Expander2

– a formal methods presenter and animator –

Beweisfeatures

Tim Rädisch Sebastian Steinfort

10. November 2003

Inhaltsverzeichnis

1 Einführung **4**
1.1 Allgemeine Funktionsweise 4
1.1.1 Zustandsvariablen 4
1.1.2 Interaktion 4
1.2 Allgemeine Formalia 5
1.2.1 Built-in Signatur 5
1.2.2 Bag-Terms 8
1.2.3 Gentzen Klauseln 9
1.2.4 Herbrand Modell 9
1.2.5 Subsumption 10
1.3 Axiome, Theoreme, Ableitungen 11
1.3.1 Horn-Klauseln 11
1.3.2 Anwendung 12
1.3.3 Ableitungen 13
1.3.4 Polarität 13
1.4 Natürliches Beweisen 14
1.4.1 Normalisierung 15

2 Low Level: Simplifikation **15**
2.1 Implikationen 16
2.2 Konjunktionen 16
2.3 Disjunktionen 17
2.4 Sonstiges 17

3 Medium Level: Narrowing und Rewriting **18**
3.1 Narrowing entlang einem Prädikat p 18
3.2 Narrowing entlang einem Co-Prädikat p 19
3.3 Narrowing entlang einer Funktion f 19
3.4 Needed Narrowing entlang einer Funktion f 20
3.5 Rewriting entlang einer Funktion f 21

4 High Level: Syntaktische Transformation und Induktion **21**
4.1 Syntaktische Transformationen 22
4.1.1 Kopieren 22
4.1.2 Entfernen 22
4.1.3 Umkehrung 22

4.1.4 Kongruenz-Regeln 22
4.1.5 Unifikation 23
4.1.6 Instantiierung 23
4.1.7 Rewriting und Narrowing 23
4.1.8 Anwendung von Axiomen und Theoremen 23
4.2 Induktion 24
4.2.1 Noethersche Induktion 24
4.2.2 Generalisierung 26
4.2.3 Fixpunktinduktion 27
4.2.4 Starke Fixpunktinduktion 28
4.2.5 Co-Induktion 29
4.3 Schleifeninvarianten 29
4.3.1 Hoare-Induktion 30
4.3.2 Subgoal-Induktion 32

5 Anhang **33**

1 Einführung

1.1 Allgemeine Funktionsweise

1.1.1 Zustandsvariablen

Der Benutzer des *Expander2* steuert den Prozess durch Anwenden bzw. Ändern von z.B. Signaturen, Theoreme, Axiome oder Knoten durch z.B. Betätigen von Schaltflächen oder Texteingabe. Jede dieser Aktionen stellen Eingaben für den *Solver* dar, die auch diverse Zustandsvariablen ändern können.

Die Liste in der Zustandsvariable **current trees** besteht aus logischen Formeln oder algebraischen Termen. Im Falle von Formeln repräsentiert die Liste die Disjunktion oder Konjunktion ihrer Elemente. Im Falle von Termen steht die Liste für die disjunkte Vereinigung ihrer Elemente. Der auf der Zeichenfläche dargestellte Baum ist in der Zustandsvariable **current tree** abgelegt. Die boolesche Variable **formula** gibt an, ob die Liste aus Formeln oder Termen besteht.

Die Liste **solPositions** enthält die Positionen der gelösten bzw. normalisierten Formeln innerhalb des aktuellen Baumes. Die Liste **finPositions** enthält die Positionen der *non-narrowable* Formeln, der *non-rewritable* Terme und vereinfachter Terme und Formeln. Die Zustandsvariable **treePos** enthält eine Liste aller durch den Benutzer selektierter Teilbäume.

In der Zustandsvariable **current proof** wird die folge der Transformationsschritte, die seit dem letzten Parsevorgang der Eingabezeile am aktuellen Baum vollzogen wurden, gespeichert.

Eine detaillierte Auflistung aller Zustandsvariablen ist in [6] und [4] zu finden.

1.1.2 Interaktion

Expander2 erlaubt es dem Benutzer den Verlauf des Beweises bzw. der Berechnung auf drei unterschiedlichen Interaktionsleveln zu steuern bzw. zu beeinflussen.

Auf dem obersten Level werden analytische und synthetische Inferenzregeln sowie syntaktische Transformationen individuell auf lokal ausgewählte Teilbäume angewandt. Die Regeln umfassen zum Beispiel die Anwendung einzelner Axiome, Substitutions- oder Unifikationsschritte, Noethersche Induktion, Hoare-Induktion, Subgoal-Induktion, Fixpunktinduktion oder Co-Induktion.

Im mittleren Level realisieren *Narrowing* und *Rewriting* die vollständige Anwendung aller Axiome für die definieren Funktionen, Prädikate und Co-Prädikate der aktuellen Signatur.

Das Rewriting endet mit Normalformen, z.B. Terme bestehend aus Konstruktoren und Variablen. Narrowing endet entweder mit *True* oder mit der Lösung der Ausgangsformel. Sollten die Axiome ein funktional-logisches Programm re-

präsentieren, so stimmt das Narrowing entlang der Axiome mit der Ausführung des Programms überein.

Das mittlere Level erlaubt es also Programme zu testen, während die Inferenzregeln auf dem obersten Level die Programme "nur" verifizieren.

Alle zulässigen und "passenden" Transformationen des mittleren und oberen Levels werden in der Zustandsvariable *proof* protokolliert.

Auf dem untersten Level werden die Terme und Formeln mittels eingebauter Haskell-Funktionen vereinfacht und, soweit automatisiert möglich, ausgewertet. Ein Großteil dieser Beweis- und Berechnungsschritte geschehen automatisch und sind daher nicht mit allen Zwischenschritten erkennbar.

Die Funktionen umfassen arithmetische, Listen-, Bag- und Mengenoperationen sowie Berechnung von Term-Äquivalenz bzw Inäquivalenz und Methoden zur logischen Vereinfachung von Formeln zu "geschachtelten Gentzen Klauseln".

Wenn eine Funktion f im mittleren Level angewandt wird, bedeutet dies, dass man mit Hilfe der Axiome von f vereinfacht. Wendet man f dagegen auf dem untersten Level an, so bedeutet dies, dass man die eingebaute Haskellimplementierung von f ausführt bzw. benutzt.

So wird ermöglicht, dass man einzelne Algorithmen testen und das Ergebnis visuell darstellen kann, sowohl individuell während eines Beweises als auch während einer *narrowing sequence*. Auf diese Weise sind z.B. Konverter zwischen verschiedenen Darstellungen boolscher Funktionen oder auch Iteratoren zur schrittweisen Berechnung von relationalen Fixpunkten (Automatenminimierer, Datenflussanalyse, Modelchecker) im *Expander2* integriert.

1.2 Allgemeine Formalia

1.2.1 Built-in Signatur

Die in *Expander2* integrierte Signatur besteht aus den folgenden Symbolen:

Prädikate: `<`, `<=`, `>`, `>=`, `~/~`, `~~`, `==`, `` `in` ``, `` `not in` ``, `select`, `any`, `zipAny`, `>>`, `>>`, `INV`, `§`

Co-Prädikate: `~`, `all`, `zipAll`, §[1]

Konstruktoren: `<+>`, `()`, `[]`, `{}`, `:`, `0`, `suc`, `fun`, `bool`, `pile`, `piles`

definierte Funktionen: [2] `+`, `++`, `-`, `*`, `**`, `/`, `^`, `bag`, `bisim`, `dnf`, `foldl`, `lg`, `max`, `min`, `minimize`, `` `mod` ``, `nerode`, `obdd`, `optimize`, `parse`, `permute`,

[1] da dies durch *enclosure by text* verwendert werden soll, muss es als Prädikat und Co-Prädikat definiert werden

[2] Funktion, die durch eine Menge von Axiomen, ein funktionales Programm, o.ä. definiert wurde

`postflow`, `prod`, `range`, `rev`, `sat`, `set`, `stateflow`, `subsflow`, `sum`, `take`, `drop`, `x0`, `x1`, ...

fovars: `i`, `z`

Die Prädikate `<,<=,>,>=` sind für Integer- und Realzahlen sowie für die Funktionen `x0,x1,x2,...`[3] vordefiniert. `<=`, `'in'` und `'not in'` stehen für $\subseteq$, $\in$ und $\notin$ bezüglich Relationen bzw. Listen und Mengen.

`'in'` und `'not in'` behandeln ein Tripel $(i, x, [i_1, \ldots, i_n])$, wobei $i, i_1, \ldots, i_n$ Integer sind und x ein String, wie die Liste $[(i, x, i_1), \ldots, (i, x, i_n)]$; es gilt also $(i, x, [i_1, \ldots, i_n]) := [(i, x, i_1), \ldots, (i, x, i_n)]$

Es sei nun as eine endliche Liste und x und s Variablen. Der Ausdruck $select\,(as, x, s)$ wertet die Disjunktion aller Formeln der Form $x = a \& s = bs$ aus, wobei $a \in as$ und $bs = as \setminus \{a\}$.

Der Ausdruck $any\,(P)\,(as)$ wird zu $True$, falls es mindestens ein $a \in as$ gibt, das P erfüllt; es gilt also $\frac{any(P)(as)}{\exists a \in as : P(a)}$.

Der Ausdruck $all\,(P)\,(as)$ wird zu $True$, falls alle Elemente von as P erfüllen; es gilt also $\frac{all(P)(as)}{\forall a \in as : P(a)}$.

Der Ausdruck $zipAny\,(P)\,([a_1, \ldots, a_n])\,([b_1, \ldots, b_n])$ wird $True$, falls für ein $1 \leq i \leq n$ das Paar (a_i, b_i) P erfüllt; es gilt also $\frac{zipAny(P)([a_1,\ldots,a_n])([b_1,\ldots,b_n])}{\bigvee_{1 \leq i \leq n} P(a_i, b_i)}$.

Der Ausdruck $zipAll\,(P)\,([a_1, \ldots, a_n])\,([b_1, \ldots, b_n])$ wird $True$, falls für alle $1 \leq i \leq n$ das Paar (a_i, b_i) P erfüllt; es gilt also $\frac{zipAll(P)([a_1,\ldots,a_n])([b_1,\ldots,b_n])}{\bigwedge_{1 \leq i \leq n} P(a_i, b_i)}$.

Mittels des Konstruktors `()` lassen sich Produkte beliebiger, aber endlicher Länge generieren, mittels `[]` werden Listen und mittels `{}` Mengen erzeugt.

Der Konstruktor `<+>` ist ein Infix-Konstruktor für das Erzeugen einer Summe beliebiger Terme. Mit dem Konstruktor `:` wird ein Element von links an eine Liste angefügt.

Die Konstruktoren `0` und `suc` sind die Konstruktoren für natürliche Zahlen. Wenn s eine Liste von Zahlen ist, so liefert $suc\,(s)$ die nächste Permutation von s in umgekehrter Reihenfolge; z.B. $suc\,([2, 5, 3, 4, 1, 9]) = [3, 2, 5, 4, 1, 9]$. Falls s sortiert ist, so gilt $suc\,(s) = rev\,(s)$.

Der Konstruktor $fun\,(p, t)$ steht für die Lambda-Abstraktion $\lambda p \rightarrow t$. Mit dem Konstruktor `bool` wird eine Formel in einen Term eingebettet.

Die Funktionen `+`, `-` , `*` , `**` , `/` , `'mod'` , `max` , `min` sind im gewöhnlichen Sinne für natürliche und reelle Zahlen definiert. Es seien nun as und bs Mengen, dann berechnet $as - bs$ die Menge $as \setminus bs$, $lg\,(as)$ berechnet die Kardinalität $|as|$ der Menge as. Für zwei Integer m und n liefert $range\,(m, n)$ das Intervall der $[m, n]$.

Die Funktionen `++` index$++@++$, `map` indexmap$@map$, `foldl` indexfoldl$@folfl$,

[3] werden bei OBDDs verwendet

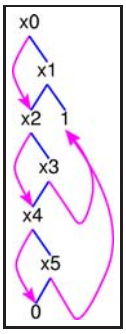

Abbildung 1: OBDD

`sum` , `take` , `drop` , `zip` und `zipWidth` stimmen mit den gleichnamigen Haskell-Funktionen überein. Lediglich `sum` wurde derart erweitert, dass es auch auf Bags und Mengen angewandt werden kann. Die Funktion `rev` ist identisch mit der Haskell-Funktion `reverse` und `prod` mit der Haskell-Funktion `product`.

Der `$`-Operator entspricht dem Haskell-Äquivalent. Das erste Argument ist ein Term t höherer Ordnung, der ein Prädikat oder eine Funktion f repräsentiert. Die weiteren Parameter sind die Argumente für f. Es gilt also $\$\,(t, a_1, \ldots, a_b) = t\,(a_1, \ldots, a_n)$

Mittels `bag` wird entweder eine Liste in ein Multiset transformiert oder Terme mittels *flattening* in flache Terme überführt, die mit dem Infix-Operator `^` gebildet wurden. Mit `set` werden Listen oder Multisets in Mengen transformiert.

Eine DNF wird repräsentiert durch eine Liste von Strings mit gleicher positiver Länge. Sie enthält an jeder Position i nur die Zeichen

0: in der Klausel ist das Literal $\overline{x_i}$ vorhanden

1: in der Klausel ist das Literal x_i vorhanden

#: in der Klausel ist weder das Literal x_i noch das Literal $\overline{x_i}$ vorhanden

Mit `obdd` wird nun eine DNF in ein äquivalentes, minimales OBDD transformiert. Mit `dnf` wird ein OBDD in eine äquivalente, minimale DNF überführt. Wird `minimize` auf eine DNF angewandt, so wird die Anzahl der Summanden minimiert; bei Anwendung auf ein OBDD (*Abb. 1*) wird die Anzahl der Knoten reduziert.

Mengenklammern, die in Klauseln verwendet werden, umschließen optionale Teilformeln.

Die Operatoren `&` , `|` , `=` , `=/=` , `~` , `~/~` , `~~` , `==` , `+` , `*` , `^`, `{}` , `<+>` werden als Permutatoren interpretiert, so spielt zum Beispiel die Reihenfolge ihrer Argumente keine Rolle. Die unter "Bag-Terms" beschrieben AC-Unifikation ersetzt

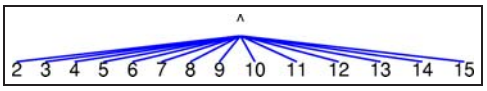

Abbildung 2: Primzahlen - Eingabe

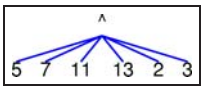

Abbildung 3: Primzahlen - Lösung

die herkömmliche Unifizierung immer dann, wenn zwei Argumentlisten eines Permutators unifiziert werden sollen.

Die Permutatoren `~` und `~~` sind für Kongruenzrelationen vorgesehen. `==` ist das semantische Äquivalent zur Gleichheit, sollte aber nur in Axiomen auftreten.

Der Operator `<` ist die Negation von `>=`, `>` von `<=`, `=/=` von `=` und `~/~` die Negation von `~`. Für jedes (Co-)Prädikat `P` bezeichnet `not_P` das Komplement.

Der Operator `=` berücksichtigt, dass Terme der Form `{t1,...,tn}` Mengen und Terme der Form `t1^...^tn` Multisets repräsentieren.

1.2.2 Bag-Terms

Der Operator `^` ist ein Infixoperator für das Erzeugen von Bags und wird von dem Unifikationsalgorithmus als assoziative und kommutative Funktion behandelt. Wenn ein Bag-Term $t = t_1 \hat{} \ldots \hat{} t_n$ mit einem weiteren Bag-Term unifiziert werden soll, so ist die Unifikation bereits möglich, wenn nur eine Permutation von t mit u unifiziert werden kann. Es werden höchstens 720 Permutationen überprüft.

Können mehrere Unifikatoren gebildet werden, so werden diejenigen bevorzugt, die nur Variablen mit Variablen substituieren. Existieren auch von dieser Art mehrere, so werden die bevorzugt, die nur mit Variablen aus u substituieren.

Beispiel: Die wiederholte Anwendung der AC-Gleichung $i'mod'j = 0 \Rightarrow !\hat{}i\hat{}j = j$ auf den Term *Abb. 2* ermittelt durch "aus sieben" die Primzahlen und endet mit dem Term *Abb. 3* .
Im ersten Schritt könnte zum Beispiel auf die Permutation *Abb. 4* die "instantiierte' AC-Gleichung

$$4'mod'2 = 0 \Rightarrow !\hat{}4\hat{}2 = 2$$

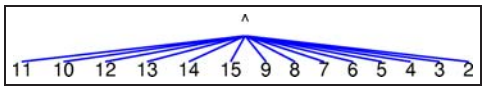

Abbildung 4: Primzahlen - Permutation

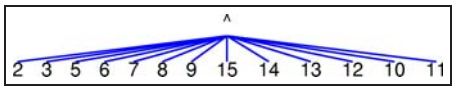

Abbildung 5: Primzahlen - Zwischenschritt

angewandt werden und so zu *Abb. 5* reduziert werden.

1.2.3 Gentzen Klauseln

Expander2 transformiert Formeln in minimale eingebettete Gentzen Klauseln der Form $\forall\vec{x}\left(\exists\vec{y}\left(t_1 \wedge \ldots \wedge t_n\right) \Rightarrow \forall\vec{z}\left(u_1 \vee \ldots \vee u_n\right)\right)$, wobei $t_1 \ldots t_n, u_1 \ldots u_n$ entweder Atome oder wieder Gentzen Klauseln sind.

Eingebettete Gentzen-Klauseln lassen sich durch die folgende kontextfreie Grammatik bilden:

$$\begin{array}{llll} S & \longrightarrow & A|B & \qquad C \longrightarrow \forall\vec{x}\left(A_1 \Rightarrow B_1\right)|atom \\ A & \longrightarrow & A_1|C & \qquad B \longrightarrow B_1|C \\ A_1 & \longrightarrow & \exists\vec{x}\left(B \wedge \ldots \wedge B\right) & \qquad B_1 \longrightarrow \forall\vec{x}\left(A \vee \ldots \vee A\right) \end{array}$$

1.2.4 Herbrand Modell

Das Herbrand-Universum $D\left(F\right)$ einer geschlossenen[4] Formel F in Skolemform ist die Menge aller variablen freien Terme, die aus den Bestandteilen von F gebildet werden können. Für den Spezialfall, dass in F keine Konstante vorkommt, wählt man eine beliebige Konstante, zum Beispiel a, und bildet dann die variablen freien Terme. Formal ausgedrückt wird $D\left(F\right)$ wie folgt induktiv definiert:

1. alle in F vorkommenden Konstanten sind in $D\left(F\right)$. Falls F keine Konstante enthält, ist a in $D\left(F\right)$
2. Für jedes in F vorkommende n-stellige Funktionssymbol f und Terme $t_1, \ldots, t_n \in D\left(F\right)$ ist der Term $f\left(t_1, \ldots, t_n\right)$ in $D\left(F\right)$

Für die Formel $F = \forall x \forall y \forall z P\left(x, f\left(y\right), g\left(z, x\right)\right)$ liegt z.B. der Spezialfall vor; daher ist $D\left(F\right) = \left\{a, f\left(a\right), g\left(a, a\right), f\left(g\left(a, a\right)\right), g\left(a, f\left(a\right)\right), \ldots\right\}$. In der Formel $G = \forall x \forall y Q\left(c, f\left(x\right), h\left(y, b\right)\right)$ kommen Konstanten vor; daher gilt $D\left(G\right) = \left\{c, b, f\left(c\right), f\left(b\right), h\left(c, c\right), f\left(f\left(c\right)\right)\right\}$.

Es sei F eine Aussage in Skolemform. Dann heißt jede zu F passende Struktur $A = \left(U_A, I_A\right)$ eine Herbrand-Struktur für F, falls folgendes gilt:

1. $U_A = D\left(F\right)$

[4] bereinigt und in Pränexform

2. für jedes in F vorkommende n-stellige Funktionssymbol F und $t_1, \ldots, t_n \in D(F)$ gilt $f^A(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$

Bei Herbrand-Strukturen sind also der Grundbereich und die Interpretation der Funktionssymbole per Definition festgelegt. Lediglich die Interpretation der Prädikatensymbole kann frei gewählt werden.

Eine Herbrand-Struktur $A = (U_A, I_A)$ für obige Beispielformel F muss also folgende Begingungen erfüllen:

- $U_A = D(F) = \{a, f(a), g(a,a), \ldots\}$
- $f^A(a) = f(a)$
- $f^A(f(a)) = f(f(a))$
- $f^A(g(a,a)) = f(g(a,a))$
- ...

Es ist nur noch die Wahl von P^A frei. Eine gültige Definition wäre z.B. $P^A = \{(\alpha, \beta, \gamma) \,|\, \alpha, \beta, \gamma \in D(f)\}$.

Es sei $F = \forall y_1 \ldots \forall y_n F^*$ eine Aussage in Skolemform. Dann ist $E(F)$, die Herbrand-Expansion von F, definiert als

$$E(F) = \left\{ F^*_{[y_1/t_1]\ldots[y_n/t_n]} \mid t_1, \ldots, t_n \in D(F) \right\}$$

Die Formeln in $E(F)$ entstehen also, indem die Terme in $D(F)$ in jeder Möglichen Weise für die Variablen in F^* substituiert werden.

In diesem Modell werden Prädikate (inklusive der Graphen begrenzter Funktionen) als geringste bzw kleinste Lösung und Co-Prädikate als größte Lösung ihrer Horn-Repräsentationen interpretiert.

1.2.5 Subsumption

Unter Subsumption versteht man, dass eine Formel eine andere Formel überdeckt. So gilt z.B. $\exists y : (x \geq 0 \wedge x < y + 1) \sqsubset \exists y : x < y$; gesprochen $\exists y : (x \geq 0 \wedge x < y + 1)$ subsumiert $\exists y : x < y$.

Subsumption ist die schwächste binäre Relation auf Termen und Formeln. Für sie gelten die folgenden Implikationen:

$$\begin{array}{rcl}
\exists 1 \leq i \leq n : \varphi \sqsubset \psi_i & \Rightarrow & \varphi \sqsubset \psi_1 \vee \ldots \vee \psi_n \\
\forall 1 \leq i \leq n : \varphi \sqsubset \psi_i & \Rightarrow & \varphi \sqsubset \psi_1 \wedge \ldots \wedge \psi_n \\
\left(\exists \vec{t} : \varphi(\vec{t}) = \psi(\vec{y})\right) \wedge (\exists \vec{u} : \psi(\vec{u}) = \varphi(\vec{x})) & \Rightarrow & \exists \vec{x} \varphi(\vec{x}) \sqsubset \exists \vec{y} \psi(\vec{y}) \\
\exists \vec{u} : \psi(\vec{u}) = \varphi & \Rightarrow & \varphi \sqsubset \exists \vec{y} \psi(\vec{y}) \\
\forall 1 \leq i \leq n : \varphi_i \sqsubset \psi & \Rightarrow & \varphi_1 \vee \ldots \vee \varphi_n \sqsubset \psi \\
\exists 1 \leq i \leq n : \varphi_i \sqsubset \psi & \Rightarrow & \varphi_1 \wedge \ldots \wedge \varphi_n \sqsubset \psi \\
\left(\exists \vec{t} : \varphi(\vec{t}) = \psi(\vec{y})\right) \wedge (\exists \vec{u} : \psi(\vec{u}) = \varphi(\vec{x})) & \Rightarrow & \forall \vec{x} \varphi(\vec{x}) \sqsubset \forall \vec{y} \psi(\vec{y}) \\
\exists \vec{t} : \varphi(\vec{t}) = \psi & \Rightarrow & \forall \vec{x} \varphi(\vec{x}) \sqsubset \psi
\end{array}$$

1.3 Axiome, Theoreme, Ableitungen

1.3.1 Horn-Klauseln

Eine Formel F ist eine Hornformel, falls F in KNF vorliegt und jedes Disjunktionsglied in F höchstens ein positives Literal enthält. Hornformel können anschaulicher als Konjunktionen von Implikationen geschrieben werden (prozedurale Deutung). So kann zum Beispiel die Hornformel $F = (A \vee \neg B) \wedge (\neg C \vee \neg A \vee D) \wedge (\neg A \vee \neg B) \wedge D \wedge \neg E$ auch als $F \equiv (B \rightarrow A) \wedge (C \wedge A \rightarrow D) \wedge (A \wedge B \rightarrow 0) \wedge (1 \rightarrow D) \wedge (E \rightarrow 0)$ dargestellt werden, wobei 0 für eine beliebige unerfüllbare Formel und 1 für eine beliebige Tautologie steht.

Axiome und Theoreme, die *Expander2* benutzt, müssen entweder als **Hornklauseln** (1-7) oder als **co-Hornklauseln** (8-13) vorliegen. Es sei nun f eine Funktion, p ein Prädikat, q ein Co-Prädikat und $a_1, \ldots, a_n$ Atome. Sind f, p oder q höherwertig, so steht $(\vec{t})$ für ein "Curry"-Tupel $(\vec{t_1}) \ldots (\vec{t_n})$. Ferner seien die Teilausdrücke in geschweiften Klammern optional. Die unterstrichenen Terme oder Atome bezeichnet man als **Anker** bzw. **anchor**. Daraus ergibt sich folgendes Schemata:

1. $\{Guard \Rightarrow\} \left(\underline{f\,(\vec{t})} = u\{\Leftarrow Premisse\}\right)$
2. $\{Guard \Rightarrow\} \left(\underline{t_1\hat{}\ldots t_k\hat{}\,!\,\hat{}t_{k+1}\hat{}\ldots\hat{}t_n} = u\{\Leftarrow Premisse\}\right)$
3. $\{Guard \Rightarrow\} \left(\underline{p\,(\vec{t})}\{\Leftarrow Premisse\}\right)$
4. $\underline{t} = u\{\Leftarrow Premisse\}$
5. $\underline{q\,(\vec{t})}\{\Leftarrow Premisse\}$
6. $\underline{at_1} \wedge \ldots \wedge \underline{at_n}\{\Leftarrow Premisse\}$
7. $\underline{at_1} \vee \ldots \vee \underline{at_n}\{\Leftarrow Premisse\}$
8. $\{Guard \Rightarrow\} \left(\underline{f\,(\vec{t})} = u \Rightarrow Konklusion\right)$
9. $\{Guard \Rightarrow\} \left(\underline{q\,(\vec{t})} \Rightarrow Konklusion\right)$
10. $\underline{t} = u \Rightarrow Konklusion$
11. $\underline{p\,(\vec{t})} \Rightarrow Konklusion$
12. $\underline{at_1} \wedge \ldots \wedge \ldots \underline{at_n} \Rightarrow Konklusion$
13. $\underline{at_1} \vee \ldots \vee \ldots \underline{at_n} \Rightarrow Konklusion$

Die Horn-Klauseln vom Typ 6 und 7 werden auch als verteilte Horn-Klauseln bezeichnet. Analog werden die Co-Horn-Klauseln vom Typ 12 und 13 als verteilte Co-Horn-Klauseln bezeichnet.

Wie in 4.1.8 gezeigt, muss man, um verteilte Horn-Axiome oder verteilte Co-Horn-Axiome anzuwenden, im aktuellen Baum n quantorfreie Teilformeln $at'_1, \ldots, at'_n$ wählen, so dass für alle $1 \leq i \leq n$ die Teilformel at'_i mit at_i unifizierbar ist.

Im Falle von verteilten Horn-Axiomen wird die Teilformel at'_i durch die entsprechende Premisse ersetzt. Die resultierenden Redukte werden nun konjunktiv vereint.

Im Falle von verteilten Co-Horn-Axiomen wird die Teilformel at'_i durch die entsprechende Konklusion ersetzt. Die resultierenden Redukte werden nun disjunktiv vereint.

1.3.2 Anwendung

Jede Anwendung einer Klausel auf ein Redex, z.B. ein Teilterm oder eine Teilformel des aktuellen Baumes, beginnt mit der Suche eines allgemeinsten Unifikators des Redex und des Ankers der Klausel. Wenn die Vereinheitlichung bzw Unifikation erfolgreich ist, und der Unifikator dem Guard genügt, so wird das Redex durch das Redukt, z.B. der Instanz einer Prämisse, Konklusion oder u bzw. dem Unifikator, ersetzt. Des weiteren ist das Redukt um Gleichungen erweitert, die die Einschränkungen des Unifikators auf die Redexvariablen repräsentieren.

Eine Klausel mit einem Guard wird nur angewandt, wenn die Instanz des Guards durch den Unifikator lösbar ist. Einerseits sind "bewachte" Axiome nötig, um effizient auswertbare Grundformeln[5] zu formulieren. Andererseits sind "unbewachte" Formeln nötig um Axiome und Theoreme zu definieren, die während des Beweisvorgangs als Lemmata benutzt werden können. Diese müssen "unbewacht" sein, da sonst eine Suche nach einer Lösung für einen Guard den gesamten Beweisvorgang blockieren könnte.

Ein Beispiel für eine bewachte Klausel[6] ist z.B.

$$split\,(s) = (s_1, s_2) \Rightarrow sort\,(x : (y : s)) = merge\,(sort\,(x : s_1)\,, sort\,(y : s_2))\,.$$

Dagegen ist das logisch äquivalente Axiom[7]

$$sort\,(x : (y : s)) = merge\,(sort\,(x : s1)\,, sort\,(y : s_2)) \Leftarrow split\,(s) = (s_1, s_2)$$

unbewacht.

[5] Formeln ohne Variablen
[6] entnommen aus *Expander2* -LISTEVAL
[7] entnommen aus *Expander2* -LIST

1.3.3 Ableitungen

In *Expander2* ist eine Ableitung eine Sequenz von aufeinander folgenden Werten der Zustandsvariable *current trees*, die in der Zustandsvariable *proof* gespeichert ist. Die Werte beider Variablen werden initialisiert, wenn die zu verarbeitende Eingabe geparst wurde, und der daraus resultierende Baum dargestellt wurde. Beide werden mit der einelementigen Menge $[t]$ initialisiert.

Eine Ableitung ist gültig bzw. korrekt, wenn die hergeleitet Disjunktion (Summe) der aktuellen Bäume den Ausgangsbaum impliziert (zu ihm äquivalent ist). Die zugrunde liegende Semantik ist durch das Herbrand Model l gegeben. *Expander2* überprüft die Korrektheit jedes Ableitungschrittes und gibt eine Warnung aus, falls ein Schritt nicht korrekt ist oder nicht korrekt sein könnte.

Eine korrekte Ableitung, die mit der Formel $True$ endet, ist ein Beweis für die Ursprungsformel, die durch die initiale Liste *current trees* gegeben ist.

1.3.4 Polarität

Für eine Formel F sind die Polaritäten der Vorkommen von Teilformeln von F wie folgt rekursiv definiert:

1. F ist positiv in F
2. sei G ein Vorkommen einer Teilformel von F der Gestalt G_1
   - ist G positiv in F, so ist G_1 negativ in F
   - ist G negativ in F, so ist G_1 positiv in F
3. sei G ein Vorkommen einer Teilformel von F der Gestalt $(G_1 \wedge G_2)$ oder $(G_1 \vee G_2)$
   - ist G positiv in F, so sind G_1 und G_2 positiv in F
   - ist G negativ in F, so sind G_1 und G_2 negativ in F
4. sei G ein Vorkommen einer Teilformel von F der Gestalt $(G_1 \Rightarrow G_2)$
   - ist G positiv in F, so ist G_1 negativ in F und G_2 positiv in F
   - ist G negativ in F, so ist G_1 positiv in F und G_2 negativ in F
5. sei G ein Vorkommen einer Teilformel von F der Gestalt $(G_1 \Leftrightarrow G_2)$
   - G_1 und G_2 sind sowohl positiv als auch negativ in F
6. sei G ein Vorkommen einer Teilformel von F der Gestalt QxG_1 für einen Quantor Q
   - ist G positiv in F, so ist G_1 positiv in F
   - ist G negativ in F, so ist G_1 negativ in F

> In diesem Fall spricht man auch von der Polarität des Vorkommens des Quantors Q und meint damit die Polarität des Vorkommens der Teilformel, die mit Q beginnt.

Diese Definition lässt sich am einfachsten veranschaulichen, in dem man die Formel F durch ihren Syntaxbaum darstellt; also einen Baum, dessen Blätter die Vorkommen von atomaren Teilformeln von F sind und dessen innere Knoten mit Junktoren und Quantoren beschriftet sind. Die Wurzel dieses Baums wird wegen (1) als positiv markiert. Der Rest der Definition gibt an, wie das Vorzeichen bzw. die Polarität auf alle Nachfolgerknoten bis hin zu den Blättern ermittelt wird.

Bei diesem Verfahren wechselt das Vorzeichen also bei jedem Negationsjunktor. Sei zum Beispiel $F = \neg\neg\neg p$, dann haben $\neg\neg\neg p$ und $\neg p$ positive Polarität in F, während $\neg\neg p$ und p negative Polarität in F haben.

Die Polarität eines Vorkommens einer Teilformel drückt aus, ob dieses Vorkommen im Bereich einer geraden oder einer ungeraden Anzahl von Negationen steht, ob diese Negationen einander also aufheben oder nicht.

Die Korrektheit einer Ableitung hängt von der Polarität des Redex sowie dessen Position im aktuellen Baum ab.
Eine Regel ist analytisch oder *expanding* , falls das Redukt das Redex impliziert. In diesem Fall muss das Redex positive Polarität haben, damit der Ableitungsschritt korrekt ist.
Eine Regel ist synthetisch oder *contracting* , falls das Redex das Redukt impliziert. In diesem Fall muss das Redex negative Polarität haben, damit der Ableitungsschritt korrekt ist.
Expander2 berechnet die Polarität eines Redex automatische während der Verifizierung eines Ableitungsschrittes.

1.4 Natürliches Beweisen

Expander2 bietet keine vollständig automatischen Routinen zur Beweisführung. Es werden natürliche Beweismethoden zur Verfügung gestellt, denen man einfach folgen und die man einfach steuern kann. Natürliche Beweismethoden haben den Vorteil, dass die zu beweisenden Formeln, Ausdrücke, ... nicht normalisiert oder in andere sie repräsentierende Formen überführt werden, die sich von ihnen derart unterscheiden wie z.B. Assemblerprogramme von äquivalenten Hochsprachenprogrammen unterscheiden.

Es gibt zahlreiche "Vermutungen" die sich automatisiert und effizient beweisen lassen. Solche Beweise folgen oft dem selben Schema, so dass sie mit vorgegebenen "Vereinfachungsregeln" bzw. "simplification rules" geführt werden können. In der Regel fallen Beweise zur Korrektheit von Programmen nicht in diese Kategorie. Insbesondere Beweise, bei denen Induktion oder Co-Induktion benötigt wird oder in denen Verallgemeinerungen gesucht werden müssen, lassen sich nicht automatisiert, also nicht ohne Userinteraktion, führen.

1.4.1 Normalisierung

Die durch *Expander2* vorgenommen Normalisierungen resultieren aus dem Verfahren des natürlichen Beweisens. Diese Normalisierung unterscheidet sich jedoch von der "klassischen" insofern, dass z.B. Implikationen und Quantoren nicht durch ihre Negationen und neue Signatursymbole ersetzt werden.

Es wird sogar eher gegenteilig verfahren. Negationszeichen werden durch den Simplifier eliminiert, indem sie als Literal den (Co-)Prädikaten vorangestellt werden. Dann werden die negierten (Co-)Prädikate komplett entfernt, in dem sie in ihre Komplemente überführt werden.

Axiome für Komplemente können aus den Axiomen für die (Co-)Prädikate gebildet werden, und Axiome für (Co-)Prädikate ebenso aus den Komplementen. Wenn P ein durch Horn-Axiome spezifiziertes Prädikat ist, so kann das Komplement $notP$ durch Co-Horn-Axiome spezifiziert werden, und ist somit ein Co-Prädikat.

2 Low Level: Simplifikation

Der Simplifier vereinfacht Formeln durch "Entfernen" von Symbolen der integrierten Signaturen, Konstruktoren und logischen Operatoren[8] und überführt sie in minimale Gentzen Klauseln. Dies wird durch wiederholtes Anwenden der folgenden "Äquivalenztransformationen" erreicht.

Weitere Simplifikationen lösen Formeln bzw. werten Terme teilweise aus, die durch in *Expander2* integrierte Prädikate und Funktionen gebildet wurden. Der Simplifier arbeite nach einer top-down-Strategie. So wird sicher gestellt, dass der Prozess terminiert und dass, falls nötig, alle anwendbare Regeln angewendet werden.

Dies ist vor allem deswegen notwendig, da der Simplifier - abgesehen von expliziten aufrufen - im Hintergrund arbeitet. So werden z.B. Narrowing-Reducts automatisch vereinfacht bevor sie ausgegeben werden.

Simplifizierungen sind z.B.

- $\frac{\forall x,y,z{:}(x{=}f(y){\wedge}Q(z)) \Rightarrow \forall x',y',z'{:}\big(Q\big(z'\big){\wedge}f\big(y'\big){=}x'\big)}{True}$
- $\frac{P(x,y) \wedge Q(z) \wedge (P(x,y){\Rightarrow}R(x,y,z))}{P(x,y) \wedge Q(z) \wedge R(x,y,z)}$
- $\frac{zipAll(=)\ ([1,x,3,4])\ ([1,2,y,4])}{x{=}2{\wedge}3{=}y}$

[8] *Expander2* parst logische Operatoren mit folgender Priorität: $\{\forall, \exists\} > \wedge > \vee > \Rightarrow$

2.1 Implikationen

- Eine Implikation, die eine Disjunktion als Prämisse oder eine Konjunktion als Konklusion besitzt, wird in eine Konjunktion von Implikationen überführt:
  - $\frac{\forall\vec{x}(\varphi_1\vee\ldots\vee\varphi_n\Rightarrow\psi)}{\forall\vec{x}(\varphi_1\Rightarrow\psi)\wedge\ldots\wedge\forall\vec{x}(\varphi_n\Rightarrow\psi)}$, z.B. $\frac{\forall\vec{x}(x>0\vee x<0\Rightarrow x\neq 0)}{\forall\vec{x}(x>0\Rightarrow x\neq 0)\vee\forall\vec{x}(x<0\Rightarrow x\neq 0)}$
  - $\frac{\forall\vec{x}(\phi\Rightarrow\psi_1\wedge\ldots\wedge\psi_n)}{\forall\vec{x}(\varphi\Rightarrow\psi_1)\wedge\ldots\wedge\forall\vec{x}(\varphi\Rightarrow\psi_n)}$, z.B. $\frac{\forall x{:}(x>0\Rightarrow -x\leq 0\wedge|x|=x)}{\forall x{:}\ x>0\Rightarrow -x\leq 0\wedge\forall x{:}\ x>0\Rightarrow|x|=x}$
- **Flattening:** eine Implikation mit einer disjunktiven Konklusion, die wiederum eine Implikation enthält, wird "verflacht": $\frac{\varphi\Rightarrow(\theta\Rightarrow\psi)\vee\psi_1\vee\ldots\vee\psi_n}{\varphi\wedge\theta\Rightarrow\psi\vee\psi\vee\psi_1\vee\ldots\vee\psi_n}$; z.B. $\frac{x>0\Rightarrow\left(x^2<1\Rightarrow x<1\right)\vee x\geq 1}{x>0\wedge x^2<1\Rightarrow x<1\vee x\geq 1}$
- Eine Implikation $\forall\vec{x}\,(x=t\wedge\varphi\Rightarrow\psi)$, wobei $x\in\vec{x}$ und x nicht in t auftritt, wird zu $\forall\vec{x}\,(\varphi\Rightarrow\psi)\,[t/x]$ reduziert
- Eine Implikation $\forall\vec{x}\,(\varphi\Rightarrow x\neq t\vee\psi)$, wobei $x\in\vec{x}$ und x nicht in t auftritt, wird zu $\forall\vec{x}\,(\varphi\Rightarrow\psi)\,[t/x]$ reduziert

2.2 Konjunktionen

- eine Konjunktion die $False$ oder komplementäre Literale enthält, wird zu $False$ reduziert; z.B. $\frac{false\wedge(\forall x{:}x<x+1)}{false}$ oder $\frac{x_1\wedge\overline{x_1}}{false}$
- der Faktor $True$ wird aus Konjunktionen entfernt; $\frac{true\wedge(\forall x{:}x<x+1)}{(\forall x{:}x<x+1)}$
- die Menge der Faktoren einer Konjunktion wird auf die in Bezug auf Subsumption minimalen Elemente reduziert; z.B. $\frac{x>10\ \wedge\ x>0}{x>10}$
- ein Allquantor vor einer Konjunktion wird in die Konjunktion gesetzt: $\frac{\forall\vec{x}(\varphi_1\wedge\ldots\wedge\varphi_n)}{\forall\vec{x}\varphi_1\wedge\ldots\wedge\forall\vec{x}\varphi_n}$; z.B. $\frac{\forall x{:}\left(x<x+1\ \wedge\ x^2\geq 0\right)}{\forall x{:}x<x+1\ \wedge\ \forall x{:}x^2\geq 0}$
- eine Konjunktion $\exists\vec{x}\,(x=t\wedge\varphi)$, wobei $x\in\vec{x}$ und x nicht in t auftritt, wird zu $\exists\vec{x}\varphi\,[t/x]$ reduziert; z.B. $\frac{\exists(x,y){:}\left(y=t\ \wedge\ x-t^2=y\right)}{\exists(x,y){:}\ (x-y^2=y)}$
- eine Konjunktion $\exists\vec{x}\,(\varphi\wedge\psi)$ mit $\psi=\forall\vec{z}\,(\psi_1\vee\ldots\vee\psi_n)$ wird simplifiziert zu $\exists\vec{x}\forall\vec{z}\,((\varphi\wedge\psi_1)\vee\ldots\vee(\varphi\wedge\psi_n))$; z.B. $\frac{\exists x\ \left(\forall y\left(y\geq y+x\ \vee\ y^2>x\right)\right)\wedge x\geq 0}{\exists x\forall y\ [(y\geq y+x\wedge x\geq 0)\vee(y^2>x\wedge x\geq 0)]}$
- eine Konjunktion $\varphi\wedge(\psi\Rightarrow\theta)$ in der ψ durch φ subsumiert wird, wird zu $\varphi\wedge\theta$ reduziert; z.B. $\frac{(x\geq 10)\ \wedge\ (x\geq 0\Rightarrow x=|x|)}{x\geq 10\Rightarrow x=|x|}$

2.3 Disjunktionen

- eine Disjunktion die $True$ oder komplementäre Literale enthält, wird zu $True$ reduziert; z.B. $\frac{true \vee x_1 \vee x_2}{true}$ oder $\frac{x_1 \vee \overline{x_1}}{true}$
- der Summand $False$ wird aus Disjunktionen entfernt; z.B. $\frac{false \vee \forall x{:}x \geq 0}{\forall x{:}x \geq 0}$
- die Menge der Summanden einer Disjunktion wird auf die in Bezug auf Subsumption maximalen Elemente reduziert; $\frac{x>10 \vee x>0}{x>0}$
- Ein Existenzquantor vor einer Disjunktion wird in die Disjunktion gesetzt: $\frac{\exists \vec{x}(\varphi_1 \vee \ldots \vee \varphi_n)}{\exists \vec{x}\varphi_1 \vee \ldots \vee \exists \vec{x}\varphi_n}$; z.B. $\frac{\exists x(x=n\pi \vee x \leq 0)}{\exists x{:}x=n\pi \vee \exists x{:}x \leq 0}$
- Eine Disjunktion $\forall \vec{x}\,(x \neq t \vee \varphi)$, wobei $x \in \vec{x}$ und x nicht in t auftritt, wird zu $\forall \vec{x}\varphi\,[t/x]$

2.4 Sonstiges

- gebundene aber ungenutzte Variablen werden entfernt; z.B. $\frac{\forall x \forall y \exists z{:}\,(x+z)^2<x+z}{\forall x \exists z{:}\,(x+z)^2<x+z}$
- direkt aufeinander folgende Quantoren werden vereint; z.B. $\frac{\forall x(\forall y(\forall z{:}p(x,y,z)))}{\forall x,y,z{:}p(x,y,z)}$
- Existenzquantoren, die vor eine Implikation oder einer Disjunktion stehen, und Allquantoren, die vor einer Konjunktion stehen, werden vor die Argumente gerückt:
  - $\frac{\exists \vec{x}(\varphi \Rightarrow \psi)}{\exists \vec{x}\varphi \Rightarrow \exists \vec{x}\psi}$; z.B. $\frac{\exists x{:}\left(y \leq |x| \Rightarrow y^2 \leq x\right)}{\exists x{:}y \leq |x| \Rightarrow \exists x{:}y^2 \leq x}$
  - $\frac{\exists \vec{x}(\varphi_1 \vee \ldots \vee \varphi_n)}{\exists \vec{x}\varphi_1 \vee \ldots \vee \exists \vec{x}\varphi_n}$; z.B. $\frac{\exists x\left(y^2>x \vee y=x\right)}{\exists x{:}y^2>x \vee \exists x{:}y=x}$
  - $\frac{\forall \vec{x}(\varphi_1 \wedge \ldots \wedge \varphi_n)}{\forall \vec{x}\varphi_1 \wedge \ldots \wedge \forall \vec{x}\varphi_n}$; z.B. $\frac{\forall x\left(x^2 \geq 0 \wedge x<x+1\right)}{\forall x{:}x^2 \geq 0 \wedge \forall x{:}x<x+1}$
- Negationssymbole werden vor die Literale positioniert und dann durch komplementäre Prädikate ersetzt:
  - $\frac{\neg P(t)}{notP(t)}$
  - $\frac{\neg notP(t)}{P(t)}$
- **Termsplitting:** Gleichungen und Ungleichungen mit "führenden" Konstruktoren werden aufgespalten; es seien nun c und d ungleiche Konstruktoren:
  - $\frac{c(t_1 \ldots t_n)=c(u_1 \ldots u_n)}{t_1=u_1 \wedge \ldots \wedge t_n=u_n}$
  - $\frac{c(t_1 \ldots t_n)=d(u_1 \ldots u_n)}{False}$
  - $\frac{c(t_1 \ldots t_n) \neq c(u_1 \ldots u_n)}{t_1=u_1 \wedge \ldots \wedge t_n=u_n}$
  - $\frac{c(t_1 \ldots t_n) \neq d(u_1 \ldots u_n)}{True}$

3 Medium Level: Narrowing und Rewriting

Narrowing wurde bei automatisierten Theorembeweisen eingeführt. Es löst Gleichungen durch die Berechnung von Unifikatoren, unter Beachtung der Regeln einer Gleichungstheorie. Nicht formal gesprochen, unifiziert Narrowing einen Term mit der linken Seite einer Rewriteregel und wendet diese Regel auf den so instantiierten Term an.

Eine effiziente Narrowingstrategie muss den Suchraum einschränken. Passende Regeln dürfen nicht ignoriert werden, andererseits ist es möglich die Ableitung bestimmter Teilterme zu vernachlässigen, ohne die Vollständigkeit zu verlieren, d.h. es geht dadurch kein Ergebnis verloren.

Die Narrowing-Methoden von *Expander2* wenden Axiome und Simplifikationsregeln wiederholt top-down ("outermost") auf den Baum *current tree* an. Wenn der aktuelle Baum "non-narrowable" ist, z.B. wenn kein Axiom mehr angewandt werden kann, wird ein andere Baum zufällig aus der Liste *current trees* gewählt und *current tree* zugewiesen. Die Schleife endet, wenn alle Bäume "non-narrowable" sind oder wenn die Anzahl der aufeinander folgenden Narrowing-Schritte die vorgegeben Größe erreicht hat.

Im Gegensatz zur individuellen Anwendung von Axiomen, werden durch die Narrowing-Methode alle anwendbaren Axiome für den Anker eines Redex simultan angewandt. Daher führen Narrowing-Schritte zu Fallunterscheidungen. Die Axiome für (Co-)Prädikate oder Funktionen können entweder *eager* oder *lazy* sein. Eager-Axiome werden als erstes angewandt. Lazy-Axiome werden nur angewandt, wenn innerhalb einer kompletten top-down-Bearbeitung des *current tree* , keine Eager-Axiome mehr angewandt werden können.

AC-Axiome sind typische Kandidaten für Lazy-Axiome. Die wiederholte Anwendung eines AC-Axioms auf den selben AC-Term könnte verhindern, dass andere Axiome angewandt werden können. Dies kann nicht passieren, wenn AC Axiome *lazy* und andere Axiome *eager* sind.

Bei der gleichzeitigen Anwendung aller anwendbaren (Horn-)Axiome für ein Prädikat oder eine Funktion, wird das Redex durch die Disjunktion der Prämissen zusammen mit Gleichungen, die die berechneten Unifikatoren repräsentieren, ersetzt.

Bei der gleichzeitigen Anwendung aller anwendbaren (Co-Horn-)Axiome für ein Co-Prädikat, wird das Redex durch die Konjunktion der Konklusionen ersetzt.

3.1 Narrowing entlang einem Prädikat p

Expander2 folgt hierbei dem Schema

$$\frac{p\,(t)}{\bigvee_{i=1}^{k} \exists Z_i \ : \ (\varphi_i \sigma_i \wedge \vec{x} = \vec{x}\sigma_i)} \ ,$$

wobei $p(t)$ durch die Horn-Axiome $\forall_{i=1}^{n} : \gamma_i \Rightarrow (p(u_i) \Leftarrow \varphi_i)$ spezifiziert ist. Ferner muss gelten

- $\vec{x}$ sind Variablen aus t
- $\forall_{i=1}^{k}$:
  - $t\sigma_i = u_i\sigma_i$, d.h. t und u_i sind mit der Substitutionsfunktion σ_i unifizierbar
  - $\gamma_i\sigma_i \vdash True$, σ_i also dem Guard γ_i löst[9]
  - $Z_i = var(u_i, \varphi_i)$, also ist Z_i die Menge der Variablen, die in u_i oder φ_i vorkommen, also die Menge der Variablen, die nicht nur in γ_i verwendet werden
- $\forall_{k<i}^{n}$: t ist nicht unifizierbar mit u_i

3.2 Narrowing entlang einem Co-Prädikat p

Expander2 folgt hierbei dem Schema

$$\frac{p(t)}{\bigwedge_{i=1}^{k} \forall Z_i \ : \ (\vec{x} = \vec{x}\sigma_i \Rightarrow \varphi_i\sigma_i)} ,$$

wobei $p(t)$ durch die Co-Horn-Axiome $\forall_{i=1}^{n} : \gamma_i \Rightarrow (p(u_i) \Rightarrow \varphi_i)$ spezifiziert ist. Ferner muss gelten

- $\vec{x}$ sind Variablen aus t
- $\forall_{i=1}^{k}$:
  - $t\sigma_i = u_i\sigma_i$, d.h. t und u_i sind mit der Substitutionsfunktion σ_i unifizierbar
  - $\gamma_i\sigma_i \vdash True$, σ_i also dem Guard γ_i löst
  - $Z_i = var(u_i, \varphi_i)$, also ist Z_i die Menge der Variablen, die in u_i oder φ_i vorkommen, also die Menge der Variablen, die nicht nur in γ_i verwendet werden
- $\forall_{k<i}^{n}$: t ist nicht unifizierbar mit u_i

3.3 Narrowing entlang einer Funktion f

Expander2 folgt hierbei dem Schema

$$\frac{p(\ldots, f(t), \ldots)}{\bigvee_{i=1}^{k} : \exists Z_i \ : \ (p(\ldots, v_i\sigma_i, \ldots) \wedge \varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i)} ,$$

wobei $f(t)$ durch die Horn-Axiome $\forall_{i=1}^{n} : \gamma_i \Rightarrow (f(u_i) = v_i) \Leftarrow \varphi_i$ spezifiziert ist. Ferner muss gelten

[9] *Expander2* versucht γ_i in höchstens 500 Narrowing-Schritten zu lösen

- $\vec{x}$ sind Variablen aus t
- $\forall_{i=1}^{k}$:
  - $t\sigma_i = u_i\sigma_i$, d.h. t und u_i sind mit der Substitutionsfunktion σ_i unifizierbar
  - $\gamma_i\sigma_i \vdash True$, σ_i also dem Guard γ_i löst
  - $Z_i = var\,(u_i, \varphi_i)$, also ist Z_i die Menge der Variablen, die in u_i oder φ_i vorkommen, also die Menge der Variablen, die nicht nur in γ_i verwendet werden
- $\forall_{k<i}^{n}$: t ist nicht unifizierbar mit u_i

f und p müssen hierbei nicht unär sein. Die Unifikation von t und u_i kann fehlschlagen, da an einer Stelle t eine Funktion f hervorbringen kann und u_i einen Konstruktor enthält.

3.4 Needed Narrowing entlang einer Funktion f

Expander2 folgt hierbei dem Schema

$$\frac{p\,(\ldots, f\,(t)\,, \ldots)}{\begin{array}{ll} & \bigvee_{i=1}^{k} \exists Z_i : \; (p\,(\ldots, v_i\sigma_i, \ldots) \wedge \varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i) \\ \vee & \bigvee_{i=k+1}^{l} (p\,(\ldots, f\,(t\sigma_i)\,, \ldots) \wedge \vec{x} = \vec{x}\sigma_i) \end{array}},$$

wobei f durch die Horn-Axiome $\forall_{i=1}^{n} : \; \gamma_i \Rightarrow (f\,(u_i) = v_i) \Leftarrow \varphi_i$ spezifiziert ist. Ferner muss gelten

- $\vec{x}$ sind Variablen aus t
- $\forall_{i=1}^{k}$:
  - $t\sigma_i = u_i\sigma_i$, d.h. t und u_i sind mit der Substitutionsfunktion σ_i unifizierbar
  - $\gamma_i\sigma_i \vdash True$, σ_i also dem Guard γ_i löst
  - $Z_i = var\,(u_i, \varphi_i)$, also ist Z_i die Menge der Variablen, die in u_i oder φ_i vorkommen, also die Menge der Variablen, die nicht nur in γ_i verwendet werden
- $\forall_{k<i}^{l}$: σ_i ist ein partieller Unifikator von t und u_i
- $\forall_{l<i}^{n}$: t ist nicht partiell unifizierbar mit u_i

3.5 Rewriting entlang einer Funktion f

Expander2 folgt hierbei dem Schema

$$\frac{f\,(t)}{\langle + \rangle_{i=1}^{n} v_i \sigma_i}\ ,$$

wobei f durch die Horn-Axiome $\forall_{i=1}^{n}:\ \gamma_i \Rightarrow f\,(u_i) = v_i$ spezifziert ist. Ferner muss gelten

- $\vec{x}$ sind Variablen aus t
- $\forall_{i=1}^{k}$:
  - $t\sigma_i = u_i\sigma_i$, d.h. t und u_i sind mit der Substitutionsfunktion σ_i unifizierbar
  - $\gamma_i\sigma_i \vdash True$, σ_i also dem Guard γ_i löst
- $\forall_{k<i}^{n}$: t ist nicht unifizierbar mit u_i

Wenn ein Rewriting-Schritt eine Summe von Termen zurück gibt, spezifizieren die angewandten Axiome eine nicht deterministische Funktion; jedes Redukt $v_i\sigma_i$ ist eine mögliche Lösung.

4 High Level: Syntaktische Transformation und Induktion

Narrowing und Simplifikationen sind beide analytisch und synthetisch und transformieren Formeln in semantisch äquivalente Formeln. Auf dem obersten Level werden analytische und synthetische Inferenzregeln sowie syntaktische Transformationen individuell auf lokal ausgewählte Teilbäume angewendet. Die Regeln umfassen zum Beispiel die Anwendung einzelner Axiome, Substitutions- oder Unifikationsschritte, Noethersche Induktion, Hoare-Induktion, Subgoal-Induktion, Fixpunktinduktion oder Co-Induktion.

Falls der Baum einen Term repräsentiert, ist eine Ableitung korrekt, falls die Summe der Ableitung äquivalent zur Summe des Vorgängers bzw. Ursprungs ist. Repräsentiert der Baum eine Formel, so ist eine Ableitung korrekt, falls die Disjunktion (Konjunktion) der Ableitung durch die Disjunktion (Konjunktion) des Vorgängers impliziert wird.

Inkorrekte Ableitungen werden erkannt und verursachen eine Warnmeldung. Alle gültigen Transformationen werden u.a. durch Korrektheitsbeweise protokolliert. Alle Terme und Formeln werden durch die Symbole der Signatur *current signature* gebildet.

4.1 Syntaktische Transformationen

4.1.1 Kopieren

Es muss ein Teilbaum t mit Wurzel v ausgewählt werden. Es sei nun w der Parent-Knoten von v. Dann wird eine Kopie von t erstellt und diese in die Liste der Kinder von w eingefügt. Diese Transformation ist nur korrekt, falls der Parent-Knoten w eine Konjunktion oder eine Disjunktion beschreibt.

4.1.2 Entfernen

Es werden Teilbäume $\psi_1 \dots \psi_n$ ausgewählt. Sie werden aus dem aktuellen Baum entfernt. Die Transformation ist korrekt, wenn

- $\psi_1 \dots \psi_n$ Faktoren der gleichen Konjunktion φ sind, und die Polarität von φ negativ ist
- $\psi_1 \dots \psi_n$ Summanden der gleichen Disjunktion φ sind, und die Polarität von φ positiv ist

4.1.3 Umkehrung

Die Liste der selektierten Teilbäume wird umgekehrt. Diese Transformation ist korrekt, wenn alle Teilbäume Argumente eines permutativen Operators sind. Folgende Operatoren werden als permutativ betrachtet: `&`, `|`, `=`, `=/=`, `~`, `~~`, `~/~`, `+`, `*`, `^`, `{}`

4.1.4 Kongruenz-Regeln

Es muss eine Formel eines der folgenden Typen ausgewählt werden.

1. ein Atom tRt' mit positiver Polarität und $R \in \{=, \sim\}$
2. ein Atom tRt' mit negativer Polarität und $R \in \{\neq, \not\sim\}$
3. ein Atom tRt' mit positiver Polarität und $R \in Trans$, wobei $Trans := \{<, \leq, >, \geq\}$
4. $n-1$ Faktoren $t_i R t_{i-1}$ einer Konjunktion mit negativer Polarität und $R \in Trans$

Dies ausgewählten Formeln werden entsprechend der Voraussetzung, das R transitiv und kompatibel zu den Funktionssymbolen ist, abgeändert.

4.1.5 Unifikation

- es müssen zwei Faktoren einer Konjunktion $\varphi = \exists \vec{x} (\varphi_1 \wedge \ldots \varphi_n)$ ausgewählt werden. Wenn die beiden unifizierbar sind, und der Unifikator nur Variablen aus $\vec{x}$ instantiiert, so wird einer der beiden Faktoren entfernt, und der Unifikator auf die verbleibende Konjunktion angewendet. Die Transformation ist korrekt, wenn φ positive Polarität besitzt.

- es müssen zwei Summanden einer Disjunktion $\psi = \forall \vec{x} (\varphi_1 \vee \ldots \vee \varphi_n)$ ausgewählt werden. Wenn die beiden unifizierbar sind, und der Unifikator nur Variablen aus $\vec{x}$ instantiiert, so wird einer der beiden Faktoren entfernt, und der Unifikator auf die verbleibende Disjunktion angewendet. Die Transformation ist korrekt, wenn ψ eine negative Polarität besitzt.

4.1.6 Instantiierung

- es wird eine durch einen Existenzquantor gebundene Variable x gewählt. Wenn der Scope bzw. Wirkungsbereich des Quantors positive Polarität besitzt, so werden alle Vorkommen von x durch den im Solver eingegeben Term ersetzt.

- es wurde eine durch einen Allquantor gebundene Variable x gewählt. Wenn der Scope bzw. Wirkungsbereich des Quantors negative Polarität besitzt, so werden alle Vorkommen von x durch den im Solver eingegeben Term ersetzt.

Der Ersetzungsterm kann auch aus dem aktuellen Baum stammen. Hierfür zieht man einen Teilbaum über eine Position von x.

4.1.7 Rewriting und Narrowing

Es muss ein Teilbaum ausgewählt werden und eine Funktion oder ein (Co-)Prädikat p in den Solver eingegeben werden. Nun werden Narrowing- und Rewriting-Schritte mittels des Axioms p nur im ausgewählten Teilbaum durchgeführt.

4.1.8 Anwendung von Axiomen und Theoremen

Es muss ein Teilbaum selektiert werden und die Nummer eines Axioms über das Eingabefeld spezifiziert werden. Das ausgewählte Axiom bzw. Theorem wird dann, je nach Auswahl, von links nach rechts oder von rechts nach links auf den Teilbaum angewandt.

Wenn das anzuwendende Axiom der Form 1, 2, 4, oder 8 ist, bezieht sich links bzw. rechts auf die entsprechende Seite der führenden Gleichung. Andernfalls bezieht es sich auf die Prämisse bzw. Konklusion des anzuwendenden Axioms.

Die Transformation ist korrekt, falls die Prämisse auf eine Teilformel mit negativer Polarität angewandt wird, oder die Konklusion auf eine Teilformel mit positiver Polarität.

Axiome der Form 6, 7, 12 oder 13 werden auf quantorfreie Atome $at'_1, \ldots, at'_n$, die Teile einer Konjunktion oder Disjunktion sind, angewandt. Die Atome müssen quantorfrei sein, da sonst die Suche nach passenden Redices erheblich verkompliziert würde. Es sei nun V die Menge der Variablen, die in der Prämisse *prem* bzw. der Konklusion *conc* verwendet werden, aber nicht in einem Atom at'_i auftreten. Es gilt nun

Konjunktive Resolution: $\dfrac{\bigwedge_{i=1}^{n} \varphi_i\left(at'_i\right)}{\bigwedge_{i=1}^{n} \varphi_i\left(\exists V : \left(prem\sigma \wedge \bigwedge_{x \in dom(\sigma)} x \equiv x\sigma\right)\right)} \Uparrow^{10},$

wobei $\forall_{i=1}^{n} : at'_i\sigma = at_i\sigma$ und die logischen Operatoren von φ_i auf $\wedge$ und $\forall$ beschränkt sind

Disjunktive Resolution: $\dfrac{\bigvee_{i=1}^{n} \varphi_i\left(at'_i\right)}{\bigwedge_{i=1}^{n} \varphi_i\left(\exists V : \left(prem\sigma \wedge \bigwedge_{x \in dom(\sigma)} x \equiv x\sigma\right)\right)} \Uparrow,$

wobei $\forall_{i=1}^{n} : at'_i\sigma = at_i\sigma$ und die logischen Operatoren von φ_i auf $\vee$ und $\exists$ beschränkt sind

Konjunktive Coresolution: $\dfrac{\bigwedge_{i=1}^{n} \varphi_i\left(at'_i\right)}{\bigwedge_{i=1}^{n} \varphi_i\left(\forall V \left(\bigwedge_{x \in dom(x)} x \equiv x\sigma \Rightarrow conc\sigma\right)\right)} \Downarrow,$

wobei $\forall_{i=1}^{n} : at'_i\sigma = at_i\sigma$ und die logischen Operatoren von φ_i auf $\wedge$ und $\forall$ beschränkt sind.

Disjunktive Coresolution: $\dfrac{\bigvee_{i=1}^{n} \varphi_i\left(at'_i\right)}{\bigwedge_{i=1}^{n} \varphi_i\left(\forall V \left(\bigwedge_{x \in dom(x)} x \equiv x\sigma \Rightarrow conc\sigma\right)\right)} \Downarrow,$

wobei $\forall_{i=1}^{n} : at'_i\sigma = at_i\sigma$ und die logischen Operatoren von φ_i auf $\vee$ und $\exists$ beschränkt sind.

Während Resolution nur auf ein Atom angewand werden kann, kann man mit Hilfe der obigen 4 Regeln n Atome gleichzeitig ersetzen. Ausserdem wird die erzeugte Substitution immer als Konjunktion $\bigwedge_{x \in dom(\sigma)} x \equiv x\sigma$ von Gleichungen wiedergegeben.

Zum Beweis der Korrektheit sei hier auf Seite 82ff [6] verwiesen.

4.2 Induktion

4.2.1 Noethersche Induktion

Die explizite oder Noethersche Induktion bezeichnet den Ansatz eine Behauptung $\forall x \varphi(x)$ dadurch zu beweisen, dass man eine wohl fundierte Ordnung $\gg$

[10] Domain; $dom(\sigma) = \{x | \sigma(x) \neq x\}$

angibt und $\varphi(x)$ herleitet unter der Annahme, dass $\forall y \ll x : \varphi(y)$ gilt. Explizite Induktion ist also die Anwendung der Expansion $\frac{\forall x:\varphi(x)}{\forall x(\forall y(x \gg y \Rightarrow \varphi(y)) \Rightarrow \varphi(x))}$.

Die vollständige Induktion ist somit ein Spezialfall der expliziten Induktion; es gilt nämlich $x, y \in N$ mit der "gewöhnlichen Grösserbeziehung" $\gg:=>$.

Das Prinzip der noetherschen Induktion ist sehr stark, mit ihm lässt sich beispielsweise das "Lemma von König" beweisen. Das Lemma von König besagt, dass jeder Baum T, der keinen unendlichen Ast besitzt, und der keinen Knoten mit unendlich vielen Söhnen besitzt, endlich ist; d.h. T hat endlich viele Knoten.

Für den Beweis sind noch folgende Definitionen notwendig. $P(u)$ gilt genau dann, wenn der Knoten u in dem Baum T nur endlich viele Nachkommen hat. Ferner gilt $u > w$ genau dann, wenn der Knoten v ein Nachkomme von u in T ist.

Es sei nun u ein beliebiger Knoten in T. Hat u keinen Sohn, so gilt offenbar $P(u)$. Ansonsten besitzt u die endlich vielen Söhne $u_1, \ldots, u_n$. Nach Induktionsvoraussetzung besitzt jeder Knoten u_i nur endliche viele Nachkommen k_i. Damit hat u genau $n + \sum_{i=1}^{n} k_i$, also endliche viele, Nachkommen. Es gilt also auch in diesem Fall $P(u)$. Da jetzt nach dem Induktionsprinzip $P(u)$ für alle Knoten u gilt, gilt auch $P(w)$ für die Wurzel w von T. Somit ist T endlich.

Um eine explizite Induktion mittels *Expander2* durchzuführen, muss man zunächst die zu beweisende Teilformeln φ, i.d.R. eine Implikation $Premisse \Rightarrow Konklusion$, und anschließend die Induktionsvariablen auswählen. Durch *Expander2* werden dann zwei Lemmata bzw. Hypothesen, das so genannte Schrittlemma, erzeugt und in die Liste *current theorems* aufgenommen:

- $conc' \Leftarrow (x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n) \wedge prem'$
- $prem' \Rightarrow ((x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n) \Rightarrow conc')$

Ist die Teilformel keine Implikation, also quasi eine Konklusion, so wird nur $conc' \Leftarrow (x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n)$ aufgenommen. Dabei entstehen $conc'$ und $prem'$ aus $conc$ bzw $prem$, indem jedes Vorkommen einer Induktionsvariable x durch x' ersetzt wird. Die Induktionsordnung wird durch $\gg$ gegeben, und die "Abstiegsbedingung" $(x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n)$ wird als Atom betrachtet.

Jede Anwendung der Induktionshypothese verursacht nun ein Vorkommen von $\gg$. Durch "geeignete" Narrowing-Schritte werden diese Vorkommen nun entfernt. Es werden also $prem'$ und $conc'$ bewiesen - auf $True$ reduziert - und somit die Teilformeln φ bewiesen.

Als Beispiel sei hier nun die Vorbereitung des Beweises eines logischen Programms zur Berechnung der Partitionen einer Liste. Es ist Teil der folgenden Spezifikation:

```
PARTITION = LIST then
  defuncts  flatten : list(list) -> list
```

```
preds     part : list * list(list)
vars      x,y : entry
          s,s' : list
          p : list(list)
axioms    1: flatten( [] ) = []
          2: flatten( s:p ) = s ++ flatten( p )

          3: part( x:[], (x:[]):[] )
          4: part( x:y:s, (x:[]):p ) <== part( y:s, p )
          5: part( x:y:s, (x:s'):p ) <== part( y:s, s':p )
```

Die Korrektheit der Axiome wird durch die Hornformel $\varphi = part\,(S, Pt) \Rightarrow S \equiv flatten\,(Pt)$ beschrieben. Es muss also eine Implikation bewiesen werden. In φ kommen zwei freie Variablen - S vom Typ $list$ und Pt vom Typ $list\,(list)$ vor. Für das Beispiel wird S als Induktionsvariable[11] gewählt. Für den Induktionsbeweis wird also ein Prädikat $\gg$: $list \times list$ benötigt. Als Schrittlemma werden $S_3 \equiv flatten\,(Pt_0) \Leftarrow part\,(S_3, Pt_0) \wedge !S \gg S_3$ und $part\,(S_3, Pt_0) \Rightarrow (!S \gg S_3 \Rightarrow S_3 \equiv flatten\,(Pt_0))$ aufgenommen. Die zu beweisende Formel hat nun also die Form

$$\forall Pt : (part\,(!S, Pt) \Rightarrow !S \equiv flatten\,(Pt))$$

Da es ab hier mehr oder weniger nur ein Ab- bzw Umschreiben des Beispiels 7.2.1 aus [6] wäre, sei an dieser Stelle für ein komplettes Beispiel darauf verwiesen.

4.2.2 Generalisierung

Für Induktionsbeweise ist es oft notwendig, statt der ursprünglichen Formel φ eine allgemeinere, stärkere Formel ψ für die Instanziierung des Induktionsschemas zu verwenden; es gilt dann $\psi \Rightarrow \varphi$. Der Grund liegt in der Struktur des Induktionsbeweises. Verschärft man eine zu beweisende Aussage, dann muss man zwar einerseits einen stärkerem Induktionsschluss beweisen, doch kann man andererseits eine stärke Induktionshypothese voraussetzen. Es besteht allerfings die Gefahr, dass die nach der Generalisierung zu beweisende Aussage falsch wird und damit nicht mehr aus der bestehenden Datenbasis ableitbar ist.
Eine geeignete Generalisierung von φ ist in vielen Fällen die konjunktiv Verknüpfung von φ mit einem weiteren Beweisziel φ'; also $\varphi \wedge \varphi' \Rightarrow \varphi$. Lässt sich nun $\varphi \wedge \varphi'$ beweisen, ist damit auch φ bewiesen.

Um eine Generalisierung mit *Expander2* durchzuführen muss eine (Teil-)Formel φ der Form

1. $f\,(\vec{t}) = u\,\{\wedge prem\} \Rightarrow conc$

[11] *Expander2* markiert Induktionsvariablen mit einem Ausrufezeichen

2. $p\left(\vec{t}\right)\{\wedge prem\} \Rightarrow conc$
3. $\{prem \Rightarrow\}\, q\left(\vec{t}\right)$

ausgewählt werden. Ferner muss im Eingabefeld des *Expander2* eine Formel φ' eingetragen sein. Diese Formel wird nun wie beschrieben konjunktiv mit φ verknüpft und somit ψ erzeugt.

1. $f\left(\vec{t}\right) = u\,\{\wedge prem\} \Rightarrow conc \wedge \varphi'$
2. $p\left(\vec{t}\right)\{\wedge prem\} \Rightarrow conc \wedge \varphi'$
3. $\{prem \vee\}\, \varphi' \Rightarrow q\left(\vec{t}\right)$

4.2.3 Fixpunktinduktion

Im Gegensatz zur expliziten Induktion kommt die Fixpunktinduktion ohne wohlfundierte Ordnungen aus. Um eine Fixpunktinduktion mit *Expander2* durchzuführen, muss eine Teilformel φ ausgewählt werden, die einer der folgenden Formen entsprechen muss, wobei p ein Prädikat ist, das weder von $prem_i$ noch von $conc_i$ abhängt, und f eine Funktion, die weder von $prem_i$, $conc_i$ noch t_i abhängt.

1. $\bigwedge_{i=1}^{k}\ p\left(\vec{t_i}\right) \Rightarrow conc_i$
2. $\bigwedge_{i=1}^{k}\ f\left(\vec{t_i}\right) = u_i \Rightarrow conc_i$
3. $\bigwedge_{i=1}^{k}\ f\left(\vec{t_i}\right) = u_i\,\{\wedge conc_i\}$

Diese Teilformeln werden nun für die Induktion in folgende Ausdrücke überführt, wobei $\vec{x}$ eine Liste von Variablen und z eine Variable ist.

1. $p\left(\vec{x}\right) \Rightarrow \bigwedge_{i=1}^{k}\ \left(\vec{x} = \vec{t_i} \Rightarrow conc_i\right)$
2. $f\left(\vec{x}\right) = z \Rightarrow \bigwedge_{i=1}^{k}\ \left(\vec{x} = \vec{t_i} \wedge z = u_i \Rightarrow u_i\right)$
3. $f\left(\vec{x}\right) = z \Rightarrow \bigwedge_{i=1}^{k}\ \left(\vec{x} = \vec{t_i} \wedge z = u_i\,\{\wedge conc_i\}\right)$

Die "überführten" Formeln werden nun auf alle passenden Redizes innerhalb von Axiomen, die f bzw. p repräsentieren, angewendet. Die Konjunktion der transformierten Axiome ersetzt nun die ursprünglich zu beweisende Formel.

Als Beispiel sei hier wieder der Beginn des Korrektheitsbeweises[12] von $\varphi = part\left(S, Pt\right) \Rightarrow S \equiv flatten\left(Pt\right)$ gegeben. Wie obiger Aufstellung zu entnehmen ist, wird *part* durch die folgenden Axiome spezifiziert:

- $part\left(x : [\,],\ (x : []) : []\right)$

[12] der komplette Beweis ist in [6] ab Seite 94 zu finden

- $part\,(x : y : s\ ,\ (x : []) : p) \Leftarrow part\,(y : s\ ,\ p)$
- $part\,(x : y : s\ ,\ (x : s') : p) \Leftarrow part\,(x : s\ ,\ s' : p)$

Auf jedes dieser Axiome wird nun φ' angewandt, wobei bereits zur eindeutigen Unterscheidung der Variablen Indizes vergeben werden.

- $[x_0]\ =\ flatten\,([[x_0]])$
- $x_1 : (y_0 : s_5) = flatten\,([x_1] : p_0)\ \Leftarrow\ y_0 : s_5 = flatten\,(p_0)$
- $x_2 : (y_1 : s_6) = flatten\,((x_2 : s_0') : p_1)\ \Leftarrow\ y_1 : s_6 = flatten\,(s_0' : p_1)$

Diese drei Ausdrücke werden nun konjunktiv verbunden und ersetzen die Ausgangsgleichung $part\,(S, Pt) \Rightarrow S \equiv flatten\,(Pt)$. Um diesen Gesamtausdruck nun zu beweisen, wird jeder einzelne Faktor soweit möglich für sich durch Simplifikations- und Narrowingschritte aufgelöst, und der Gesamtterm letztlich zu $True$ abgeleitet.

Als weiteres Beispiel dient der Beweis von MergeSort, der sich ebenfalls in [6] befindet, der aber auch in etwas andere Form in [3] zu finden ist.

4.2.4 Starke Fixpunktinduktion

Eine starke Fixpunktinduktion verläuft mit *Expander2* analog zur normalen Fixpunktinduktion. Der einzige Unterschied liegt in der Art und Weise wie *Expander2* arbeitet.

Bevor die transformierten Formeln jedoch auf die Axiome für p oder f angewandt werden, werden diese Axiome jedoch verändert. Jedes Axiom $p\,(t_1, \ldots, t_n) \Leftarrow prem$ für p wird durch $ax := p\,(t_1, \ldots, t_n) \Rightarrow prem\,[p'/p] \wedge p''\,(t_1, \ldots, t_n)$ ersetzt. Jedes Axiom $f\,(t_1, \ldots, t_n) = t \Rightarrow prem$ für f wird durch $ax' := f\,(t_1, \ldots, t_n) = t \Rightarrow prem\,[f'\,(u_1, \ldots, u_n, u)\,/\,f\,(u_1, \ldots, u_n) = u] \wedge f''\,(t_1, \ldots, t_n, t)$ ersetzt.

Daher werden im weiteren Verlauf - analog zur Fixpunktinduktion - durch Anwendung auf die Axiome ax bzw. ax' nur die Konklusionen modifiziert.
So wird z.B. ax in

$$conj := \bigvee_{i=1}^{k} \left(\{prem_i \Rightarrow\} \bigwedge_{j=1}^{n} (t_j = t_{i,j}) \Rightarrow conc_i \right)$$

$$\Rightarrow prem\,[p'/p] \wedge p''\,(t_1, \ldots, t_n)$$

und ax' in

$$conj' := \bigvee_{i=1}^{k} \left(\bigwedge_{j=1}^{n} (\{prem_i \wedge\}\, t_j = t_{i,j}) \Rightarrow conc_i \right)$$

$$\Rightarrow prem\,[f'\,(u_1, \ldots, u_n, u]\,/\,f\,(u_1, \ldots, u_n) = u] \wedge f''\,(t_1, \ldots, t_n, t)$$

überführt.

Dann wird, wie angegeben, p' bzw. f' als neues Coprädikat in die Signatur aufgenommen und die Co-Horn-Klauseln der Fixpunktinduktion werden mit p' statt p bzw. $f'(x_1, \ldots, x_n, x)$ statt $f(x_1, \ldots, x_n) = x$ in die Liste der Axiome aufgenommen. Ausserdem wird $p'(x_1, \ldots, x_n) \Rightarrow p(x_1, \ldots, x_n)$ ein weiteres Axiom für p', sowie $f'(x_1, \ldots, x_n, x) \Rightarrow f(x_1, \ldots, x_n) = x$ ein weiteres Axiom für f wird. Abschliessend wird das Vorkommen von p'' in $conj$ durch p ersetzt und das Vorkommen von f'' in $conj'$ durch f.

4.2.5 Co-Induktion

Die Co-Induktion ist ein zur Fixpunktinduktion duales Verfahren für Co-Prädikate[13]. Für das Durchführen einer Co-Induktion muss eine Teilformel der Form

$$\bigwedge_{i=1}^{k} \left(\{prem_i \Rightarrow\}\, q\left(\vec{t_i}\right)\right)$$

ausgewählt werden, wobei q ein Co-Prädikat ist, dass von keinem Prädikat und keiner Funktion aus $prem_i$ abhängt. Die Ausgangsformel wird nun in

$$q(\vec{x}) \Leftarrow \bigwedge_{i=1}^{k} \left(\{prem_i \wedge\}\, \vec{x} = \vec{t_i}\right)$$

transformiert, wobei $\vec{x}$ analog zur Fixpunktinduktion eine Liste von Variablen ist.

Nun wird analog zur Fixpunktinduktion die transformierte Ausgangsformel auf alle passenden Redizes, innerhalb von Axiomen die q spezifizieren, angewandt. Die Konjunktion der modifizierten Axiome ersetzt auch hier wieder die Ausgangsformel.

4.3 Schleifeninvarianten

Eine bestimmte Struktur der Axiome für eine Funktion oder ein Prädikat nennt man auch Programmschema. Gegeben sei nun folgendes Schema, das für iterative Programme charakteristisch ist. Es besteht aus den drei Sorten A, B, C, einer Funktion $f : A \to C$, einer Schleifenfunktion $g : B \to C$, Hilfsfunktionen $in : A \to B$, $out : B \to C$ und $loop_1 \ldots loop_n : B \to B$, sowie einer Axiommenge der Form

- $f(x) \equiv g(in(x))$
- $\forall 1 \leq i \leq n : g(y) \equiv g(loop_i(y)) \Leftarrow \delta_i(y)$
- $g(y) \equiv out(y) \Leftarrow \delta(y)$

[13] Prädikate, die mittels co-Horn-Axiomen spezifiert sind

Der Verifikation von Schleifenprogrammen dienen zwei komplementäre Verfahren: Hoare-Induktion und Subgoal-Induktion.

4.3.1 Hoare-Induktion

Dual zur Subgoal-Induktion beschreibt eine Hoare-Invariante $\vartheta(x, y)$ einen Zusammenhang zwischen Werten der Eingabevariable x und Werten der Schleifenvariable y. Die Erzeugung der Hoare-Invariante folgt dem folgenden Schema:

$$\frac{f(x) \equiv z \Rightarrow \varphi(x, z)}{\begin{array}{ll} & \vartheta(x, in(x)) \\ \bigwedge_{i=1}^{n} & \vartheta(x, y) \wedge \delta_i(y) \Rightarrow \vartheta(x, loop_i(y)) \\ \wedge & \vartheta(x, y) \wedge \delta(y) \Rightarrow \varphi(x, out(y)) \end{array}} \Uparrow$$

Die Implementierung von Hoare-Induktionen in *Expander2* bezieht sich direkt auf Fixpunktinduktion über der Schleifenfunktion g. Dies bedeutet, dass man zunächst eine Korektheitsanforderung an g stellt, die eine noch unspezifizierte Hoare-Invariante ϑ enthält. Auf diese Regel wendet man dann die Fixpunktinduktion an. Die Hintereinanderausführung der beiden Regelanwendungen liefert genau die Konklusion der Hoare-Induktion. Formal gilt also für eine Hoare-Invarianten-Erzeugung mit *Expander2*

$$\frac{f(x) \equiv z \Rightarrow \varphi(x, z)}{\begin{array}{ll} & \vartheta(x, in(x)) \\ \wedge & g(y) \equiv z \wedge \vartheta(x, y) \Rightarrow \varphi(x, z) \end{array}} \Uparrow$$

Diese Regel ist auch korrekt, wenn die Axiome für g nicht dem obigen iterativen Schema entsprechen. Jedeglich f muss dem Schema folgen und durch das Axiom $f(x) \equiv g(in(x))$ definiert sein.

Um nun eine Hoare-Induktion mit *Expander2* durchzuführen, muss eine Formel der Form $f(\vec{x}) = loop(\vec{v}) \Rightarrow conc$ oder $f(\vec{x}) = loop(\vec{v})\{\wedge conc\}$ selektiert werden. Die Ausgangsformel wird dann in die folgende Korrektsheitsanforderung mit Invariante überführt

1. $INV(\vec{x}, \vec{v})$
2. $loop(\vec{y}) = z \wedge INV(\vec{x}, \vec{y}) \Rightarrow conc$

und muss anschliessend mittels Fixpunktinduktion bewiesen werden.

Als Beispiel sei nun das folgende iterative Programm zur Berechnung von Quotient und Rest gegeben.

```
DIVLOOP = NAT then
  defuncts div : nat -> nat*nat
           loop : nat*nat*nat -> nat*nat
```

```
preds    INV : nat * nat * nat * nat * nat
vars     k,n : nat
         x : entry
         L,L' : list
         P : list(list)
axioms   div( x, y ) = loop( 0, x, y )
         loop( q, r, y ) = (q, r) <== x < y
         loop( q, r, y ) = loop( q+1, r-y, y ) <== x>=y
         INV(x, y, q, r, y) <== x = (y*q)+r
```

Die Korrektheitsanforderung ist durch $div\,(X,Y) = (Q,R) \Rightarrow X = (Y * Q) + R \wedge R < Y$ und das iterative "Hauptprogramm" durch $div\,(x,y) = loop\,(0,x,y)$ gegeben. Nun wird die Hoare-Invariante erzeugt und auf die Korrektheitsbedingung angewendet, was zur folgenden Formel führt.

$$INV\,(X,Y,0,X,Y)$$

$$\wedge$$

$$\left(\left(\begin{array}{ll} & loop\,(z_1,z_2,z_3) = z_0 \\ \wedge & INV\,(X,Y,z_1,z_2,z_3) \end{array}\right) \Rightarrow\ (z_0 = (Q,R) \Rightarrow X = (Y * Q) + R \wedge R < Y)\right)$$

Durch Narrowing- und Simplifikationsschritte[14] wird dieser Ausdruck in

$$loop\,(z_1,z_2,z_3) = z_0$$

$$\Downarrow$$

$$(INV\,(x,Y,z_1,z_2,z_3) \Rightarrow (z_0 = (Q,R) \Rightarrow X = (Y * Q) + R \wedge R < Y))$$

überführt. Auf diesen Ausdruck lässt sich nun das Fixpunktinduktonsschema

$$\left(\begin{array}{ll} & loop\,(q_0,r_0,y_0) = (q_0,r_0) \Leftarrow r_0 < y_0 \\ \wedge & loop\,(q_1,r_1,y_1) = z_5 \Leftarrow r1 \geq y_1 \wedge loop\,(q_1+1, r_1-y_1, y_1) = z_5 \end{array}\right)$$

anwenden. Dies führt zu

$$\forall q_0, r_0, y_0, q_1, r_1, y_1, z_5 :$$

$$\left(\begin{array}{ll} & \forall X_0, Y_0, Q_0, R_0 : \\ & \left(\begin{array}{ll} & INV\,(X_0,Y_0,q_0,r_0,y_0) \\ \Rightarrow & (q_0,r_0) = (Q_0,R_0) \\ \Rightarrow & X_0 = (Y_0 * Q_0) + R_0 \wedge R_0 < Y_0 \end{array}\right) \Leftarrow r_0 < y_0 \\ \wedge & \forall X_1, Y_1, Q_1, R_1 : \\ & \left(\begin{array}{ll} & \left(\begin{array}{ll} & INV\,(X_1,Y_1,q_1,r_1,y_1) \\ \Rightarrow & z_5 = (Q_1,R_1) \\ \Rightarrow & X_1 = (Y_1 * Q_1) + R_1 \wedge R_1 < Y_1 \end{array}\right) \Leftarrow r_1 \geq y_1 \\ \wedge & \forall X_2, Y_2, Q_r, R_2 : \\ & \left(\begin{array}{ll} & \left(\begin{array}{ll} & INV\,(X_2,Y_2,q_1+1,r_1-y_1,y_1) \\ \Rightarrow & z_5 = (Q_2,R_2) \\ \Rightarrow & X_2 = (Y_2 * Q_2) + R_2 \wedge R_2 < Y_2 \end{array}\right) \end{array}\right) \end{array}\right)$$

[14] aufgelistet in Beispiel 7.5.2 [6]

Dieser Ausdruck kann nun mittels Anwendung von Simplifikationen und Axiomen zu $True$ abgeleitet werden.

4.3.2 Subgoal-Induktion

Dual zur Hoare-Induktion beschreibt eine Subgoal-Invariante $\psi(y,z)$ einen Zusammenhang zwischen Werten der Schleifenvariable y und Werten der Ausgabevariable z. Die Erzeugung der Subgoal-Invariante folgt dem folgenden Schema:

$$\frac{\begin{array}{ll} & f(x) \equiv z \Rightarrow \varphi(x,z) \\ \wedge & g(y) \equiv z \Rightarrow \psi(y,z) \end{array}}{\begin{array}{ll} & \psi(in(x),z) \Rightarrow \varphi(x,z) \\ \bigwedge_{i=1}^{n} & \psi(loop_i(y),z) \wedge \delta_i(y) \Rightarrow \psi(y,z) \\ \wedge & \delta(y) \Rightarrow \psi(y,out(y)) \end{array}} \Uparrow$$

Die Implementierung von Subgoal-Induktionen in *Expander2* bezieht sich - dual zur Hoare-Induktion - direkt auf Fixpunktinduktion über der Schleifenfunktion g. Dies bedeutet, dass man zunächst eine Korektheitsanforderung an g stellt, die eine noch unspezifizierte Subgoal-Invariante ψ enthält. Auf diese Regel wendet man dann die Fixpunktinduktion an. Die Hintereinanderausführung der beiden Regelanwendungen liefert genau die Konklusion der Subgoal-Induktion. Formal gilt also für eine Hoare-Invarianten-Erzeugung mit *Expander2*

$$\frac{f(x) \equiv z \Rightarrow \varphi(x,z)}{\begin{array}{ll} & \psi(in(x),z) \Rightarrow \varphi(x,z) \\ \wedge & g(y) \equiv z \Rightarrow \psi(y,z) \end{array}} \Uparrow$$

Diese Regel ist auch korrekt, wenn die Axiome für g nicht dem obigen iterativen Schema entsprechen. Jedeglich f muss dem Schema folgen und durch das Axiom $f(x) \equiv g(in(x))$ definiert sein.

Um nun eine Subgoal-Induktion mit *Expander2* durchzuführen, muss eine Formel der Form $f(\vec{x}) = loop(\vec{v}) \Rightarrow conc$ oder $f(\vec{x}) = loop(\vec{v})\,\{\wedge conc\}$ selektiert werden. Die Ausgangsformel wird dann in die folgende Korrektheitsbedingung mit Invarainte transformiert

1. $INV(\vec{v},z) \Rightarrow conc$
2. $loop(\vec{y}) = z \Rightarrow INV(\vec{y},z)$

und muss anschliessend noch mit Fixpunktinduktion bewiesen werden.

5 Anhang

Literatur

[1] Avenhaus, Jürgen; *Reduktionssysteme*; Springer 1995; 3-540-58599-1

[2] Ehrig, Mahr, Cornelius, Große-Rhode, Zeitz; *Mathematisch-strukturelle Grundlagen der Informatik*; Springer 1999; 3-540-41923-3

[3] Lüth, Christoph; *Die Korrektheit von MergeSort*; http://www.informatik.uni-bremen.de/ cxl/lehre/pi3.ws02/msort-proof.pdf

[4] Padawitz, Peter; *Expander2: A Formal Methods Presenter and Animator*; http://ls5-www.cs.uni-dortmund.de/∼peter/Expander2/Manual.html

[5] Padawitz, Peter; *Expander2: Towards a Workbench for Interactive Formal Reasoning*; http://ls5-www.cs.uni-dortmund.de/∼peter/Expander2/Chiemsee.ps

[6] Padawitz, Peter; *Formale Methoden des Systementwurfs - WS 2003/2004*; http://ls5-www.cs.uni-dortmund.de/∼peter/TdP96.ps.gz

[7] Wagner, Hubert; *Logische Systeme der Informatik - Wintersemester 2002/2003*; http://ls1-www.cs.uni-dortmund.de/Lehre/LSI/lsi.pdf

[8] Wirth, Claus-Peter; *Multi Sets*; http://www.ags.uni-sb.de/∼cp/p/multisets/all.ps.gz

[9] Frank, Stephan; *Needed Narrowing / Berechnungsmodell von Curry*; http://uebb.cs.tu-berlin.de/csem00/vortraege/sfrank.ps

Index

(), 6
{}, 6, 7
∗, 6
∗∗, 6
/, 6
:, 6
<, 6
⟨+⟩, 6, 7
≤, 6
>, 6
≥, 6
[], 6
$, 7

0, 6

Ableitung, 13
    korrekt, 14
    Korrektheit, 21
    unkorrekt, 21
all, 6
    zipAll, 6
analytisch, 14, 21
Anker, 12, 18
any, 6
    zipAny, 6
Axiom, 4
    analytisch, 14
    Anwendung, 23
    contracting, 14
    eager, 18
    expanding, 14
    lazy, 18
    synthetisch, 14

Bag, 7, 8
bag, 7
bool, 6

Co-Induktion, 29
contracting, 14
Coresolution
    disjunktive, 24
    konjunktive, 24

DNF, 7
dnf, 7
drop, 7

eager, 18
Entfernen, 22
expanding, 14
Expansion, 25
explizie Induktion, 25

Fixpunktinduktion, 27
    starke, 28
Flattening, 7, 16
fun, 6

Generalisierung, 26
Gentzen Klausel, 9, 15

Haskell, 7
Herbrand
    Expansion, 10
    Modell, 13
    Struktur, 9
    Universum, 9
Hoare-Induktion, 30
Hornklauseln, 11
    verteilt, 12
Hypothese, 25

Implikation, 10
in, 6
    not in, 6
Induktion
    Co-Induktion, 4, 29
    explizite Induktion, 25
    Fixpunktinduktion, 4, 27
      starke, 28
    Generalisierung, 26
    Hoare-Induktion, 4, 30
    Noethersche Induktion, 4, 24
    Schleifeninvarianten, 29
    Subgoal-Induktion, 4, 32
    vollständige Induktion, 25
Inferenzregeln, 4, 21

Infixoperator, 8
Instantiierung, 23
Invariante
    Hoare, 30
    Subgoal, 32

KNF, 11
Kongruenz, 22
Kopieren, 22
Korrektheitsbeweis, 21

λ-Abstraktion, 6
lazy, 18
Lemma von König, 25
Liste, 6, 7
    Einfügen, 6

max, 6
Menge, 6, 7
min, 6
minimize, 7
mod, 6
Multiset, 7

Narrowing, 4, 18, 21, 23
    Co-Prädikat, 19
    Funktion, 19
    Needed Narrowing, 20
    non-narrowable, 4, 18
    Prädikat, 18
    Strategie, 18
Natürliches Beweisen, 14
Noethersche Induktion, 24
non-narrowable, 4, 18
Normalform, 4
Normalisierung, 15

OBBD, 6, 7
obdd, 7
Ordnung
    wohl fundiert, 24

Permutation, 6
Permutator, 7
Polarität, 13, 14
prod, 7
Produkt, 6
Programm, 4
    Ausführung, 5
    Korrektheit, 14
    Verifikation, 5
prozedurale Deutung, 11

Redex, 12, 18
Redukt, 12
Resolution, 24
    disjunktive, 24
    konjunktive, 24
rev, 7
Rewriting, 4, 21, 23
    Regel, 18

Schleifeninvarianten, 29
Schrittlemma, 25
Scope, 23
select, 6
set, 7
Signatur, 4, 5
Simplifier, 15
Simplifikation, 15, 21
    Beispiel, 15
    Disjunktion, 17
    Implikation, 16
    Konjunktion, 16
starke Fixpunktinduktion, 28
Subgoal-Induktion, 32
Subsimption, 16
Subsitution, 19, 20
Subsitutionsfunktion, 21
Substitution, 4
Substitutionsfunktion, 20
Subsumption, 10, 16, 17
suc, 6
sum, 7
Summe, 6
Syntaxbaum, 14
synthetisch, 14, 21

take, 7
Termsplitting, 17
Theorem, 4
Theorembeweisen, 18
Transformation, 4, 5, 21

transitiv, 22

Umkehrung, 22
Unifikation, 4, 20, 23
    Disjunktion, 23
    Konjunktion, 23
Unifikator, 12, 18

vollständige Induktion, 25

Wirkungsbereich, 23

zip, 7
zipAll, 6
zipAny, 6
Zustandsvariable, 4
    current proof, 4
    current signature, 21
    current tree, 18
    current trees, 4, 13, 18
    finPositions, 4
    formula, 4
    proof, 5, 13
    solPosiitons, 4
    treePos, 4