Je mehr Funktionen spezifiziert sind, desto größer ist die Funktionenhierarchie und desto größer die Werte für die Funktionen. Damit die Bewertung unabhängig von der Anzahl der spezifizierten Funktionen ist, werden die anderen Meßgrößen mit der Hierarchielänge multipliziert.

Grundsätzlich gehen wir davon aus, daß Formeln, die DL-Formeln enthalten, schwieriger einzustufen sind, als Formeln, die nur noch PL-Formeln enthalten. Deshalb wird zu dem Formelgewicht einer Formel, die DL-Formeln enthält, noch ein Mindestwert addiert.

Diese Bewertung hat sich, mit der entsprechenden Auswahlstrategie der Variablen für die Expansion, die wir als nächstes besprechen, gut bewährt.

Auswahl der Variablen für die Expansion

Wie für die Bewertung einer Formel wurde auch für die Auswahl der zu expandierenden Variable schon einige Kriterien angegeben (siehe 5.1.3), nämlich

- Anzahl der Expansionen,
- Anzahl der Vorkommen in der Formel,
- Sorte der Variablen,
- Vorkommen auf rekursiven Positionen,
- und Bevorzugung von Variablen, die nur einen Konstruktor haben.

Variablen, deren Sorte nur aus einem Konstruktor besteht, führen bei der Expansion zu keiner weiteren Fallunterscheidung. Sind solche Variablen in einer Sequenz vorhanden, werden diese immer zuerst für die Expansion ausgewählt. Sind keine Variablen dieser Art mehr in der Sequenz, werden die Variablen erzeugter Sorten bewertet und dann die Variable mit dem maximalen Wert für die Expansion ausgewählt.

Die Anzahl der Expansionen einer Variablen wird negativ bewertet, denn durch diese Größe soll garantiert werden, daß nicht immer dieselbe Variable expandiert wird. Einen positiven Wert erhält eine Variablen für die Sorte und für Vorkommen auf rekursiven Positionen von Funktionen der Sequenz. Für die Sorte bekommt eine Variable einen Werte, entsprechend der Sortenhierarchie. Je höher die Sorte der Variable in der Sortenhierarchie steht, desto höher ist der Wert für die Variable. Für die Vorkommen auf rekursiven Positionen werden nur die Funktionen einer Formel betrachtet, die auf oberster Formelebene auftreten. Funktionen, die als Parameter anderer Funktionen auftreten, werden nicht berücksichtigt. Für die Sequenz $p(f(g(x), y)) \rightarrow q(z)$ werden die

Funktionen p und q betrachtet, aber nicht die Funktionen f und g. Ist p rekursiv definiert, so bekommt sowohl x als auch y den Wert der Funktion p nach der Funktionenhierarchie, da beide Variablen als Argumente vorkommen. Schlußendlich werden die Werte für die einzelnen Variablen aufsummiert, und die Variable, mit dem maximalen Wert für die Expansion ausgewählt.

Für die Auswahl der Variablen gilt im wesentlichen dasselbe, wie für die Bewertung von Formeln. Es wurden einige Varianten für die Bewertung getestet (siehe 7.2.2) und die vorgestellte Variante, da sie mit der oben beschriebenen Bewertung von Formeln einen guten Eindruck hinterlassen hat, integriert.

6.3 Aufwand für die Implementierung

Die Implementierung besteht aus den zwei Teilgebieten Suche und Rückverfolgung. Die Suche wurde vollständig über Heuristiken implementiert, deren Abarbeitungsstrategie in KIV schon integriert war. Deshalb belief sich der Hauptaufwand der Implementierung, die Funktionen für die Bewertung der Sequenzen und der Variablen zu schreiben, und die dafür benötigten Größen, wie z.B die Funktionenhierarchie und die Sortenhierarchie, zur Verfügung zu stellen.

Für die Rückverfolgung mußten Funktionen geschrieben werden, die die Anpassung des Gegenbeispiels vornehmen. Dabei können immer wieder Beweisverpflichtungen auftreten, die erneut zu einem Unterbeweis führen, weshalb die Rückverfolgung eng mit der Steuerung der Oberfläche verbunden ist. Für den Unterbeweisaufruf stand die benötigte Funktionalität in KIV schon zur Verfügung, so daß ein Unterbeweis nur mit den entsprechenden Parametern aufgerufen werden mußte. Deshalb war die Programmierung der Rückverfolgung und die Steuerung der Oberfläche in zwei bis drei Wochen abgeschlossen. Die Programmierung der Suche dauerte dagegen mit vier bis sechs Wochen etwa doppelt so lange. In dieser Zeit sind die in Abschnitt 7.2 aufgeführten Versuche für die Wahl einer günstigen Bewertungs- und Auswahlstrategie enthalten.

Während des Projektes entstanden ca. 3500 Programmzeilen PPL-Code. Dabei fallen jeweils ca. 1000 Zeilen für die Programmierung der Bewertungsstrategien und der Rückverfolgung an. Die restlichen Programmzeilen wurden für die einzelnen Heuristiken und für Änderungen, die sich durch die Gegenbeispiel-Suche im System ergaben, benötigt. Ein Überblick über die erstellten Dateien befindet sich im Anhang D.

Kapitel 7

Experimentelle Ergebnisse

In diesem Kapitel besprechen wir die Ergebnisse von Experimenten, die mit der in KIV implementierten Gegenbeispiel-Suche durchgeführt wurden. Wir haben getestet, wie sich die Gegenbeispiel-Suche für die fehlerhaften Beispiele aus Kapitel 3 verhält, wo wir für die einzelnen Beispiele Erwartungen an das konkrete Gegenbeispiel und an das Verhalten der Rückverfolgung gestellt haben. Im ersten Abschnitt besprechen wir, inwieweit diese Erwartungen erfüllt werden und wie eventuelle Abweichungen zustande kommen. Desweiteren wurden einige Bewertungsfunktionen für die A^*-Suche und einige Strategien zur Variablenauswahl für die Expansionen getestet. Diese Ergebnisse werden im nachfolgenden Abschnitt besprochen.

7.1 Die Beispiele aus Kapitel 3

Die Gegenbeispiel-Suche, die in KIV implementiert wurde, konnte alle Beispiele aus Kapitel 3 lösen. Die Abbildung 7.1 gibt einen Überblick, wieviel Zeit für die einzelnen Beispiele benötigt wurde, um ein Gegenbeispiel zu finden.

Die Beispiele *graph1* und *graph2* tauchen in der Tabelle zweimal auf. Für die formulierten Behauptungen (siehe Beispiel 3.8 bzw. 3.9) wurden zusätzlich noch Versuche mit Formeln durchgeführt, die sich aus Beweisversuchen für die beiden Beispiele ergeben haben. Diese (einfacheren) Formeln werden, mit einem * gekennzeichnet, in der Tabelle aufgeführt.

Für die Zeitmessung wurde auf die in Lisp vorhandene Funktion **time** zurückgegriffen. Die Spalte *time* in Abbildung 7.1 gibt die Zeit in Sekunden an, die für die Lösung benötigt wurde. Die Spalten *steps* und *goals* geben an, wieviele Beweisschritte erzeugt wurden, bzw. wieviele verschiedenen Prämissen bei der Gegenbeispiel-Suche entstanden sind. Diese Daten sind vor allem für Vergleiche mit Messungen aus Abschnitt 7.2 interessant. Die Messungen wurden auf einer Sun Ultrasparc 1 durchgeführt.

Die gemessenen Zeiten enthalten nicht die Berechnungen für die Hilfsdaten (Funktionenhierarchie, Sortenhierarchie, ...), die für die Bewertungsfunktionen benötigt werden. Diese Berechnung muß pro Beweis nur einmal ausgeführt werden, und führt deshalb nur beim ersten Aufruf der Gegenbeispiel-Suche zu Verzögerungen. Bei Spezifikationen, die in der Größenordnung der Beispielspezifikationen liegen, dauert diese Berechnung ein bis zwei Sekunden.

7.1.1 Binärsuche auf Arrays

Im Versuch über die binäre Suche auf Arrays, ist das Array über eine Parametersorte spezifiziert. Deshalb entstehen bei der Gegenbeispiel-Suche Anfragen an den Benutzer. Im ersten Beispiel muß der Benutzer entscheiden, ob $\Box \not\ll ele \wedge \Box \neq ele$ in einem Modell der Spezifikation erfüllt werden kann. Dabei ist $\Box$ ein Defaultelement, welches zum Initialisieren des Arrays benutzt wird. Für das Defaultelement sind keine Axiome in der Spezifikation vorhanden. Deshalb kann es jeden Wert der Elementsorte annehmen. In den Modellen der Spezifikation, in denen es kleinere Element als

Beispiel	time (sek.)	steps	goals
Binärsuche			
hoare1 (3.1)	222.06	77	12
hoare2 (3.2)	342.11	234	33
Mergesort			
merge1 (3.3)	3.12	11	1
merge2 (3.4)	7.17	32	2
merge3 (3.5)	35.61	86	8
Reg.Expr.			
match1 (3.6)	8.82	3	1
match2 (3.7)	8.21	1	1
Compiler			
comp. (3.15)	12.83	67	18
comp. (3.14)	3.99	24	7

Beispiel	time (sek.)	steps	goals
Graphen			
graph1* (3.8)	27.36	31	10
graph1 (3.8)	20.25	31	11
graph2* (3.9)	144.11	178	26
graph2 (3.9)	539.57	667	79
Bin.Arith.			
binär1 (3.10)	8.98	101	5
binär2 (3.11)	7.81	45	2
Int.List			
iv-list1 (3.12)	81.31	255	23
iv-list2 (3.13)	34.75	127	15

Abbildung 7.1: Zusammenfassung der Ergebnisse

das Defaultelement gibt, kann diese Gleichung erfüllt werden. Damit erhalten wir für das Beispiel *hoare1* (3.1) das Gegenbeispiel

$$min = n_1 + 1\ ,\ n = 0,\ lower_0 = 0,\ a = put(mkarray(n_4 + 1\ ),\ 0,\ ele),\ max = 0,$$
$$upper_0 = n_5 + 2,\ \Box \not\ll ele \wedge \Box \neq ele$$

Trotz der Korrektheit der Gegenbeispiel-Suche überzeugen wir uns davon, daß diese Belegung ein Gegenbeispiel für das Beispiel *hoare1* ist. Die Variablen $n_1,\ n_4,\ n_5$ können noch frei belegt werden. Dadurch wird nur festgelegt, daß die Variablen $min \geq 1$ und $upper_0 \geq 2$ sein müssen und das Array mindestens die Größe 1 haben muß. Diese Angaben genügen, um ein Gegenbeispiel zu erhalten, denn die Funktion *get* liefert das Defaultelement zurück wenn der Index größer als das Array ist oder auf einen Index zugegriffen wird, an dessen Stelle noch kein Element eingetragen wurde. Mit den obigen Belegungen liefert die Funktion *get* für jeden Index größer 0 immer das Defaultelement zurück. Da auf dem Index 0 das Element *ele* und nach Voraussetzung $\Box \ll ele$ ist, ist das Array nicht sortiert und wir haben ein Gegenbeispiel gefunden.

Beim zweiten Beispiel *hoare2* (3.2) verhält es sich ähnlich. Dort entsteht die Anfrage $ele \ll \Box \wedge ele \ll searched \wedge ele \neq \Box$, die in den Modellen erfüllt werden kann, in denen das Defaultelement einen größten Wert als irgendein anderes Element erhält. Die Gegenbeispiel-Suche liefert das Gegenbeispiel

$$upper_0 = 1,\ min = 0,\ n = 0,\ lower_0 = 0,\ max = 1,$$
$$a = put(mkarray(n_6 + 2,\ (lower_0 + upper_0) \mid 2),\ ele),\ ele \ll \Box \wedge ele \ll searched \wedge ele \neq \Box$$

Die beiden Gegenbeispiele unterscheiden sich von den Erwartungen aus Abschnitt 3.1. Dort wurde jeweils ein zweielementiges Array als Lösung angegeben, welches jedoch nicht die ,,einfachste" Lösung ist. Für solch eine Lösung müßte die Suche in dem Zweig des Suchbaumes suchen, in dem ein zusätzliches Element in das Array einfügt wird. Formeln, die diese Situation beschreiben, haben aber ein größeres Formelgewicht (mehr Konstruktorfunktionen) und werden bei der Suche nicht bevorzugt betrachtet. Die Rückverfolgung stoppt in beiden Beispielen an der *while*-Regel, bei der die falsche Invariante eingegeben wurde, und erfüllt somit die Erwartungen.

Für beide Beispiele wurde die *rewrite*-Heuristik benötigt, die speziell für die Gegenbeispiel-Suche erstellt wurde. Arrays sind als erzeugte Datentypen spezifiziert. Deshalb existiert das Extensionalitätsaxiom $a_1 = a_2 \leftrightarrow size(a_1) = size(a_2) \wedge \forall\ i.\ i < size(a_1) \rightarrow get(a_1,\ i) = get(a_2,\ i)$, welches die Gleichheit von Arrays ausdrückt. Dieses Axiom ist nicht als Simplifier-Lemma eingetragen, da in einer Sequenz die Gleichung $a_1 = a_2$ leichter zu verstehen ist, wie die rechte Seite der Äquivalenz.

Durch die Expansion einer Arrayvariablen a entstehen die Fälle $a = mkarray(n)$ bzw. $a = put(a_0,\ n,\ ele) \wedge a \neq a_0$. Für den zweiten Fall benötigt die Gegenbeispiel-Suche die Information, wie sich die Arrays a und a_0 unterscheiden. Um diese Information zu erhalten, muß das Extensionalitätsaxiom angewendet werden. Deshalb benötigen wir die *rewrite*-Heuristik, die Axiome anwendet, die nicht als Simplifier-Lemmas eingetragen sind.

Die Zeiten für die Suche betragen vier bis fünf Minuten. Im Vergleich zu den automatischen Theorembeweisern (siehe Kapitel 4) ist dies ganz gut, da die getesteten automatischen Theorembeweiser das Problem überhaupt nicht lösen können[1], aber die Zeiten erfüllen nicht ganz die gestellten Ansprüche. Die beiden anderen Beispiele über Arrays, die mit den automatischen Theorembeweisern noch getestet wurden, kann die Gegenbeispiel-Suche in zwei (Beispiel *hoare1′* (4.1)) bzw. drei (Beispiel *hoare1″* (4.2)) Sekunden, und damit deutlich schneller als die automatischen Theorembeweiser (siehe Abbildung 4.1), lösen.

7.1.2 Mergesort auf Listen

Wie die Arrays sind auch die Listen, die im Versuch Mergesort auf Listen verwendet werden, über eine Parametersorte spezifiziert. Für alle drei Versuche entsteht die Anfrage, ob

$$ele_1 \neq ele$$

erfüllbar ist. Für das Beispiel *merge1* (3.3) ist dies schon die Lösung. Das Beispiel *merge2* (3.4) berechnet daraus die Lösung

$$ele_0 = ele_1, ele_1 \neq ele$$

und das Beispiel *merge3* (3.5) die Lösung

$$l_4 = @,\ ele_2 = ele_1,\ l_8 = @,\ l_6 = @,\ ele_1 \neq ele.$$

Diese Gegenbeispiele wurden auch in Kapitel 3 erwartet. Die Rückverfolgung konnte alle drei Fälle bis zum Beweisanfang zurückverfolgen und auch die Suchzeiten sind im erwarteten Bereich. Im Gegensatz zu den automatischen Theorembeweisern läßt sich die in KIV implementierte Gegenbeispiel-Suche nicht so sehr von der Anzahl der Variablen in den Formeln beeinflussen, denn die automatischen Theorembeweiser konnten das Beispiel *merge3* (3.5) nicht mehr lösen (siehe Abbildung 4.2).

In den Beispielen entstehen während der Gegenbeispiel-Suche immer wieder Formeln der Form $del\_once(ele, ele_1 \oplus l) = l'$. Die Funktion *del_once* entfernt das Element *ele* einmal aus der Liste $ele_1 \oplus l$ und wird für die Berechnung der Permutation benötigt. Die Axiomatisierung von *del_once* ist

ax-1: $del\_once(ele,\ @) = @$
ax-2: $del\_once(ele,\ ele \oplus l) = l$
ax-3: $ele \neq ele_1 \rightarrow del\_once(ele,\ ele_1 \oplus l) = ele_1 \oplus del\_once(ele,\ l).1$

Die Sorte der Listenelemente ist nicht erzeugt, weshalb man keine Bedingungen einführen kann, die entscheiden, ob in der Formeln $del\_once(ele, ele_1 \oplus l) = l'$ die Elemente *ele* und ele_1 gleich sind, oder nicht. Man kann also nie die Definition von *del_once* einsetzten, da nicht entschieden werden kann, ob das Axiom ax-2 oder das Axiom ax-3 angewendet werden soll. Deshalb wird mit der *axiom cut*-Heuristik, für die beiden Fälle $ele = ele_1$ bzw. $ele \neq ele_1$, eine Fallunterscheidung eingeführt. An der Vorbedingung des Axioms ax-3 erkennt die Heuristik, daß diese Fallunterscheidung nützlich sein kann. Nachdem in beiden Fällen die Definition von *del_once* eingesetzt worden ist, erhält man die Formeln $l = l'$ bzw. $ele_1 \oplus del\_once(ele, l')$ und die Suche kann fortfahren. Ohne diese Heuristik kann eine Formel ohne erzeugte Variablen wie z.B. $del\_once(ele, ele_1 \oplus @) = @$ nicht zu $ele = ele_1$ vereinfacht werden. Für den Benutzer ist es aber viel einfacher zu entscheiden, ob $ele = ele_1$ erfüllbar ist, als zu entscheiden, ob $del\_once(ele, ele_1 \oplus @) = @$ erfüllbar ist.

[1] auch nicht, wenn sie ohne Zeitschranke die Lösung suchen könnten; dieser Versuch wurde nach ca. 20 Minuten abgebrochen

7.1.3 Reguläre Ausdrücke

Die beiden Beispiele über reguläre Ausdrücke enthalten keine Variablen mehr. Deshalb muß nur noch getestet werden, ob die Beispiele widersprüchlich sind. Dies konnte die Gegenbeispiel-Suche **erfolgreich** entscheiden. Die Rückverfolgung konnte die Gegenbeispiele bis zum jeweiligen Beweisanfang zurückberechnen.

An diesem Beispiel traten einige Besonderheiten auf. Um eine Lösung für die Beispiele zu erhalten, mußte die Definition von *matches* für *-Ausdrücke eingesetzt werden. Diese lautet

$$r^* \; matches \; s \leftrightarrow \exists \; s_1, s_2. \; s = s_1 \mathbin{++} s2 \wedge r \; matches \; s_1 \wedge r^* \; matches \; s_2,$$

wobei ++ zwei Strings aneinanderhängt. Die Definition ist aber nicht als Simplifier-Lemma eingetragen, denn sie kann bei der Vereinfachung durch den Simplifier zu einer Endlosschleife führen. Die Formel $@_r{}^* \; matches \; @_s$ würde der Simplifier mit der Instantiierung $s_1 = @_s$ und $s_2 = @_s$ zu der Formel

$$@_s = @_s \mathbin{++} @_s \wedge @_r \; matches \; @_s \wedge @_r{}^* \; matches \; @_s$$

vereinfachen. In dieser Formel ist wieder die $@_r{}^* \; matches \; @_s$ enthalten, und es könnte erneut der obige Vereinfachungsschritt durchgeführt werden und man wäre in einem Zyklus.

Deshalb wird dieses Axiom durch die *rewrite*-Heuristik angewendet. Diese wendet jedes Axiom mit derselben Instantiierung nur einmal an. Damit kann kein Zyklus entstehen und die Gegenbeispiel-Suche kann das Problem auf die leere Sequenz zurückführen. Das erste Beispiel kann damit gelöst werden.

Eine weitere Besonderheit trat für das Beispiel $\vdash mkreg(c)^* \; matches \; @_s$ auf. Wenn für diese Formel die Definition von *matches* eingesetzt wird, gelangt man zu der Formel

$$\vdash \exists \; s_1, s_2. \; @_s = s_1 \mathbin{++} s_2 \wedge mkreg(c) \; matches \; s_1 \wedge mkreg(c)^* \; matches \; s_2 \; (*),$$

die nicht weiter vereinfacht werden kann. Für die Benutzeranfrage, ob eine Formel erfüllbar ist, oder nicht, möchte man keine Quantoren haben. Die *quantor weakening*-Heuristik kann die Formel entfernen. Sie versucht das Negat von (*), nämlich

$$@_s = s_1 \mathbin{++} s_2, \; mkreg(c) \; matches \; s_1, \; mkreg(c)^* \; matches \; s_2 \vdash ,$$

zu beweisen. Die Variable s_1 kann nur den Wert $@_s$ annehmen und dadurch entsteht mit dem Axiom $mkreg(c) \; matches \; s_1 \leftrightarrow s_1 = c$ ein Widerspruch im Antezedenten und der Beweis kann geschlossen werden. Das Axiom besagt, daß ein regulärer Ausdruck c nur einen String s *matchen* kann, wenn s der String c ist. Mit der *quantor weakening*-Heuristik führt auch das zweite Beispiel zu einer leeren Sequenz, und damit ist keine Benutzerinteraktion notwendig.

7.1.4 Wege in Graphen

Die Beispiele für die Graphen konnten gelöst werden. Für das Beispiel *graph1* (3.8) berechnet die Gegenbeispiel-Suche mit

$$x = a \oplus a \oplus @, \; g = @_g +_e v \mapsto v$$

und für das Beispiel *graph2* (3.9) mit

$$v_1 \neq v_2, \; x = v_1 \oplus v_2 \oplus v_1 \oplus @, \; g = @_g +_e v_1 \mapsto v_2 +_e v_2 \mapsto v_1$$

das gewünschte Ergebnis. Wurden die Beispiele aus einem laufenden Beweisversuch heraus aufgerufen, konnte die Rückverfolgung auch für die Konklusio die erwarteten Gegenbeispiele berechnen.

Die beiden Beispiele für den Graphen ohne Schlingen dauern allerdings beinahe 10 Minuten. Dies liegt daran, daß für einen Graphen zwei Kanten und für die Pfadliste drei Elemente eingesetzt werden müssen, um ein Gegenbeispiel zu erhalten. Dadurch benötigen wir für den Graphen drei

und für die Pfadliste vier Fallunterscheidungen. Es entstehen viele Fälle, die untersucht werden müssen, um ein Gegenbeispiel zu erhalten. Deshalb ist das Problem so schwierig zu lösen.

Weiterhin ist zu bemerken, daß die Knoten im Graphen Parametersorten sind. Für das Beispiel ohne Schlingen entsteht deshalb die Anfrage

$$v_2 \neq v_1 \wedge \neg\ (\exists\ v_0, x, y, z.\ v_2 \oplus v_1 \oplus @ = x \oplus v_0 \oplus y \oplus v_0 \oplus z),$$

die erfüllbar ist. Die Ungleichung $v_2 \neq v_1$ ist erfüllbar, da es Modelle der Spezifikation gibt, die verschiedene Knoten enthalten. Der Quantor kann nicht erfüllt werden, denn es kann keine Belegung von $x \oplus v_0 \oplus y \oplus v_0 \oplus z$ gleich $v_2 \oplus v_1 \oplus @$ sein, wenn die Elemente v_2 und v_1 verschieden sind. Dazu müßte die Liste genau aus zwei verschiedenen Elementen bestehen. Dies kann nicht sein, da schon zwei mal das Element v_0 vorkommt. Diesen Quantor kann die *quantor weakening*-Heuristik nicht entfernen, denn der Simplifier kann die Aussage

$$v_1 \neq v_2 \rightarrow \neg\ (\exists\ v_0, x, y, z.\ v_2 \oplus v_1 \oplus @ = x \oplus v_0 \oplus y \oplus v_0 \oplus z)$$

nicht beweisen. Ein Unterbeweis mit weiteren Heuristiken könnte diesen Beweis führen, ist in der Implementierung aber nicht vorgesehen.

Für Graphen, die Schlingen enthalten dürfen, fällt keine Anfrage an der Benutzer an, denn für die Lösung werden keine verschiedenen Knoten benötigt. In diesem Fall entfernt die *quantor weakening*-Heuristik den Quantor

$$\exists\ v_0, x, y, z.\ a \oplus @ = x \oplus v_0 \oplus y \oplus v_0 \oplus z.$$

Das Negat dieser Aussage kann der Simplifier beweisen. An diesem Beispiel sieht man, daß die *quantor weakening*-Heuristik noch verbessert werden könnte, wenn zusätzlich zum Simplifier weitere Heuristiken für die Beweise benutzt werden würden.

7.1.5 Binär-Arithmetik

Die Beispiele über die Binärarithmetik besitzen Programmformeln. Deshalb muß für diese Versuche das Heuristik-Set mit Programmheuristiken ausgewählt werden. Diese können die leere Sequenz ableiten und die Rückverfolgung berechnet das Gegenbeispiel

$$b = 1.0$$

für die Konklusio.

7.1.6 Intervall-Listen

Die Intervall-Listen enthalten auch Programme. Deshalb wird wieder das DL-Heuristik-Set für die Gegenbeispiel-Suche ausgewählt. Wird die Gegenbeispiel-Suche von einem Beweisversuch aus aufgerufen, entsteht durch die Rückverfolgung das Gegenbeispiel

$$n = 1,\ x = [0,\ 1] \oplus [2,\ 2] \oplus @,\ x = [2,\ 2] \oplus @$$

an der Konklusio des Beweises. Dieses Gegenbeispiel erhält man auch, wenn man die Gegenbeispiel-Suche direkt für die Konklusio startet, dann benötigt man allerdings keine Rückverfolgung mehr.

Ohne *Normierung* (siehe 6.2.2) würde bei dem Beispiel, das aus einem Beweisversuch heraus die Gegenbeispiel-Suche aufruft, das Gegenbeispiel

$$n = ((x.\mathit{first}).2)\ +1\ ,\ x_1 = [(x.\mathit{first}).1,\ ((x.\mathit{first}).2)\ +1\ ] \oplus x.\mathit{rest},\ x = [0,\ 0] \oplus [2,\ 2] \oplus @$$

berechnet werden. Diese Form entsteht durch die Struktur des Programmes *insert#*, das ein Element in eine Intervall-Liste einsortieren soll. Bei einem Beweisversuch führt der Benutzer das Programm aus. Durch die Kontrollstrukturen des Programms *insert#* werden Fallunterscheidungen eingeführt. Für $m_2 = (x.\mathit{first}).2$ und $m1 = (x.\mathit{first}).1$ entschiedet das Programm, ob $n = m_2\ +1$

ist. In diesem Fall wird die Ergebnisliste zu $[m_1, m_2 + 1\,] \oplus x.rest$ berechnet. Diese Struktur findet sich im Gegenbeispiel wieder. Die Normierung kann aus diesem Gegenbeispiel, wie wir oben gesehen haben, aber die wesentlichen Informationen herausfiltern und übersichtlich darstellen.

Führt man die Gegenbeispiel-Suche von Anfang an aus, erhält man auch ohne Normierung das obige Gegenbeispiel. In diesem Fall wird das Programm *insert#* erst ausgeführt, wenn die Bedingungen der Kontrollstrukturen aufgrund der Variablenbelegungen entschieden werden können. Die Kontrollstrukturen werten dann die Belegungen der Variablen nur noch aus und betrachten den entsprechenden Fall. Deshalb spiegelt sich die Struktur des Programms nicht im Gegenbeispiel wieder.

7.1.7 Compiler

Für das Beispiel *compiler2* (3.15) entsteht das Gegenbeispiel

$$x = @,\ t_0 = 1,\ t_1 = 0,$$

das bis zur Konklusio zurückverfolgt werden kann. Dort hat es die Form

$$t = 1\ \rightarrow\ 0,\ x = @$$

Dieses Ergebnis erhält man auch, wenn das Gegenbeispiel von Anfang an gesucht wird (Beispiel *compiler1* (3.14)). Damit sind auch für dieses Beispiel alle Erwartungen aus Kapitel 3 erfüllt. Es werden die entsprechenden Gegenbeispiele gefunden und die Rückverfolgung kann sie bis zur Konklusio zurückberechnen.

Für den Beweisversuch benötigt man eine strukturelle Induktion, bei der die Induktionshypothese, eine $\forall$-quantifizierte Formel, entfernt wird. Bei der Rückverfolgung muß getestet werden, ob das gefundene Gegenbeispiel auch den Quantor erfüllt. Dieser Test kann in diesem Fall automatisch vom Simplifier gelöst werden. Die Rückverfolgung läuft deshalb ohne Benutzerinteraktion bis zur Konklusio und liefert das angegebene Gegenbeispiel.

7.1.8 Zusammenfassung

Die Gegenbeispiel-Suche konnte alle Beispiele lösen. Damit ist sie der Lösung mit Hilfe der automatischen Theorembeweiser weit überlegen. Auch die Rückverfolgung hat in allen Beispielen die Erwartungen, die in Kapitel 3 gestellt wurden, erfüllt. Für vier von 17 Beispielen benötigt die Gegenbeispiel-Suche länger, als man sich vorstellt. Dies ist vor allem dann der Fall, wenn viele Variablen im Laufe der Gegenbeispiel-Suche auftreten. Jedoch konnte keiner der automatischen Theorembeweiser eines dieser vier Beispiele lösen. Dies ist ein Indiz dafür, daß es sich um schwierige Probleme handelt und man kann sagen, daß die Gegenbeispiel-Suche die gestellten Anforderungen erfüllt.

Im Abschnitt 2.2 entschieden wir uns für eine optimistische Strategie für die Gegenbeispiel-Suche, d.h. die Gegenbeispiel-Suche soll erst dann aufgerufen werden, wenn man bei einem Beweisversuch an eine Formel gelangt, von der man annimmt, daß man sie nicht beweisen kann. Die Vermutung war, daß durch die optimistische Strategie die Gegenbeispiel-Suche schneller ein Gegenbeispiel findet, da sie schon Informationen besitzt, in welchem Bereich ein Gegenbeispiel zu suchen ist. Aufgrund der Meßergebnisse (siehe Abbildung 7.1) kann diese Vermutung bestätigt werden. Bei allen Beispielen, bei denen sowohl ein Gegenbeispiel für die Konklusio, als auch ein Gegenbeispiel für eine Formel, die während eines Beweisversuchs entstanden ist, gesucht wurde, benötigte die Gegenbeispiel-Suche (teilweise erheblich) länger, wenn sie ein Gegenbeispiel für den Beweisanfang suchen mußte. Deshalb ist der optimistische Ansatz durchaus gerechtfertigt.

7.2 Test der Auswahlstrategien

In diesem Abschnitt besprechen wir die Tests der Auswahlstrategien für Formeln und Variablen für die Expansion. Diese Auswahl wird durch Bewertungsfunktionen gesteuert. Für die Vergleiche werden die Beispiele *iv-list2* (3.13), *graph1* (3.8), *graph2* (3.9), *hoare1* (3.1) und *hoare2* (3.2)

Beispiel	Breitensuche		Tiefensuche		Strategie 1		Strategie 2	
	goals	steps	goals	steps	goals	steps	goals	steps
iv-list2 (3.13)	23	220	7	40	4	64	2	58
graph1* (3.8)	23	74	-	-	8	23	4	23
graph1 (3.8)	58	170	-	-	16	44	5	23
graph2* (3.9)	-	-	-	-	12	69	6	49
graph2 (3.9)	-	-	-	-	21	108	6	50
hoare1 (3.1)	-	-	9	62	-	-	9	62
hoare2 (3.2)	-	-	9	48	-	-	8	48

Beispiel	Strategie 3		implementierte Suche	
	goals	steps	goals	steps
iv-list2 (3.13)	15	126	15	127
graph1* (3.8)	8	23	10	31
graph1 (3.8)	9	23	11	31
graph2* (3.9)	10	55	26	178
graph2 (3.9)	11	56	79	667
hoare1 (3.1)	11	75	12	77
hoare2 (3.2)	8	48	33	234

Abbildung 7.2: Teststrategien für das Formelgewicht

ausgewählt. Die nicht behandelten Beispiele aus Kapitel 3 besitzen nur wenige Variablen, für die eine Belegung gesucht werden muß. Dadurch ergeben sich für die Suchstrategien wenige Entscheidungsmöglichkeiten und die Suchzeiten unterscheiden sich für die verschiedenen Strategien kaum.

Zum Vergleich der Strategien geben wir die Anzahl der benötigten Beweisschritte (*steps*) und die Anzahl der Prämissen (*goals*) an, die während der Gegenbeispiel-Suche entstanden sind. Diese beiden Größen geben im wesentlichen dieselbe Information wie die Suchdauer, sind aber einfacher zu messen. Das Verhältnis der Größen *steps* bzw. *goals* für die verschiedene Strategien spiegelt das Zeitverhältnis wieder, das für die Suche zustande kommt. Werden für Beispiele zusätzlich Formeln getestet, die bei einem Beweisversuch entstanden sind (siehe *graph1* und *graph2*), werden diese mit einem * versehen. Zuerst besprechen wir die Bewertungsfunktion für die Formeln. Danach sehen wir uns die Auswahlstrategie für die Variablen an.

7.2.1 Bewertung der Formelgewichte

Für die Bewertung der Formelgewichte wurden Versuche mit einer *Breitensuche*, *Tiefensuche*, Bewertung der Funktionen (*Strategie 1*), Bewertung der Variablen (*Strategie 2*) und eine Bewertung, die alle Kriterien berücksichtigt (*Strategie 3*) durchgeführt. Die Ergebnisse sind in der Tabelle 7.2 aufgeführt.

Eine faire Suche ist die *Breitensuche*, die die Formeln in der Reihenfolge expandiert, in der sie entstanden sind. Deshalb werden alle Formeln gleich oft expandiert. Da die Beweistiefe stark mit der Anzahl der Expansionen korreliert, wird sie für die Gewichtung gewählt, da dies implementierungstechnisch einfacher zu realisieren ist.

Die Versuche *graph2*, *hoare1* und *hoare2* wurden abgebrochen, da sie nach ca. zehn Minuten noch keine Ergebnisse lieferten. Bei den Beispielen *iv-list2* und *graph1* hat sich die Suchzeit, gegenüber der implementierten Suche, ungefähr verdoppelt und bei dem Beispiel *graph1** sogar verdreifacht. Die Suchzeiten sind deshalb sehr lange, da nur wenige Variablenbelegungen zu einer Lösung führen. Die Breitensuche betrachtet aber alle Belegungen gleich und wählt nicht bevorzugt diejenige aus, die zu einem Gegenbeispiel führt.

Die *Tiefensuche* führt bei den Beispielen *hoare1*, *hoare2* und *iv-list2* sehr schnell zu einer Lösung, da die Lösung genau in dem Ast liegt, der betrachtet wird. Daß die Tiefensuche nicht

fair ist, sieht man an den Beispielen zu den Graphen. Wenn eine Variable g der Sorte **graph** expandiert wird, entstehen die Fälle $g = @_g$, $g = g' +_v v$ und $g = g' +_e e$. Die Tiefensuche betrachtet immer die linken Fälle zuerst. Die Fälle, in denen zum leeren Graphen expandiert wird, liefern keine Lösung und der Beweisast wird geschlossen. Deshalb werden in den Graphen immer nur Knoten eingefügt. Für das Gegenbeispiel benötigen wir aber Kanten, die einen Pfad beschreiben. Aufgrund der Tiefensuche gelangen aber keine Kanten in den Graphen, bzw. die Suche fährt nicht in den Beweisästen fort, in denen Kanten eingefügt worden sind. Deshalb kann kein Gegenbeispiel gefunden werden.

Im nächsten Test, der in der Tabelle unter *Strategie 1* aufgeführt ist, wurden die Formeln nach den Gewichten bewertet, die sich für die Funktionen der Formel aus der Funktionenhierarchie ergeben. Vor allem für die Beispiele *graph2* und die *iv-list2* ergaben sich starke Verbesserungen gegenüber der implementierten Strategie. Jedoch konnten die Beispiele *hoare1* und *hoare2* nicht mehr gelöst werden. Dies liegt daran, daß ein Gegenbeispiel nur für Formeln, die Funktionen enthalten, die weiter oben in der Funktionenhierarchie stehen, existiert. Diese Formeln werden bei der gewählten Suchstrategie aber nicht bevorzugt betrachtet.

Erstaunlich gute Ergebnisse lieferte die *Strategie 2*, bei der nur die Anzahl der freien, erzeugten Variablen als Maßzahl für das Formelgewicht herangezogen wurde. Dies führte bei allen Versuchen schnell zu einer Lösung. Deshalb wurden mit dieser Strategie noch die anderen Beispiele aus Kapitel 3 getestet. Bei dem Beispiel 3.14 (*compiler*) zeigte sich der Effekt, daß überhaupt keine Lösung mehr gefunden werden konnte. Der Grund dafür ist, daß immer wieder Fallunterscheidungen über die booleschen Terme gemacht werden, die der Compiler auswerten soll. Diese können 0, 1, $\neg\ t'$, $t'\ \rightarrow\ t''$ sein. Die Fälle 0 und 1 führen zu keiner Lösung und der Beweisast kann geschlossen werden. Der Term $t = \neg\ t'$ hat weniger Variablen wie $t = t'\ \rightarrow t''$. Deshalb wird in dem Ast weitergesucht, in dem für eine Variable t in der Formel $\neg\ t'$ eingesetzt wird. Da nicht weiter vereinfacht werden kann, wird nun t' expandiert und das Spiel beginnt von neuem. Es werden immer nur Negationen eingeführt, der Fehler entsteht aber, wenn der Compiler eine Implikation übersetzen soll. Deshalb findet diese Strategie für das Beispiel *compiler* keine Lösung.

Die Anzahl der Variablen zu messen, scheint nach dem obigen Test meist eine gute Strategie zu sein. Deshalb wurde in der Strategie, die implementiert wurde, das Gewicht, das die Anzahl der Variablen zu einem Formelgewicht beitragen, stärker bewertet (vier mal so stark wie ursprünglich). Die Ergebnisse stehen in der Abbildung 7.2 unter dem Punkt *Strategie 3*. Bis auf das Beispiel *compiler*, werden alle Beispiele erheblich schneller gelöst.

7.2.2 Auswahl der Variablen für die Expansion

Ein weiterer Punkt, der die Suche nach einem Gegenbeispiel beeinflußt, ist die Auswahl der Variable, die expandiert werden soll. Bei den folgenden Tests wurden immer die in Abschnitt 6.2.3 beschriebene Implementierung der Bewertungsfunktion für das Formelgewicht benutzt. Die Tests *#Expansionen* bevorzugen die Variablen, die bisher am wenigsten oft expandiert wurden und die Tests *rekursive Positionen* bevorzugen die Variablen, die in den Funktionen, die weit oben in der Funktionenhierarchie stehen, auf rekursiven Positionen vorkommen. In den Tests werden die Kriterien für die Auswahl der Variablen einzeln getestet, um die Auswirkungen der Kriterien auf die Gegenbeispiel-Suche zu testen (siehe Tabelle 7.3). Nach den Ergebnissen dieser Tests kommt nur eine gemischte Variante aller Kriterien in Betracht. Deshalb wurden die anderen Kriterien für die Variablenauswahl, die in Abschnitt 5.1.3 beschrieben sind, nicht mehr untersucht.

Betrachtet man nur die Anzahl der Expansionen für die Auswahl der Variablen, so erhält man eine Art Breitensuche auf den Variablen. Alle Variablen werden gleich behandelt. Damit für eine rekursiv über Konstruktoren definierte Funktion die Definition eingesetzt werden kann, muß über die Expansion ein Konstruktor an die rekursive Position der Funktion gebracht werden. Müssen für bestimmte Funktionen öfters die Definitionen eingesetzt werden, um ein Gegenbeispiel zu erhalten, kann die zweite Einsetzung der Definition erst spät erfolgen, nämlich dann, wenn durch die Breitensuche auf den Variablen alle anderen Variablen der Formel expandiert wurden. Deshalb führt diese Strategie selten schnell zum Erfolg.

Ähnlich verhält es sich, wenn die Variablen bevorzugt werden, die auf rekursiven Positionen

	# Expansionen		rekursive Positionen	
	goals	steps	goals	steps
iv-list2 (3.13)	3	19	-	-
graph1* (3.8)	-	-	9	27
graph1 (3.8)	-	-	10	27
graph2* (3.9)	-	-	38	216
graph2 (3.9)	-	-	-	-
hoare1 (3.1)	-	-	-	-
hoare2 (3.2)	-	-	-	-

Abbildung 7.3: Teststrategien für das Variablengewicht

vorkommen. Werden nur diese Variablen expandiert, können Bedingungen für Variablen, die nicht auf rekursive Positionen vorkommen, nicht erfüllt werden und es wird nicht für alle Variablen eine Belegung gesucht.

Aus diesen Tests hat sich ergeben, daß nur eine gemischte Betrachtung aller Kriterien zum Erfolg führen kann. Eine solche Strategie ist in der aktuellen Version der Gegenbeispiel-Suche in KIV implementiert.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Methode entwickelt, mit der man Gegenbeispiele für Formeln in formalen Beweisversuchen finden kann. Wird für eine Formel ein Gegenbeispiel gefunden, kann sie nicht bewiesen werden. Dadurch vermeidet man, daß man unnötige Energie in einen Beweisversuch für eine Formel steckt, die nicht bewiesen werden kann. Durch die Rückverfolgung kann in einem Beweis, der einen Fehler in der Beweisführung enthält, die Stelle ausfindig gemacht werden, an der der Fehler begangen wurde. Ist kein Fehler in der Beweisführung begangen worden, kann die Rückverfolgung das Gegenbeispiel bis zur Konklusio zurückrechnen. Dann ist entweder die zu beweisende Aussage falsch, oder es liegt ein Fehler in der Spezifikation vor.

Bei der Validierung von Spezifikationen können mit Hilfe der Gegenbeispiel-Suche Fehler entdeckt werden, wenn für ein Lemma, das für die Validierung formuliert wurde, ein Gegenbeispiel gefunden wird. Desweiteren kann die Gegenbeispiel-Suche Fehler in Programmen, Lemmata und Beweisversuchen aufdecken. Sie ist damit ein universelles Hilfsmittel für einen Beweisingenieur.

Zuerst untersuchten wir in der Arbeit, ob automatische Theorembeweiser Gegenbeispiele für Formeln, die bei der Beweisarbeit entstehen, finden können. Dabei stellte sich heraus, daß die automatischen Theorembeweiser nicht mit der Schachtelungstiefe der Terme zurecht kamen, die in den Formeln aufgetreten sind. Diese Termstruktur ist aber in Formeln, die während eines Programmbeweises auftreten, die Regel. Deshalb haben wir diesen Ansatz verworfen.

Stattdessen haben wir eine Gegenbeispiel-Suche durch Beweisfortführung entwickelt. Durch die strukturelle Expansion der freien Variablen einer Formel spannen wir einen Suchbaum auf. Jedes Blatt repräsentiert eine (teilweise) Belegung der Variablen der Ausgangsformel. Kann die Formel eines Blattes aufgrund der Belegung zu **false** ausgewertet werden, ist die Existenz eines Gegenbeispiels bestätigt und durch die Rückberechnung erhalten wir ein konkretes Gegenbeispiel für die Ausgangsformel der Gegenbeispiel-Suche. Die Problematik, daß durch die Expansion Formeln entstehen können, die nur noch Aussagen über die Parametersorten oder unterspezifizierten Teilen der Spezifikation enthalten, haben wir dadurch gelöst, daß in diesen Fällen der Benutzer entscheidet, ob ein Gegenbeispiel existiert, oder nicht. Entscheidet der Benutzer, daß die negierte Aussage der Formeln erfüllbar ist, haben wir ein Gegenbeispiel gefunden und die Formel wird als Bedingung mit in das Gegenbeispiel aufgenommen.

Damit die frühestmögliche Stelle für ein Gegenbeispiel in einem Beweis gefunden werden kann, berechnet die Rückverfolgung das Gegenbeispiel nicht nur bis zur Ausgangsformel der Gegenbeispiel-Suche zurück, sondern versucht, das Gegenbeispiel im Originalbeweis weiter zurückzurechnen. Dadurch ist es möglich Beweisfehler zu entdecken oder festzustellen, daß das Lemma falsch ist, für das man den Beweisversuch gestartet hat, obwohl die Gegenbeispiel-Suche nur für eine Prämisse des Beweisbaumes aufgerufen wurde.

Die vorgestellte Methode wurde in das interaktive Beweissystem KIV integriert und an zahlreichen Beispielen erfolgreich getestet. Für die Suchstrategie, die entscheidet, welche Formel als nächstes expandiert wird, wurden noch einige Varianten getestet. Dabei hat sich herausgestellt, daß bei der Gewichtung der Formeln, die Anzahl der freien Variablen einer Formel eine große Rolle spielt. Berücksichtigt man die Anzahl der Variablen bei der Gewichtung der Formeln stark,

verbessert sich die Suchzeit für ein Gegenbeispiel bei den meisten getesteten Beispielen enorm.

Interessant wäre es herauszufinden, ob man der Formel, für die man ein Gegenbeispiel sucht, oder der Spezifikation ansehen kann, wann eine starke Gewichtung der Anzahl der Variablen (oder eine andere Bewertungsstrategie) schnellere Suchzeiten bringt. Damit wäre eine effizientere Suche möglich. In dem Beispiel, das bei der starken Gewichtung der Variablenanzahl langsamer gelöst wird, entdeckt man den Fehler, wenn ein Konstruktor eingesetzt wird, der zwei Parameter benötigt. Deshalb wird dieses Beispiel auch langsamer gelöst, wenn man die Anzahl der Variablen stark gewichtet, denn man muß eine Formel für ein Gegenbeispiel betrachten, die man als schwer zu lösen einstuft. Vielleicht läßt sich daraus ein Kriterium finden, welche der Suchstrategien angewendet werden sollte, um eine schnelle Suche zu ermöglichen.

Ein anderer Punkt, den man noch verbessern könnte, ist die Anzahl der Benutzerinteraktionen bei der Gegenbeispiel-Suche. Eine Benutzerinteraktion tritt bei den Beispielen hauptsächlich dann auf, wenn Aussagen über Parametersorten zu entscheiden sind. Diese Aussagen sind meist von der Form ,,gibt es verschiedene Elemente einer Parametersorte", ,,gibt es ein Element, das kleiner als ein anderes Element der Parametersorte ist" oder ,,gibt es ein Element, das kleiner als ein Defaultelement der Parametersorte ist". Diese Aussagen sind für Parametersorten erfüllbar und man kann dies den Spezifikationen auch ,,ansehen", d.h. ein entsprechender Algorithmus könnte diese Anfragen entscheiden. Würde man diesen Mechanismus in die Gegenbeispiel-Suche integrieren, würde bei den Beispielen aus Kapitel 3 nur noch in einem Fall eine Anfrage an den Benutzer anfallen.

Eine weitere Erweiterung, die man in die Gegenbeispiel-Suche integrieren könnte, wäre, allgemeine Gegenbeispiele zu finden. In der aktuellen Implementierung sind Gegenbeispiele dann nicht konkret, wenn in der Formel Variablen auftreten, für die die Rückverfolgung keine Belegung berechnet hat. Je allgemeiner die Gegenbeispiele jedoch sind, desto mehr Informationen geben sie dem Benutzer, um die Fehlerursache zu finden. Deshalb wäre ein Mechanismus, der allgemeine Gegenbeispiele liefert, sehr nützlich, um die Fehlersuche noch besser zu unterstützen.

Anhang A

Fallstudien

Wir geben hier die Spezifikationen der Beispiele aus Kapitel 3 an. Die abgedruckten Spezifikationen sind automatisch vom KIV-System erstellt worden. Die Bezeichner unterscheiden sich teilweise von denen in Kapitel 3, denn dort sind einige Spezifikationen zur Vereinheitlichung und einfacheren Lesbarkeit angepasst worden.

A.1 Binärsuche auf Arrays

obarray, 85
barray, 85

obarray =
enrich barray **with**
predicates $<_{sor}$: array × nat × nat;
variables m_0: nat;

axioms

$<_{sor}$(a, m, n)
$\leftrightarrow$ n < size(a) $\wedge$ ($\forall$ m_0, n_0. m $\leq$ m_0 $\wedge$ m_0 < n_0 $\wedge$ n_0 $\leq$ n $\rightarrow$ get(a, m_0) $\ll$ get(a, n_0))

end enrich

---

barray =
generic specification
parameter data **using** nat **target**
sorts array;
constants $@_{array}$: array;
functions

mkarray	:	nat	→	array	;
get	:	array × nat	→	data	;
put	:	array × nat × data	→	array	;
size	:	array	→	nat	;

variables a', a: array;

axioms

array **generated by** mkarray, put;
a = a' $\leftrightarrow$ size(a) = size(a') $\wedge$ ($\forall$ n. n < size(a) $\rightarrow$ get(a, n) = get(a', n)),
size(mkarray(n)) = n,

size(put(a, n, d)) = size(a),
get(mkarray(n), m) = □,
m < size(a) → get(put(a, m, d), m) = d,
m ≠ n → get(put(a, n, d), m) = get(a, m),
$@_{array}$ = mkarray(0)

end generic specification

A.2 Mergesort auf Listen

enrlist =
enrich list **with**
functions
del_once : elem × list → list ;
predicates
is_perm : list × list;
. ∈ . : elem × list;

axioms

¬ ele ∈ nil,
ele ∈ ele $+_l$ l,
ele ≠ ele_1 → (ele ∈ ele_1 $+_l$ l ↔ ele ∈ l),
del_once(ele, nil) = nil,
del_once(ele, ele $+_l$ l) = l,
ele ≠ ele_1 → del_once(ele, ele_1 $+_l$ l) = ele_1 $+_l$ del_once(ele, l),
is_perm(nil, nil),
l ≠ nil → ¬ is_perm(nil, l),
is_perm(ele $+_l$ l, l_1) ↔ ele ∈ l_1 ∧ is_perm(l, del_once(ele, l_1))

end enrich

A.3 Reguläre Ausdrücke

match =
enrich rspair-less **with**
predicates
. matches . : regexp × string;
variables s_2, s': string; ss: strings;

axioms

¬ $@_r$ matches s,
eps matches s ↔ s = $@_s$,

mkreg(cr) matches s $\leftrightarrow$ cr $'_s$ = s,
r_1 o r_2 matches s $\leftrightarrow$ ($\exists$ s_1, s_2. s = s_1 ++ s_2 $\wedge$ r_1 matches s_1 $\wedge$ r_2 matches s_2),
r_1 || r_2 matches s $\leftrightarrow$ r_1 matches s $\vee$ r_2 matches s,
** r matches s $\leftrightarrow$ ($\exists$ s_1, s_2. s = s_1 ++ s_2 $\wedge$ r matches s_1 $\wedge$ ** r matches s_2)

end enrich

---

regexp =
data specification
using nat-basic2, string
regexp = $@_r$ **with** $@_{rp}$
| eps **with** epsp
| mkreg (reg : char) **with** regp
| . o . (. .r1 : regexp, . .r2 : regexp) **with** op
| . || . (. .r1 : regexp, . .r2 : regexp) **with** $||_p$
| ** . (. .r1 : regexp) **with** $**_p$
;
variables r: regexp;
size functions $\#_r$: regexp $\rightarrow$ nat ;
order predicates . $\ll_r$. : regexp $\times$ regexp;
end data specification

regexp **freely generated by** $@_r$, eps, mkreg, o, ||, **;

---

string =
actualize flattern **with** char **by morphism**
elem $\rightarrow$ char, list $\rightarrow$ string, llist $\rightarrow$ strings, $\perp$ $\rightarrow$ a, @ $\rightarrow$ $@_s$, $@_l$ $\rightarrow$ $@_{ss}$, $\oplus$ $\rightarrow$ $\oplus_s$, .first $\rightarrow$ first **prio** 0, .rest $\rightarrow$ rest **prio** 0, # $\rightarrow$ $\#_s$, $\odot$ $\rightarrow$ ++, ' $\rightarrow$ $'_s$, .$\oplus$ $\rightarrow$.$\oplus_s$, .last $\rightarrow$ last **prio** 0, .butlast $\rightarrow$ butlast, $\oplus_l$ $\rightarrow$ $\oplus_{ss}$, .firstl $\rightarrow$ .firstss, .restl $\rightarrow$ .restss, $\#_l$ $\rightarrow$ $\#_{ss}$, $\odot_l$ $\rightarrow$ $++_s$, $'_l$ $\rightarrow$ $'_{ss}$, .$\oplus_l$ $\rightarrow$.$\oplus_{ss}$, .lastl $\rightarrow$ .lastss, .butlastl $\rightarrow$ .butlastss, flatten $\rightarrow$ flattens, $@_p$ $\rightarrow$ $@_{ps}$, $<_l$ $\rightarrow$ $\ll_s$, $\in$ $\rightarrow$ $\in_s$, $\sqsubseteq$ $\rightarrow$ $\sqsubseteq_s$, $\sqsupseteq$ $\rightarrow$ $\sqsupseteq_s$, precedes $\rightarrow$ precedes_in, $@_{pl}$ $\rightarrow$ $@_{pss}$, $<_{ll}$ $\rightarrow$ $\ll_{ss}$, $\in_l$ $\rightarrow$ $\in_{ss}$, $\sqsubseteq_l$ $\rightarrow$ $\sqsubseteq_{ss}$, $\sqsupseteq_l$ $\rightarrow$ $\sqsupseteq_{ss}$, precedesl $\rightarrow$ precedesss, a $\rightarrow$ ac, b $\rightarrow$ bc, c $\rightarrow$ cc, a_0 $\rightarrow$ ac_0, x $\rightarrow$ s, y $\rightarrow$ ys, z $\rightarrow$ zs, x_0 $\rightarrow$ s_0, x_1 $\rightarrow$ s_1, al $\rightarrow$ ass, bl $\rightarrow$ bss, cl $\rightarrow$ css, al_0 $\rightarrow$ ass_0, xl $\rightarrow$ xss, yl $\rightarrow$ yss, zl $\rightarrow$ zss, xl_0 $\rightarrow$ xss_0, xl_1 $\rightarrow$ xss_1
end actualize

---

char =
data specification
char = a
| b
| c
| d
;
variables cr: char;
end data specification

char **freely generated by** a, b, c, d;

---

flattern =
enrich list_of_list **with**
functions flatten : llist $\rightarrow$ list ;

axioms

flatten($@_l$) = @,
flatten(al $\oplus_l$ xl) = al $\odot$ flatten(xl)

end enrich

---

list_of_list =
actualize ext-list **with** ext-list **by morphism**
elem $\rightarrow$ list, list $\rightarrow$ llist, $\perp \rightarrow$ @, @ $\rightarrow @_l$, $\oplus \rightarrow \oplus_l$, .first $\rightarrow$ .firstl, .rest $\rightarrow$ .restl, # $\rightarrow \#_l$, $\odot \rightarrow \odot_l$, ' $\rightarrow$ '$_l$, .$\oplus \rightarrow$.$\oplus_l$, .last $\rightarrow$ .lastl, .butlast $\rightarrow$ .butlastl, $@_p \rightarrow @_{pl}$, $<_l \rightarrow <_{ll}$, $\in \rightarrow \in_l$, $\sqsubseteq \rightarrow \sqsubseteq_l$, $\sqsupseteq \rightarrow \sqsupseteq_l$, precedes $\rightarrow$ precedesl, a $\rightarrow$ al, b $\rightarrow$ bl, c $\rightarrow$ cl, $a_0 \rightarrow al_0$, x $\rightarrow$ xl, y $\rightarrow$ yl, z $\rightarrow$ zl, $x_0 \rightarrow xl_0$, $x_1 \rightarrow xl_1$
end actualize

A.4 Graphen

spathp =
enrich pathp **with**
predicates spathp : list $\times$ dirgraph;

axioms

spathp(x, g) $\leftrightarrow$ pathp(x, g) $\wedge \neg$ dups(x)

end enrich

---

pathp =
enrich dirgraph, list-last-dup **with**
predicates pathp : list $\times$ dirgraph;

axioms

$\neg$ pathp(@, g),
pathp(a $\oplus$ @, g),
pathp(a $\oplus$ b $\oplus$ x, g) $\leftrightarrow$ a –> b $\in_e$ g $\wedge$ pathp(b $\oplus$ x, g)

end enrich

---

dirgraph =
enrich edge, nat **with**
sorts dirgraph;
constants $@_g$: dirgraph;
functions

. $+_v$. : dirgraph × elem → dirgraph **prio** 5 **left**;
. $+_e$. : dirgraph × edge → dirgraph **prio** 5 **left**;
. $-_v$. : dirgraph × elem → dirgraph **prio** 5 **left**;
. $-_e$. : dirgraph × edge → dirgraph **prio** 5 **left**;
$\#_v$: dirgraph → nat ;
$\#_e$: dirgraph → nat ;

predicates

. $\in_v$. : elem × dirgraph;
. $\in_e$. : edge × dirgraph;

variables g', g: dirgraph;

axioms

dirgraph **generated by** $@_g$, $+_v$, $+_e$;
g = g' ↔ (∀ a. a $\in_v$ g ↔ a $\in_v$ g') ∧ (∀ e. e $\in_e$ g ↔ e $\in_e$ g'),
¬ a $\in_v$ $@_g$,
a $\in_v$ g $+_v$ b ↔ a = b ∨ a $\in_v$ g,
a $\in_v$ g $+_e$ b –> c ↔ a = b ∨ a = c ∨ a $\in_v$ g,
¬ e $\in_e$ $@_g$,
e $\in_e$ g $+_v$ a ↔ e $\in_e$ g,
e $\in_e$ g $+_e$ e' ↔ e = e' ∨ e $\in_e$ g,
a $\in_v$ g $-_v$ b ↔ a ≠ b ∧ a $\in_v$ g,
a –> b $\in_e$ g $-_v$ c ↔ a ≠ c ∧ b ≠ c ∧ a –> b $\in_e$ g,
a $\in_v$ g $-_e$ e ↔ a $\in_v$ g,
e $\in_e$ g $-_e$ e' ↔ e ≠ e' ∧ e $\in_e$ g,
$\#_v(@_g)$ = 0,
¬ a $\in_v$ g → $\#_v$(g $+_v$ a) = $\#_v$(g)+1,
$\#_v$(g $+_e$ a –> b) = $\#_v$(g $+_v$ a $+_v$ b),
$\#_e(@_g)$ = 0,
$\#_e$(g $+_v$ a) = $\#_e$(g),
¬ e $\in_e$ g → $\#_e$(g $+_e$ e) = $\#_e$(g)+1

end enrich

---

list-last-dup = list-last + list-dup

---

edge =
generic data specification
parameter elem
edge = . –> . (. .src : elem, . .dest : elem);
variables e', e_2, e_1, e_0, e: edge;
end generic data specification

edge **freely generated by** –>;

---

list-dup =
enrich list **with**
predicates dups : list;

axioms

dups(x) ↔ (∃ a, x_0, y, z. x = x_0 ⊙ a ⊕ y ⊙ a ⊕ z)

end enrich

A.5 Binär-Arithmetik

bin, 90
bin-data, 90

bin =
enrich bin-data **with**
functions . top : bin → bin ;

axioms

_0 top = _0,
_1 top = _1,
x.0 top = _0,
x.1 top = _1

end enrich

---

bin-data =
data specification
bin = _0
| _1
| . .0 (. pop : bin)
| . .1 (. pop : bin)
;
variables y, x: bin;
order predicates . ≪ . : bin × bin;
end data specification

bin **freely generated by** _0, _1, .0, .1;

A.6 Intervall-Listen

asc, 90
intervallist, 90
interval, 91

asc =
enrich intervallist **with**
predicates asc : intervallist;
variables m_2, m_1: nat;

axioms

asc(ilnil),
asc(mkiv(m_1, n_1) $+_{il}$ ilnil) ↔ ¬ $n_1 < m_1$,
asc(mkiv(m_1, n_1) $+_{il}$ mkiv(m_2, n_2) $+_{il}$ il)
↔ ¬ $n_1 < m_1 \wedge n_1 + 1 < m_2 \wedge$ asc(mkiv(m_2, n_2) $+_{il}$ il)

end enrich

---

intervallist =
actualize list **with** interval **by morphism**

elem $\rightarrow$ interval, list $\rightarrow$ intervallist, nil $\rightarrow$ ilnil, $+_l \rightarrow +_{il}$, car $\rightarrow$ ilcar, cdr $\rightarrow$ ilcdr, $\ll \rightarrow \ll_{il}$, ele $\rightarrow$ iv, $\text{ele}_0 \rightarrow \text{iv}_0$, l $\rightarrow$ il, $\text{l}_0 \rightarrow \text{il}_0$, $\text{l}_1 \rightarrow \text{il}_1$
end actualize

---

interval =
actualize pair **with** natbasic2 **by morphism**
elem' $\rightarrow$ nat, elem" $\rightarrow$ nat, pair $\rightarrow$ interval, mkpair $\rightarrow$ mkiv, .p1 $\rightarrow$ .1, .p2 $\rightarrow$ .2, ele' $\rightarrow \text{n}_0$, ele" $\rightarrow \text{n}_1$, $\text{ele'}_0 \rightarrow \text{n}_2$, $\text{ele"}_0 \rightarrow \text{n}_3$, pr $\rightarrow$ iv
end actualize

A.7 Compiler

systemspec = eval + compile + machine

---

compile =
enrich bterm, code **with**
functions compile : bterm $\rightarrow$ list ;

axioms

$\text{compile}(0) = \text{ld}_0 \oplus @$,
$\text{compile}(1) = \text{ld}_1 \oplus @$,
$\text{compile}(\neg\text{' t}) = \text{compile(t)} \odot \text{inv} \oplus @$,
$\text{compile}(\text{t}_1 \rightarrow\text{' t}_2) = \text{compile}(\text{t}_1) \odot \text{compile}(\text{t}_2) \odot \text{imp} \oplus @$

end enrich

---

eval =
enrich bterm **with**
functions val : bterm $\rightarrow$ bterm ;

axioms

$\text{val}(0) = 0$,
$\text{val}(1) = 1$,
$\text{val(t)} = 0 \rightarrow \text{val}(\neg\text{' t}) = 1$,
$\text{val(t)} = 1 \rightarrow \text{val}(\neg\text{' t}) = 0$,
$\text{val}(\text{t}_1) = 1 \wedge \text{val}(\text{t}_2) = 0 \rightarrow \text{val}(\text{t}_1 \rightarrow\text{' t}_2) = 0$,
$\text{val}(\text{t}_1) = 0 \vee \text{val}(\text{t}_2) = 1 \rightarrow \text{val}(\text{t}_1 \rightarrow\text{' t}_2) = 1$

end enrich

---

machine =
enrich code, stack **with**
constants init : stack;
functions run : list $\times$ stack $\rightarrow$ stack ;

axioms

$\text{init} = \bot$,
$\text{run}(@, s) = s$,
$\text{run}(\text{ld}_0 \oplus x, s) = \text{run}(x, \text{push}(0, s))$,
$\text{run}(\text{ld}_1 \oplus x, s) = \text{run}(x, \text{push}(1, s))$,
$\text{top}(s) = 0 \to \text{run}(\text{inv} \oplus x, s) = \text{run}(x, \text{push}(1, \text{pop}(s)))$,
$\text{top}(s) = 1 \to \text{run}(\text{inv} \oplus x, s) = \text{run}(x, \text{push}(0, \text{pop}(s)))$,
$\text{top}(s) = 0 \vee \text{top}(\text{pop}(s)) = 1 \to \text{run}(\text{imp} \oplus x, s) = \text{run}(x, \text{push}(1, \text{pop}(\text{pop}(s))))$,
$\text{top}(s) = 1 \wedge \text{top}(\text{pop}(s)) = 0 \to \text{run}(\text{imp} \oplus x, s) = \text{run}(x, \text{push}(0, \text{pop}(\text{pop}(s))))$

end enrich

---

code =
enrich list **with**
functions . $\odot$. : list $\times$ list $\to$ list ;

axioms

$@ \odot x = x$,
$(i \oplus x) \odot y = i \oplus x \odot y$

end enrich

---

stack =
enrich bterm **with**
sorts stack;
constants $\bot$: stack;
functions

push	:	bterm $\times$ stack	$\to$	stack	;
pop	:	stack	$\to$	stack	;
top	:	stack	$\to$	bterm	;

variables s: stack;

axioms

stack **generated by** $\bot$, push;
$\bot \neq \text{push}(t, s)$,
$\text{pop}(\text{push}(t, s)) = s$,
$\text{top}(\text{push}(t, s)) = t$

end enrich

---

bterm =
data specification
bterm = 0
| 1
| ¬' . (. .1 : bterm)
| . →' . (. .1 : bterm, . .2 : bterm)
;
variables t: bterm;
end data specification

bterm **freely generated by** 0, 1, ¬', →';

instruction =
data specification
instruction = ld_0
| ld_1
| inv
| imp
;
variables i: instruction;
end data specification

instruction **freely generated by** ld_0, ld_1, inv, imp;

Anhang B

COSI Regeln

Hier werden die COSI-Regeln (computer simulation) beschrieben. Diese Regeln werden für die Beweise in KIV benutzt. Die Regeln sind nicht nur auf die erste Formel im Antezedenten bzw. Sukzedenten anwendbar, sondern auf jede Formel.

Der Abschnitt *Programm Regeln* enthält alle Regeln, die mit Programmen zu tun haben, der Abschnitt *DL Regeln* alle anderen Regeln für Sequenzen die Programme enthalten können und Regeln für *while*-Schleifen sind im Abschnitt *while Regeln* aufgeführt.

B.1 Programm Regeln

call right

$$\frac{\Gamma \vdash \langle \text{proc } \delta \text{ in } \alpha \frac{\underline{\sigma},\underline{x}}{\underline{y},\underline{z}} \rangle \varphi, \Delta}{\Gamma \vdash \langle \text{proc } \delta \text{ in } f(\underline{\sigma},\underline{x}) \rangle \varphi, \Delta}$$

δ enthält eine Deklaration $f(\underline{y},\underline{z}).\alpha$

call left

$$\frac{\langle \text{proc } \delta \text{ in } \alpha \frac{\underline{\sigma},\underline{x}}{\underline{y},\underline{z}} \rangle \varphi, \Gamma \vdash \Delta}{\langle \text{proc } \delta \text{ in } f(\underline{\sigma},\underline{x}) \rangle \varphi, \Gamma \vdash \Delta}$$

δ enthält eine Deklaration $f(\underline{y},\underline{z}).\alpha$

execute call

$$\frac{\Gamma \vdash \underline{\sigma} = \underline{\tau} \qquad \langle \text{proc } \delta_1 \text{ in } f(\underline{\sigma};\ \underline{x}) \rangle (\underline{x} = \underline{y}),\ \varphi^{\underline{y}}_{\underline{x}}, \Gamma \vdash \psi^{\underline{y}}_{\underline{z}}}{\langle \text{proc } \delta_1 \text{ in } f(\underline{\sigma};\ \underline{x}) \rangle \varphi, \Gamma \vdash \langle \text{proc } \delta_2 \text{ in } f(\underline{\tau};\ \underline{z}) \rangle \psi}$$

- δ_1 und δ_2 enthalten identische Deklarationen für f
- $\underline{y}$ sind neue Variablen

contract call left

$$\frac{\Gamma \vdash \underline{\sigma} = \underline{\tau} \qquad \langle \text{proc } \delta_1 \text{ in } f(\underline{\sigma},\underline{z}) \rangle (\underline{z} = \underline{x}'), \varphi^{\underline{x}'}_{\underline{x}}, \psi^{\underline{x}'}_{\underline{y}}, \Gamma \vdash \Delta}{\langle \text{proc } \delta_1 \text{ in } f(\underline{\sigma},\underline{x}) \rangle \varphi, \langle \text{proc } \delta_2 \text{ in } f(\underline{\tau},\underline{y}) \rangle \psi, \Gamma \vdash \Delta}$$

- δ_1 und δ_2 enthalten identische Deklarationen für f
- $\underline{z}$, $\underline{x}'$ sind neue Variablen

contract call right

$$\frac{\Gamma \vdash \underline{\sigma} = \underline{\tau} \qquad [\text{proc } \delta_1 \text{ in } f(\underline{\sigma}, \underline{z})](\underline{z} = \underline{x}'), \Gamma \vdash \varphi^{\underline{x}'}_{\underline{x}}, \psi^{\underline{x}'}_{\underline{y}}, \Delta}{\Gamma \vdash \langle \text{proc } \delta_1 \text{ in } f(\underline{\sigma}, \underline{x})\rangle\varphi, \langle \text{proc } \delta_2 \text{ in } f(\underline{\tau}, \underline{y})\rangle\psi, \Delta}$$

- δ_1 und δ_2 enthalten identische Deklarationen für f
- $\underline{z}$, $\underline{x}'$ sind neue Variablen

abort right

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \langle \text{abort}\rangle\varphi, \Delta} \qquad \frac{}{\Gamma \vdash [\text{abort}]\varphi, \Delta}$$

abort left

$$\frac{}{\langle \text{abort}\rangle\varphi, \Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \varphi, \Delta}{[\text{abort}]\varphi, \Gamma \vdash \Delta}$$

skip right/skip left

$$\frac{\Gamma \vdash \varphi, \Delta}{\Gamma \vdash \langle \text{skip}\rangle\varphi, \Delta} \qquad \frac{\varphi, \Gamma \vdash \Delta}{\langle \text{skip}\rangle\varphi, \Gamma \vdash \Delta}$$

vardecls right

$$\frac{x' = \tau, \Gamma \vdash \langle \alpha^{x'}_{x}\rangle\varphi}{\Gamma \vdash \langle \text{var } x = \tau \text{ in } \alpha\rangle\varphi} \qquad \frac{\Gamma \vdash \exists\, x'.\langle \alpha^{x'}_{x}\rangle\varphi}{\Gamma \vdash \langle \text{var } x =? \text{ in } \alpha\rangle\varphi}$$

- $x =?$ ist eine zufällige Variablendeklaration
- beide Regeln sind in einer zusammengefaßt

vardecls left

$$\frac{\langle \alpha^{x'}_{x}\rangle\varphi, x' = \tau, \Gamma \vdash \Delta}{\langle \text{var } x = \tau \text{ in } \alpha\rangle\varphi, \Gamma \vdash \Delta} \qquad \frac{\exists\, x'.\langle \alpha^{x'}_{x}\rangle\varphi, \Gamma \vdash \Delta}{\langle \text{var } x =? \text{ in } \alpha\rangle\varphi, \Gamma \vdash \Delta}$$

- $x =?$ ist eine zufällige Variablendeklaration
- beide Regeln sind in einer zusammengefaßt

assign right

$$\frac{\Gamma \vdash \varphi^{\tau}_{x}, \Delta}{\Gamma \vdash \langle x := \tau\rangle\varphi, \Delta} \qquad \frac{\Gamma^{x'}_{x}, x = \tau^{x'}_{x} \vdash \varphi, \Delta^{x'}_{x}}{\Gamma \vdash \langle x := \tau\rangle\varphi, \Delta}$$

- x' ist eine neue Variable
- die zweite Regel wird nur benutzt, wenn die erste nicht anwendbar ist (da es nicht möglich ist in φ zu substituieren)

assign left

$$\frac{\varphi_x^\tau, \Gamma \vdash \Delta}{\langle x := \tau\rangle\varphi, \Gamma \vdash \Delta} \qquad \frac{\varphi, \Gamma_x^{x'}, x = \tau_x^{x'} \vdash \Delta_x^{x'}}{\langle x := \tau\rangle\varphi, \Gamma \vdash \Delta}$$

if right/if left

$$\frac{\Gamma, \varepsilon \vdash \langle\alpha\rangle\varphi, \Delta \qquad \Gamma, \neg\varepsilon \vdash \langle\beta\rangle\varphi, \Delta}{\Gamma \vdash \langle\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Delta} \qquad \frac{\langle\alpha\rangle\varphi, \Gamma, \varepsilon \vdash \Delta \qquad \langle\beta\rangle\varphi, \Gamma, \neg\varepsilon \vdash \Delta}{\langle\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Gamma \vdash \Delta}$$

if positive right/if positive left

$$\frac{\Gamma \vdash \varepsilon \qquad \Gamma \vdash \langle\alpha\rangle\varphi, \Delta}{\Gamma \vdash \langle\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Delta} \qquad \frac{\Gamma \vdash \varepsilon \qquad \langle\alpha\rangle\varphi, \Gamma \vdash \Delta}{\langle\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Gamma \vdash \Delta}$$

if negative right/if negative left

$$\frac{\Gamma \vdash \neg\varepsilon \qquad \Gamma \vdash \langle\beta\rangle\varphi, \Delta}{\Gamma \vdash \langle\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Delta} \qquad \frac{\Gamma \vdash \neg\varepsilon \qquad \langle\beta\rangle\varphi, \Gamma \vdash \Delta}{\langle\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Gamma \vdash \Delta}$$

split left/split right

$$\frac{\langle\alpha\rangle\underline{x} = \underline{x}', \varphi_{\underline{x}}^{\underline{x}'}, \Gamma \vdash \Delta}{\langle\alpha\rangle\varphi, \Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \langle\alpha\rangle true \qquad \langle\alpha\rangle\underline{x} = \underline{x}', \Gamma \vdash \varphi_{\underline{x}}^{\underline{x}'}, \Delta}{\Gamma \vdash \langle\alpha\rangle\varphi, \Delta}$$

- $\underline{x} = \text{asgvars}(\alpha)$
- $\underline{x}'$ sind neue Variablen

B.2 DL Regeln

induction

$$\frac{\Gamma \vdash \Theta(\varphi), \Delta \qquad \Gamma, \Theta(\psi) \vdash \Delta \qquad \varphi, \text{Ind-Hyp} \vdash \psi}{\Gamma \vdash \Delta}$$

- $\text{Ind-Hyp} = \forall \underline{x}, u'.(u' \ll u \to (\varphi \to \psi)_u^{u'})$
- $\underline{x} = \text{free}(\varphi \to \psi) \setminus \{u\}$
- u ist die Induktiosvariable
- Θ ist eine parallele Substitution für $u, \underline{x}$

apply induction

$$\frac{\Gamma \vdash \Theta(u') \ll u, \Delta \qquad \Gamma \vdash \Theta(\varphi), \Delta \qquad \Theta(\psi), \Gamma, \text{Ind-Hyp} \vdash \Delta}{\Gamma, \text{Ind-Hyp} \vdash \Delta}$$

- $\text{Ind-Hyp} = \forall \underline{x}, u'.(u' \ll u \to (\varphi \to \psi))$
- Θ ist eine parallele Substitution für $\underline{x}$

structural induction

$$\frac{\vdash \varphi(c) \qquad \varphi(x) \vdash \varphi(f(x))}{\Gamma \vdash \Delta}$$

- $\varphi = \forall\, x'.\Gamma \to \Delta$ with $x' = \mathit{free}(\Gamma \to \Delta) \setminus x$
- $\mathit{sort}(x)$ generated by c, f

all left/exists right

$$\frac{[\alpha]\varphi, \forall\, \underline{x}.\varphi, \Gamma \vdash \Delta}{\forall\, \underline{x}.\varphi, \Gamma \vdash \Delta} \qquad\qquad \frac{\Gamma \vdash \langle\alpha\rangle\varphi, \exists\, \underline{x}.\varphi, \Delta}{\Gamma \vdash \exists\, \underline{x}.\varphi, \Delta}$$

- asgvars$(\alpha) \subseteq \underline{x}$
- es ist auch möglich, eine parallele Substitution für $\underline{x}$ anstatt eines Programms α anzugeben.

exists left/all right

$$\frac{\varphi^{\underline{x}'}_{\underline{x}}, \Gamma \vdash \Delta}{\exists\, \underline{x}.\varphi, \Gamma \vdash \Delta} \qquad\qquad \frac{\Gamma \vdash \varphi^{\underline{x}'}_{\underline{x}}, \Delta}{\Gamma \vdash \forall\, \underline{x}.\varphi, \Delta}$$

- $\underline{x'}$ sind neue Variablen

insert lemma/insert spec-lemma

$$\frac{\Gamma' \vdash \Delta' \qquad \Gamma \vdash \Theta(\Gamma'), \Delta \qquad \Theta(\Delta'), \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$$

- $\Gamma' \vdash \Delta'$ ist das Lemma
- Θ ist eine parallele Substitution für die freien Variablen des Lemmas.

insert elim lemma

$$\frac{\Gamma' \vdash \Delta', \sigma(\varphi) \wedge \bigwedge \sigma(\Gamma) \qquad \sigma(\psi), \Theta(\Gamma') \vdash \Theta(\Delta')}{\Gamma' \vdash \Delta'}$$

- $\Gamma \vdash \varphi \to (x_1 = t_1 \wedge \ldots x_n = t_n \leftrightarrow \exists\, \underline{y}.v = t \wedge \psi)$ ist die elimination-Regel
- σ ist eine Substitution für die freien Variablen der elimination-Regel
- Θ ist die Substitution $[\sigma(t_1), \ldots, \sigma(t_n) \leftarrow \sigma(x_1), \ldots, \sigma(x_n)][\sigma(v) \leftarrow \sigma(t)]$

insert rewrite lemma

$$\frac{\vdash \varphi \to \sigma = \tau \qquad \Gamma \vdash \Theta(\varphi), \Delta \qquad \Gamma^{\Theta(\tau)}_{\Theta(\sigma)} \vdash \Delta^{\Theta(\tau)}_{\Theta(\sigma)}}{\Gamma \vdash \Delta}$$

- $\vdash \varphi \to \sigma = \tau$ ist das rewrite-Lemma
- Θ ist eine parallele Substitution für die freien Variablen des Lemmas

weakening

$$\frac{\Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}$$

cut formula

$$\frac{\varphi, \Gamma \vdash \Delta \qquad \Gamma \vdash \varphi, \Delta}{\Gamma \vdash \Delta}$$

- φ ist die cut-Formel

case distinction

$$\frac{\psi_1, \Gamma \vdash \Delta \qquad \psi_2, \Gamma \vdash \Delta}{\psi_1 \vee \psi_2, \Gamma \vdash \Delta}$$

- die Regel ist auch für $\rightarrow$, $\leftrightarrow$ und für Formeln im Sukzedenten mit $\wedge$, $\leftrightarrow$ anwendbar (man erhält natürlich eine entsprechende Prämisse)

insert equation

$$\frac{\sigma = \tau, \Gamma^{\tau}_{\sigma} \vdash \Delta^{\tau}_{\sigma}}{\sigma = \tau, \Gamma \vdash \Delta}$$

simplifier

$$\frac{\Gamma_1, \Gamma_2' \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta}$$

- Die Formlen Γ_2 werden vereinfacht

B.3 While Regeln

while right

$$\frac{\Gamma \vdash \mathrm{Inv}, \Delta \qquad \mathrm{Inv}, \varepsilon \vdash [\alpha]\mathrm{Inv}, \Delta \qquad \mathrm{Inv}, \neg\varepsilon \vdash \phi, \Delta}{\Gamma \vdash [\textbf{while } \varepsilon \textbf{ do } \alpha]\phi, \Delta}$$

- Inv ist die Schleifeninvariante

Anhang C

Basisregeln der Dynamischen Logik

Alle griechischen Buchstaben, die in den Regeln auftreten sind Metavariablen. Um die Lesbarkeit zu erhöhen wird auf die $-Notation verzichtet, d. h. um die Conclusio der Regel ax einzugeben, die hier in der Form $\phi, \Gamma \vdash \phi, \Delta$ wiedergegeben wird, muß man

`%"$phi, $gamma |- $phi, $delta"` schreiben.

C.1 Axiom

$$\frac{}{\phi, \Gamma \vdash \phi, \Delta} \text{ (ax)}$$

C.2 Gleichheit

$$\frac{}{\vdash \tau = \tau} \text{ (reflexivity)}$$

$$\frac{}{\vdash \sigma = \tau \rightarrow \tau = \sigma} \text{ (symmetry)}$$

$$\frac{}{\vdash \sigma = \tau \rightarrow (\tau = \tau' \rightarrow \sigma = \tau')} \text{ (transitivity)}$$

$$\frac{}{\vdash \sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n \rightarrow f(\sigma_1, \ldots, \sigma_n) = f(\tau_1, \ldots, \tau_n)} \text{ (asg4g)}$$

für jedes Funktionssymbol f der Signatur.

$$\frac{}{\vdash \sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n \rightarrow p(\sigma_1, \ldots, \sigma_n) \leftrightarrow p(\tau_1, \ldots, \tau_n)} \text{ (asg5g)}$$

für jedes Prädikatsymbol p der Signatur.

C.3 Zähler

$$\frac{}{\vdash \neg \text{czero} = \text{csucc}(\$z)} \text{ (minimal\_element)}$$

$$\frac{}{\vdash \mathrm{csucc}(\$z_1) = \mathrm{csucc}(\$z_2) \rightarrow \$z_1 = \$z_2}$$ (injectivity_of_suc)

$$\frac{}{\vdash \neg \mathrm{cless}(\$z, \$z)}$$ (irreflexivity)

$$\frac{}{\vdash \mathrm{cless}(\$z, \mathrm{csucc}(\$z))}$$ (monotonicity)

$$\frac{}{\vdash \mathrm{cless}(\$z_1, \$z_2) \rightarrow (\mathrm{cless}(\$z_2, \$z_3) \rightarrow \mathrm{cless}(\$z_1, \$z_3))}$$ (transitivity_of_counter)

$$\frac{}{\vdash \phi(\mathrm{czero}) \rightarrow ((\forall\, z_1.\phi(z_1) \rightarrow \phi(\mathrm{csucc}(z_1))) \rightarrow \forall\, z.\phi(z))}$$ (induction)

z_1 neu

C.4 Programm Axiome

$$\frac{}{\vdash \langle\alpha\rangle\phi \leftrightarrow \neg[\alpha]\neg\phi}$$ (duality)

$$\frac{}{\vdash [\alpha](\phi \rightarrow \psi) \rightarrow ([\alpha]\phi \rightarrow [\alpha]\psi)}$$ (terminal_implication)

C.5 Spezielle Programm Axiome

$$\frac{}{\vdash [\textbf{skip }]\phi \leftrightarrow \phi}$$ (skip)

$$\frac{}{\vdash [\textbf{abort }]\phi}$$ (abort)

$$\frac{}{\vdash [\alpha;\ \beta]\phi \leftrightarrow [\alpha][\beta]\phi}$$ (composition)

$$\frac{}{\vdash [\textbf{if }\ \varepsilon\ \textbf{ then }\ \alpha\ \textbf{ else }\ \beta]\phi \leftrightarrow ((\varepsilon\ \rightarrow\ [\alpha]\phi) \wedge (\neg\varepsilon\ \rightarrow\ [\beta]\phi))}$$ (conditional)

$$\frac{}{\vdash [x := \tau]\phi \vee [x := \tau]\neg\phi}$$ (determinism_of_assignment)

$$\frac{}{\vdash [x := \tau]\varepsilon \leftrightarrow \varepsilon_x^\tau}$$ (asg1g)

$$\frac{}{\vdash \sigma = \tau \rightarrow ([x := \sigma]\phi \leftrightarrow [x := \tau]\phi)}$$ (substitutivity_of_assignment)

$$\frac{}{\vdash [x := \sigma][x := \tau]\phi \leftrightarrow [x := \tau_x^\sigma]\phi}$$ (asg2g)

$$\frac{}{\vdash [x := \sigma][x := \tau]\phi \leftrightarrow [x := \tau_x^\sigma][x := \sigma]\phi}$$ (asg3g)

wenn $y \notin Var(x := \sigma)$.

C.6 Quantorenverschiebung und gebundenes Umbenennen

$$\frac{}{\vdash (\forall\, v_1.[v_2 := v_1]\phi) \rightarrow \forall\, v_2.\phi} \text{ (renaming)}$$

falls $v1 \notin \{v2\} \cup \mathit{Frei}(\phi)$

$$\frac{}{\vdash (\forall\, v_1.[v_2 := \tau]\phi) \rightarrow [v_2 := \tau] \forall\, v_1.\phi} \text{ (shift)}$$

falls $v_1 \notin \mathit{Var}(v_2 := \tau)$.

C.7 Struktur Regeln

$$\frac{\phi, \phi, \Gamma \vdash \Delta}{\phi, \Gamma \vdash \Delta} \text{ (contraction\_left)} \qquad \frac{\Gamma \vdash \phi, \phi, \Delta}{\Gamma \vdash \phi, \Delta} \text{ (contraction\_right)}$$

$$\frac{\Gamma \vdash \Delta}{\phi, \Gamma \vdash \Delta} \text{ (weakening\_left)} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta} \text{ (weakening\_right)}$$

$$\frac{\phi \vdash \Delta}{\phi, \Gamma \vdash \Delta} \text{ (weakening\_left\_fl)} \qquad \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi, \Delta} \text{ (weakening\_right\_fl)}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\phi, \Gamma \vdash \Delta} \text{ (leftrotate\_left)} \qquad \frac{\Gamma \vdash \Delta, \phi}{\Gamma \vdash \phi, \Delta} \text{ (leftrotate\_right)}$$

$$\frac{\phi, \Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta} \text{ (rightrotate\_left)} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \Delta, \phi} \text{ (rightrotate\_right)}$$

$$\frac{\psi, \phi, \Gamma \vdash \Delta}{\phi, \psi, \Gamma \vdash \Delta} \text{ (leftexchange\_left)} \qquad \frac{\Gamma \vdash \psi, \phi, \Delta}{\Gamma \vdash \phi, \psi, \Delta} \text{ (leftexchange\_right)}$$

$$\frac{\Gamma, \psi, \phi \vdash \Delta}{\Gamma, \phi, \psi \vdash \Delta} \text{ (rightexchange\_left)} \qquad \frac{\Gamma \vdash \Delta, \psi, \phi}{\Gamma \vdash \Delta, \phi, \psi} \text{ (rightexchange\_right)}$$

C.8 Aussagenlogik

$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta} \text{ (con\_l)} \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \text{ (con\_r)}$$

$$\frac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta} \text{ (dis\_l)} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta} \text{ (dis\_r)}$$

$$\frac{\Gamma \vdash \phi, \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \rightarrow \psi, \Gamma \vdash \Delta} \text{ (imp\_l)} \qquad \frac{\phi, \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta} \text{ (imp\_r)}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\neg\phi, \Gamma \vdash \Delta} \text{ (neg\_l)} \qquad \frac{\psi, \Gamma \vdash \Delta}{\Gamma \vdash \neg\psi, \Delta} \text{ (neg\_r)}$$

Anmerkung: es ist praktisch (für den Menschen, nicht für das System), die letzten zwei Regel auch in folgender Gestalt formuliert zu haben (das System kennt keine *neg_l_1* und *neg_r_1* Regeln):

$$\frac{\Gamma \vdash \neg\phi, \Delta}{\phi, \Gamma \vdash \Delta}\ (\text{neg\_l\_1}) \qquad \frac{\neg\psi, \Gamma \vdash \Delta}{\Gamma \vdash \psi, \Delta}\ (\text{neg\_r\_1})$$

$$\frac{\Gamma \vdash \phi, \Delta \quad \phi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ (\text{cut})$$

$$\frac{\Gamma \vdash \phi, \psi, \Delta \quad \phi, \psi, \Gamma \vdash \Delta}{\phi \leftrightarrow \psi, \Gamma \vdash \Delta}\ (\text{equiv\_l}) \qquad \frac{\phi, \Gamma \vdash \psi, \Delta \quad \psi, \Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \leftrightarrow \psi, \Delta}\ (\text{equiv\_r})$$

$$\frac{}{\textbf{false}\ , \Gamma \vdash \Delta}\ (\text{false\_l}) \qquad \frac{}{\Gamma \vdash \textbf{true}\ , \Delta}\ (\text{true\_l})$$

C.9 Quantoren Regeln

Beachte: ϕ_v^τ bedeutet die Substitution für v mit τ in ϕ.

$$\frac{\phi_v^\tau, \Gamma \vdash \Delta}{\forall\, v.\phi, \Gamma \vdash \Delta}\ (\text{all\_left}\ v\ \tau\ \phi) \qquad \frac{\Gamma \vdash \phi_v^\tau, \Delta}{\Gamma \vdash \exists\, v.\phi, \Delta}\ (\text{ex\_right}\ v\ \tau\ \phi)$$

$$\frac{[\alpha]\phi, \Gamma \vdash \Delta}{\forall\, v.\phi, \Gamma \vdash \Delta}\ \text{all\_l if}\ \mathit{asg}(\alpha) \cap \mathit{free}(\phi) \subseteq v \qquad \frac{\Gamma \vdash \langle\alpha\rangle\phi, \Delta}{\Gamma \vdash \exists\, v.\phi, \Delta}\ \text{ex\_r if}\ \mathit{asg}(\alpha) \cap \mathit{free}(\phi) \subseteq v$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall\, v.\phi, \Delta}\ (\text{all\_r }) \text{ if } v \cap (\mathit{Free}(\Gamma) \cup \mathit{Free}(\Delta)) = \varnothing$$

$$\frac{\phi, \Gamma \vdash \Delta}{\exists\, v.\phi, \Gamma \vdash \Delta}\ (\text{ex\_l}) \text{ if } v \cap (\mathit{Free}(\Gamma) \cup \mathit{Free}(\Delta)) = \varnothing$$

C.10 Strukturelle Induktion

$$\frac{}{(\forall\, u.(\forall\, u'.u' < u \rightarrow [u := u']\phi) \rightarrow \phi) \rightarrow \forall\, u.\phi}\ \text{induction\_on\_terms}$$

$<$ ist eine Ordnungsrelation für eine beliebige Sorte und u' ist neu. Die Regeln *asg1g*, ... , *asg5g* und *induction_on_terms* sind Regelgeneratoren.

Anhang D

Implementierung

Die erstellten Dateien für die Implementierung der Gegenbeispiel-Suche befinden sich in dem Verzeichnispfad $\sim kiv/v3/ppl$.

../service/constr-cut.fct In dieser Datei sind die Hilfsfunktionen für die A^*-Suche implementiert. Dazu gehören die Bewertungsfunktionen für Formeln und Variablen und die Funktionen, die die notwendigen Informationen für die Bewertungsfunktionen zur Verfügung stellen.

../rules/constructor-cut Die Expansion der Variablen und die Auswahl der *constructor cut 1*-Regel oder *constructor cut 2*-Regel wird in dieser Datei beschrieben.

../heuristics/constructor-cut Die Auswahl der Variablen für die Expansion durch die *constructor cut*-Heuristik wird in dieser Datei implementiert.

../heuristics/give-value Die *give value*-Heuristik, die den einzelnen Prämissen eines Beweises die Forelgewichte zuordnet, wird in dieser Datei implementiert.

../heuristics/switch-goal Diese Datei implementiert die *switch goal*-Heuristik, die die Formel auswählt, die als nächstes expandiert wird.

../heuristics/weak-quant Das Entfernen überflüssigen Quantoren durch die *quantifier elimination*-Heuristik wird in dieser Datei beschrieben.

../heuristics/rewrite Die *rewrite*-Heuristik, die zusätzlich zu den Simplifier-Regln noch Axiom zur Vereinfachung einer Formel anwendet, ist dort implementiert.

../heuristics/cut In dieser Datei ist die *axiom cut*-Heuristik beschrieben, die zusätzlich zu den *(weak) cut*-Heuristik noch *cut*-Regel aus den Axiomen berechnet.

../simplifier/strong-pl-simplifier Diese Datei beschreibt den Algorithmus, der dafür sorgt, daß der Simplifier während der Gegenbeispiel-Suche nur mit dem reduzierten Regel-Satz ohne die abschwächenden Implikationsregeln aufgerufen wird.

../cntrex/cntrex In dieser Datei wird die Rückverfolgung und Validierung des Gegenbeispiels implementiert. Da die Rückverfolgung durch entstehende Beweisverpflichtungen, für die ein Unterbeweis gestartet werden muß, eng mit der Oberflächensteuerung verknüpft ist, ist die Oberflächensteuerung auch in dieser Datei integriert.

Desweiteren sind die für die Gegenbeispiel-Suche benötigeten Datentypen wie üblich in der Datei **../types.ppl** festgelegt und die *update*-Funktion für die *constructor cut*-Regel befindet sich in der Datei **../system/update-functions**.

Literaturverzeichnis

[Bal97] Balser, M.: *Wiederverwendung von Beweisen nach Modifikation.* Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 1997.

[Kum92] Kumar, V.: *Algorithms for Constraint Satisfaction Problems: A survey.* AI Magazine, 13:32–44, 1992.

[Lö94] Lötzbeyer, A.: *Strukturerhaltende Simplifikation prädikatenlogischer Formeln.* Diplomarbeit, Universität Karlsruhe, 1994.

[Par93] Partsch, H.: *Formal Problem Specification on an Algebraic Basis.* In: Schumann, S., B. Möller und H. Partsch (Herausgeber): *Formal Program Development*, Band 755 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Pro92] Protzen, M.: *Disproving Conjectures.* In: Kapar, D. (Herausgeber): *Automated Deduction – CADE 11*, Band 607 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Rei96] Reif, Wolfgang: *Vorlesung: Beweisbar korrekte Software*, 1996.

[RN95] Russell, S. und P. Norwig: *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[RS93] Reif, W. und K. Stenzel: *Reuse of Failed Proofs in Software Verifikation.* In: *Conference on Foundations of Software Technology and Theoretical Computer Science*. Shyamasundar, 1993.

[RS98] Reif, W. und G. Schellhorn: *Theorem Proving in Large Theories.* In: Bibel, W. und P. Schmitt (Herausgeber): *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, 1998. (to appear).

[Sch96] Schäfer, Dorothea: *PROTEIN, A PROver with a Theory Extention INterface*, November 1996.

[SS98] Schellhorn, Gerhard und Kurt Stenzel: *Praktikum Programmverifikation.* Universität Ulm, Fakultät für Informatik, 1998. Practical course documentation.

[Wei97] Weidenbach, Christoph: *The SPASS & FLOTTER Users Guide*, März 1997.

[Wir90] Wirsing, M.: *Algebraic Specification*, Band B der Reihe *Handbook of Theoretical Computer Science*, Kapitel 13, Seiten 675 – 788. Elsevier, 1990.