JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Integration von Entscheidungsprozeduren in einen interaktiven Beweisassistenten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

in der Studienrichtung

TECHNISCHE MATHEMATIK

Angefertigt am *Institut für Symbolisches Rechnen (RISC)*

Betreuung:

A.Univ.-Prof. DI Dr. Wolfgang Schreiner

Eingereicht von:

Stadlbauer Markus

Linz, Juni 2008

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis 1

1 Einleitung 4

2 Stand der Wissenschaft 6
2.1 Automatische Entscheidungsprozedur . . . 7
2.1.1 ARIO . . . 8
2.1.2 BarcelogicTools für SMT . . . 8
2.1.3 CVC . . . 9
2.1.4 CVC Lite . . . 9
2.1.5 HTP . . . 10
2.1.6 LPSAT . . . 12
2.1.7 ICS . . . 13
2.1.8 MathSAT . . . 13
2.1.9 SAMMY . . . 14
2.1.10 Simplics . . . 14
2.1.11 STeP . . . 15
2.1.12 TSAT++ . . . 16
2.1.13 UCLID . . . 17
2.1.14 Yices . . . 17
2.1.15 Zusammenfassung . . . 18
2.2 Interaktive Beweisassistenten . . . 20
2.2.1 ACL2 . . . 22
2.2.2 coq . . . 23
2.2.3 HOL 4 . . . 24
2.2.4 Isabelle/HOL . . . 26
2.2.5 PVS (Spezifikations und Verifikations System) . . . 28
2.2.6 Zusammenfassung . . . 30
2.3 Integration von Entscheidungsprozeduren in Beweisassistenten . . . 31

3 Formale Problembeschreibung 33
3.1 Logik der ersten Ordnung . . . 34
3.1.1 Basis Definitionen . . . 34
3.1.2 Syntax von PL1 . . . 35

3.1.3 Semantik von PL1 . . . 39
3.2 Getypte Prädikatenlogik der ersten Stufe . . . 45
3.2.1 Syntax . . . 46
3.2.2 Semantik . . . 49
3.2.3 Benutzerdefinierte Typen . . . 52
3.3 Logik höherer Ordnung . . . 56
3.3.1 Von PL1 zu HOL . . . 56
3.3.2 HOL mit Typen . . . 58
3.3.3 Übersetzung von der Logik der höheren Ordnung mit Typen in die Prädikatenlogik der ersten Stufe . . . 64

4 Software-Schnittstelle 77
4.1 Der RISC ProofNavigator . . . 78
4.1.1 Benutzerinterface . . . 79
4.1.2 Eingabesprache . . . 80
4.2 Entscheidungsprozeduren . . . 83
4.2.1 CVC Lite 2.0 . . . 83
4.2.2 Beispiel . . . 83
4.2.3 CVC 3 . . . 90
4.2.4 CVC Lite 2.0 versus CVC 3 . . . 95
4.3 Übersetzer . . . 96
4.3.1 Built-In Typen NAT . . . 96
4.3.2 Geschachtelte Funktionstypen . . . 96
4.3.3 FORALL Anweisung . . . 97
4.3.4 Indirekt geschachtelte Funktionstypen . . . 97
4.3.5 Sprachelemente . . . 98
4.4 Einschränkung . . . 99

5 Beispiele 100
5.1 Beweis einer Summenformel . . . 100
5.1.1 Deklaration . . . 100
5.1.2 Induktion . . . 104
5.1.3 Instanziierung . . . 106
5.2 Beweis einer polynomialen Gleichheit . . . 109
5.2.1 Deklaration . . . 110
5.2.2 Induktion über x . . . 111
5.2.3 1. Induktion über y . . . 113
5.2.4 2. Induktion über y . . . 115
5.2.5 Autoinstanziierung . . . 117

6 Schlussfolgerung 121

Literatur ii

Abbildungsverzeichnis

2.1 Solver 7
2.2 ARIO Solver 8
2.3 CVC Lite [8] 10
2.4 HTP [50] 11
2.5 STEP [46] 15
2.6 ACL2 System [21] 22
2.7 coq Interface [10] 23
2.8 Theorien in HOL4 [20] 25
2.9 Isabelle Umgebung [32] 27
2.10 Die PVS Umgebung [31] 29

3.1 Interpretation 54

4.1 ProofNavigator 79
4.2 Deklaration 85
4.3 Induktion 87
4.4 Instanziierung 89
4.5 Abgeschlossener Beweis 90

5.1 Deklaration 102
5.2 Auswahl 103
5.3 Induktion 104
5.4 Instanziierung 106
5.5 Abgeschlossener Beweis 108
5.6 Eingabe 109
5.7 Induktion nach x 112
5.8 Induktion nach y 114
5.9 Induktion nach y 116
5.10 Autoinstanziierung 118
5.11 Abgeschlossener Beweis 120

Tabellenverzeichnis

2.1 Zusammenfassung Entscheidungspozeduren 19
2.2 Zusammenfassung Beweisassistent 30

3.1 Terme 36
3.2 Atome 37
3.3 Terme Sematik 40
3.4 Formeln 42
3.5 Gebundene und Freie Variablen 42

Zusammenfassung

Das Ziel dieser Diplomarbeit ist die Einbindung einer automatische Entscheidungsprozedur in ein interaktives Beweissystem. Da die automatische Entscheidungsprozedur CVC 3 nur eine Logik der ersten Ordnung versteht und der interaktive Beweisassist RISC ProofNavigator die Logik der höheren Ordnung verarbeiten kann, musste eine Übersetzung zwischen der Logik ersten Ordnung und die der höheren Ordnung gefunden und verifiziert werden. Als Vorarbeit für die Übersetzungfunktion mussten die Logik ersten Ordnung, eine getypte Logik der ersten Ordnung, die Logik der höheren Ordnung und die getypte Logik der höheren Ordnung definiert werden. Diese Übersetzungsfunktion wurde dann als Schnittstelle zwischen der automatischen Entscheidungsprozedur und dem interaktiven Beweissystem eingebunden.

The goal of this thesis represents the integration of an automatic decision procedure into an interactive proof system. The given automatic decision procedure called CVC 3 works with first order logic but the RISC Proof Navigator needs high order logic as input language. Therefore a model had to be implemented for the translation and verification of first order to high order logic. As a preliminary work for the translation function, first order logic, typed first order logic, high order logic and typed high order logic had to be defined. Finally, the translation function was integrated as an interface between the decision procedure and the interactive proof system.

Danksagung

Ein herzliches Dankeschön an alle die mich während meines Studiums unterstützt haben!

Dieser Dank geht besonders an

A.Univ.-Prof. DI Dr. Wolfgang Schreiner für die exzellente Betreuung und für die Ermöglichung dieser interessanten und spannenden Diplomarbeit

meiner Familie, insbesondere meiner Mutter Gerlinde Stadlbauer für jegliche Unterstüzung und Motivation

meiner Freundin Daniela für ihre unermessliche Geduld und Aufmunterung

meinen Studienkollegen Johannes Watzl und Paul Kainz für die gemeinsame Zeit

1 Einleitung

Interaktive Beweisassistenten sind Computerprogramme, die in der Lage sind, Formeln einer bestimmten Logik zu verarbeiten und den Benutzer bei der Untersuchung der Gültigkeit einer Formel zu unterstützen. Dies geschieht Schritt für Schritt und stets in Interaktion mit dem Benutzer. Mit solchen interaktiven Beweisassistenten ist es möglich, Beweise für Formeln zu finden, die in Logiken mit höherer Ordnung geschrieben sind. Außerdem bieten sie oft die Möglichkeit, bereits geführte Beweise einzubeziehen, wenn neue Formeln bewiesen werden sollen. Abhängig von der Basislogik kann das Entscheidungsproblem eines Theorems von trivial bis unlösbar variieren. Für die Verifikation von Formeln sind automatische Entscheidungsprozeduren „Solver“ notwendig, die in den interaktiven Beweisassistenten eingebunden sind. Die meisten Solver wandeln die Eingangssprache in eine aussagenlogische Formel um, deren Erfüllbarkeit von der eigentlichen Entscheidungsprozedur geprüft wird. Als Ausgabe erhält man eine der Antworten „gültig“, „nicht gültig“ oder „unentscheidbar“. Die meisten automatische Entscheidungsprozeduren verstehen die Logik der ersten Ordnung. Daher benötigt man eine Übersetzungsfunktion, die zwischen dem interaktive Beweisassistenten und den automatischen Entscheidungsprozeduren die Kommunikation regelt.

Der RISC ProofNavigator [39] wird als interaktives Beweissystem an der Johannes Kepler Universität Linz von Wolfgang Schreiner entwickelt. In diesem System können verschiedene Kombinationen von Methoden ausgewählt werden, um mathematische Thesen zu beweisen. Die Eingabe in dieses Beweissystem erfolgt über die Logik der Höheren Ordnung. In diesem System ist die automatische Entscheidungsprozedur CVC Lite 2.0 für die Verifikation von Aussagen eingebunden. CVC Lite [8] wird an der Universtät New York von Clark W. Barrett entwickelt. CVC Lite ist die überarbeitete Neuimplementierung des Cooperating Validity Checker (CVC), der Begriff „Lite“ steht für einen schlankeren Aufbau als für das ursprüngli-

che CVC. CVC Lite unterstützt als Theorien (lineare und nichtlineare) Real- und Integerarithmetiken, uninterpretierte Funktionen, Arrays, Records, und Bitvektoren. Im Gegensatz zu anderen Entscheidungsprozeduren unterstützt CVC Lite als Eingabesprache eine Logik höherer Ordnung, was die Einbindung in den RISC ProofNavigator erleichterte, allerdings ist damit die Verwendung anderer Enscheidungsprozeduren, die nur Logiken erster Ordnung unterstützen (zum Beispiel auch die aus CVC Lite vorangegangene Entscheidungsprozedur CVC 3 nicht möglich).

Ziel dieser Diplomarbeit ist es, die Einbindung anderer Entscheidungsprozeduren als CVC Lite in den RISC ProofNavigator zu ermöglichen. Der Kern dieser Diplomarbeit befasst sich daher mit der Übersetzung von der Logik der höheren Ordnung [23] in die Logik der ersten Ordnung. Im formalen Teil werden die unterschiedlichen Logiken definiert und eine Übersetzung von einer höheren Logik in die Logik der ersten Ordnung realisiert und ihre Korrektheit bewiesen. Im praktischen Teil wird die automatische Entscheidungsprozedur CVC 3 in den interaktive Beweisassistenten RISC ProofNavigator eingebaut.

Die Arbeit ist wie folgt aufgebaut:
Im Kapitel 2 „Stand der Wissenschaft“ werden 14 automatische Entscheidungsprozeduren und 5 interaktive Beweisassistenten vorgestellt. Als Zusammenfassung dieses Kapitels werden in einer Tabelle die Beweisverfahren auf ihre Funktionalität verglichen. Das Kapitel 3 „Formale Problembeschreibung“ definiert den Aufbau der Aussagenlogik, ausgehend von der Logik der ersten Ordnung über die getypte Logik der ersten Ordnung bis zur Logik der höheren Ordnung und der getypten Logik der höheren Ordnung. Als Abschluss diese Kapitels wird der Kern dieser Diplomarbeit, die Übersetzung zwischen der getypten Logik der höheren Ordnung und der getypten Logik der ersten Ordnung definiert und ihre Korrektheit bewiesen. Im Kapitel 4 „Software Schnittstelle“ wird beschrieben, wie CVC 3 in den RISC ProofNavigator, eingebunden wird. Abschließend werden noch einige Beispiele vorgestellt, die die Funktionalität des Verfahrens demonstrieren.

2 Stand der Wissenschaft

Interaktive Beweisassistenten sind in der Lage, Formeln einer bestimmten Logik [17] zu verarbeiten, um die Gültigkeit der Formel festzustellen und dabei den Benutzter zu unterstützen. Entscheidungsprozeduren sind automatische Verfahren die eine Formel als Eingabe verarbeiten. Diese Formel wird geprüft und als Ausgabe liefert die Entscheidungsprozedur „gültig“ oder „nicht gültig“. Diese Prozeduren sind meist auf spezielle Theorien zugeschnitten und daher werden oft mehrere Entscheidungsprozeduren für einen Beweis verwendet. Der Vorteil gegenüber interaktiven Beweisassistenten liegt in der Automatisierbarkeit und daher in der Benutzerfreundlichkeit von Entscheidungsprozeduren.

In diesem Kapitel werden zuerst die Unterschiede zwischen automatischen Entscheidungsprozeduren und interaktiven Beweisassistenten beschrieben. Der Abschnitt 2.1 „Automatische Entscheidungsprozeduren“ behandelt allgemeine die Erfüllbarkeit bzw. der Gültigkeit einer Formel. Die formale Beschreibung dieser Probleme wird im Kapitel 3 behandelt. In den Abschnitten 2.1.1 bis 2.1.14 werden die für diese Diplomarbeit relevanten Entscheidungsprozeduren vorgestellt und die Vor- und Nachteile dieser Prozeduren detailliert besprochen. Der Abschnitt 2.2 stellt die Funktionsweise interaktiven Beweisassisenten vor. In den Abschnitten 2.2.1 bis 2.2.5 werden im Detail einzelne interaktiven Beweisassisenten beschieben. Zum Schlussabschnitt 2.3 wird die Integration von Entscheidungsprozeduren in Beweisassisenten besprochen [36].

2.1 Automatische Entscheidungsprozedur

Eine automatische Entscheidungsprozedur implementiert einen Algorithmus, der als Eingabe eine Formel einliest und diese anschließend zu beweisen versucht. Nach Abschluss der Prüfung gibt er „gültig“ oder „nicht gültig“ zurück. Man kann die Entscheidungsprozedur veranschaulichen, indem man die Variablen der Formel in einen Zustandsraum darstellt. Während der Ausführung der Prozedur kann sich dieser Zustandsraum in Verzweigungen unterteilen. Die meisten Solver wandeln die Eingangssprache in eine aussagenlogische Formel um. Diese wird dann in der eigentlichen Entscheidungsprozedur geprüft, dazu kann sie einen SAT-Checker aufrufen, der die aussagenlogische Formel prüft ob die Formel „gültig“ oder „nicht gültig“ ist. Einer der am häufigsten verwendeten Algorithmen ist DPLL (Davis Putnam Logemann Loveland). Dieser ist ein vollständiger „backtracking“ Algorithmus, der die Erfüllbarkeit von Satzlogiken in der Konjunktiven Normalform entscheidet. Dieser Algorithmus wird im Detail später beschrieben.

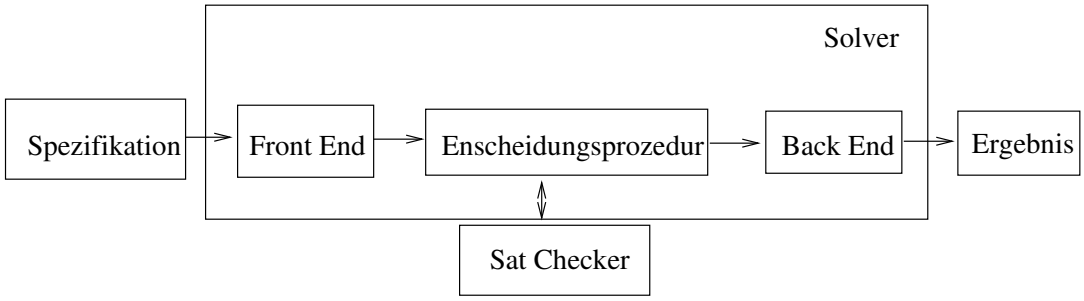

Abbildung 2.1: Solver

Als SATisfiability Modulo Theories (SMT) bezeichnet man ein Entscheidungsproblem für logische Formeln. Typische Theorien von SMT sind Gleichheit uninterpretierten Funktionssymbole (EUF), lineare Arithmetik über den Integern oder den Reals, Arrays und Listen. In den folgenden Abschitten werden einzelne Solver im Detail besprochen. Die Solver die in diesem Kapitel beschrieben werden verstehen das SMT-LIB [49] Format.

2.1.1 ARIO

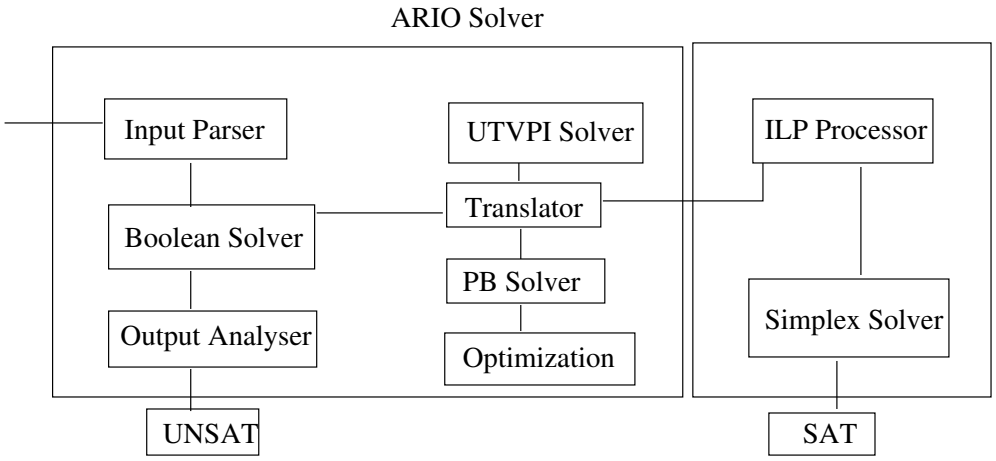

Abbildung 2.2: ARIO Solver

ARIO [4] wurde am Elektrotechnik- und Informatikinstitut der Universität Michigans von Hossein Sheini entwickelt. Es ist ein vollständig scalierbarer SAT Solver um SATisfiability Modulo Theorien (SMT) und Optimierungsprobleme für Booleans, Pseudo Booleans (PB) [42] und konstante lineare Integers zu lösen. Dieser Solver kann das SMT-LIB Format, IMACS Format für reine CNF Probleme, LP Format für Integers und SMV Format für Modulo Theorie Probleme lesen. Der Parser kann auch uninterpretierte Funktionen umwandeln, die die Ackermann Methode verwenden. Als Ausgabe wir SAT oder UNSAT zurückgegeben.

2.1.2 BarcelogicTools für SMT

Dieses Programm ist ein SMT Solver der an der Technischen Universität Kataloniens entwickelt wird. BarcelogicTools [6] wird für logisches Verifikationsaufgaben, Überprüfungsanwendungen und Optimierungsprobleme programmiert. BarcelogicTools für SMT benutzt eine Implementierung des DPLL „Frameworks“ [30]. Zur Zeit unterstützt Barcelogic Tools für SMT unterschiedliche Logiken für Reals und Integers, Gleichheit uninterpretierten Funktionssymbolen (EUF) und interpretierten Sufix und Präfix Funktionssymbolen, oder Kombinationen dieser Theorien. Die Eingangsdaten müssen im SMT-LIB Format sein. Die Ausgabe gibt „unsatisfiable“ für ein unerfülltes Problem oder „satisfiable“ für ein erfülltes Problem zurück.

2.1.3 CVC

CVC [7] entscheidet die logische Gültigkeit von quantorenfreien Formeln in der klassischen Logik der ersten Ordnung mit Gleichheit und mit zusätzlichen bestimmten Hintergrundtheorien. Die Formeln können AND, OR, NOT, und $\Rightarrow$ beinhalten. Sie können auch Gleichungen zwischen bestimmten mathematischen Bestimmungen und atomaren Formeln wie $x < y$ enthalten. In der gegenwärtigen Version sind die Hintergrundtheorien dafür lineare Arithmetik, Arrays, Listen und Bäume. CVC [47] verknüpft unabhängige Entscheidungsverfahren für diese Theorien in ein Entscheidungsverfahren für die Kombination der Theorien. Der Beweiser ist so angelegt, dass neue unabhängige Entscheidungsverfahren dem System hinzugefügt werden können. Er erreicht dieses, indem er eine Version des Nelson-Oppen-Algorithmus implementiert. Die Ausgabe gibt VALID oder ein Gegenbeispiel zurück.

2.1.4 CVC Lite

CVC Lite [8] wird an der Universtät New York von Clark W. Barrett [9] entwickelt. CVC Lite ist die überarbeitete Neuimplementierung des Cooperating Validity Checker (CVC). Lite steht für einen schlankeren Aufbau als für das ursprüngliche CVC. Der Beweiser ist als C++ Bibliothek programmiert. Es ist eine völlig automatisierte Entscheidungsprozedur für die Logik der ersten Ordnung, der Quantoren, partielle Funktionen und Prädikatuntertypen unterstützt. Die derzeit implementierten Theorien sind (lineare und nichtlineare) Real- und Integerarithmetiken, uninterpretierte Funktionen, Arrays, Records, und Bitvectoren. CVC Lite bietet auch ein interaktives Werkzeug, das auf dieser Bibliothek aufgebaut wird. Dies Werkzeug hat eine gut lesbare Eingabesprache und einen einfach zu bedienenden Befehlsinterpreter. Das Ziel der Entwicklergruppe ist, alle Funktionen von CVC einzufügen, sowie einige weitere Funktionen hinzuzufügen. So unterstützt CVC Lite Quantoren und hat die Möglichkeit partiell definierte Funktionen zu verarbeiten. Außerdem kann der Beweiser um weitere Theorien erweitert werden. Zur Zeit versteht CVC Lite eine Eingangssprache die „presentation language“ genannt wird. Die Idee dieser Syntax kommt von PVS und SAL. Die Ausgabe gibt VALID oder ein Gegenbeispiel zurück.

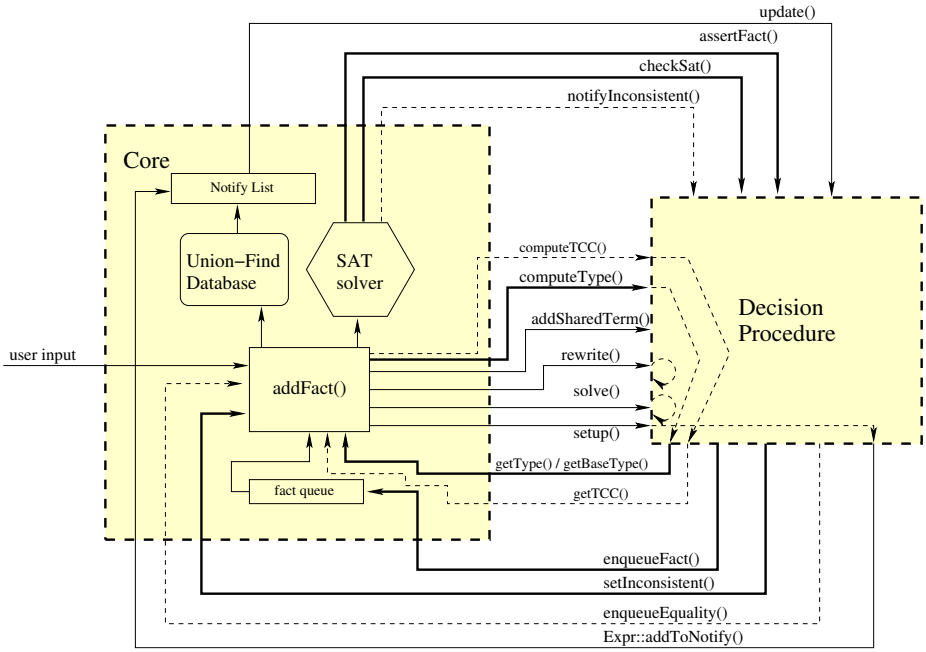

Abbildung 2.3: CVC Lite [8]

2.1.5 HTP

Der Heuristische Lehrsatz Beweisassistent [50] wird in Redwood City (CA) von Kenneth Roe [34]implementiert. Dieser Beweisassistent versteht das SMT-LIB Format als Eingabe und produziert als Ausgabe „sat“ (gültig), „unsat“ (nicht gültig) oder „unknown“ (unbekannt).

Dieses Programm versteht linearer Arithmetik, Arrays und Gleichheit zwischen Ausdrücken uninterpretierten Funktionssymbolen. Dieses Verfahren ist ähnlich zu ICS, SVC, CVC, und CVC Lite. SVC ist ein Vorgänger von CVC. Der Unterschied zu anderen Beweisassistenten besteht in der Auswahl der „case splits“:

- „unate case detection“ . Wenn es bei diesem Algorithmus ein atomares Prädikat gibt, mit dem man diesen Satz als richtig oder falsch deklarieren kann, ist dieses automatisch den anderen vorzuziehen. Um diese „unates“ schnell zu finden, kompiliert der heuristische Lehrsatzbeweisassistent zuerst eine Liste aller atomaren Prädikate im Lehrsatz. Er kommentiert dann jeden Knoten mit vier Sätzen (dargestellt als Bitvektoren). Der erste ist der Satz von Atomprädikaten, welche den kompletten Lehrsatz als wahr deklarieren. Der zweite ist der Satz von Prädikaten welche den kompletten Lehrsatz als falsch deklarie-

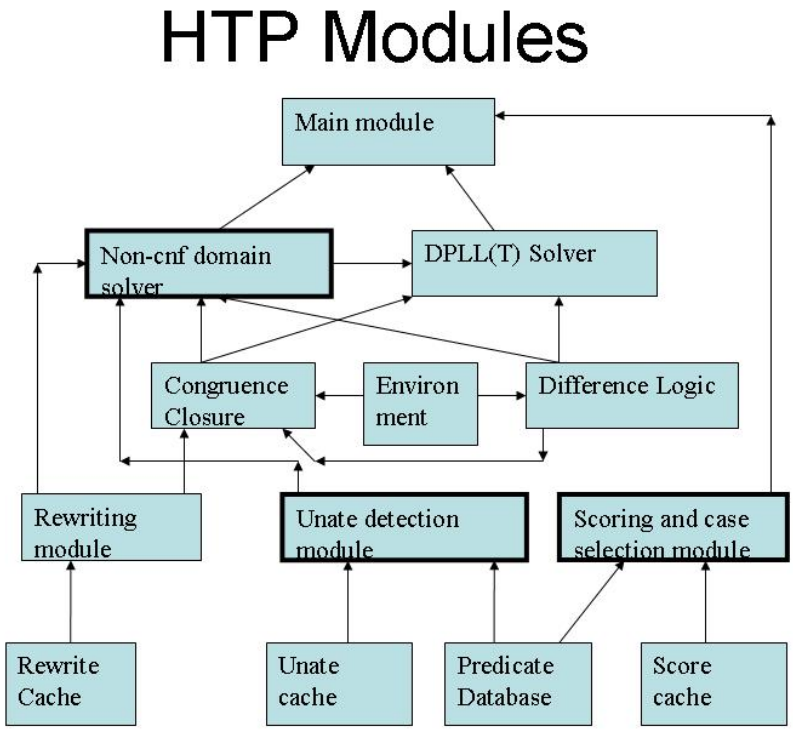

Abbildung 2.4: HTP [50]

ren. Der dritte und vierte sind die Sätze von Prädikaten, die den kompletten Lehrsatz als wahr oder falsch deklarieren. Diese Sätze werden als Bitvektoren dargestellt.

- Wenn kein „unate case splits“ existiert, dann bewertet das System jedes Atomprädikat. So kann das System erkennen welches den größten Teil des „rewritings“ produziert. Eine Methode dies zu erreichen, ist jedes Prädikat temporär als wahr oder falsch zu deklarieren um zu sehen wie viel „rewritings“ gemacht wurde. Dies ist leider sehr langsam, daher wird der heuristische Lehrsatz Beweisassistent durch Zwischenspeicherung und Abspeichern jedes Subausdrucks in den Cachespeicher beschleunigt. Das Caching reduziert den Aufwand auf einen Bruchteil der ursprünglichen Zeit. Zusätzlich zum Algorithmus des Heuristischen Lehrsatz Beweisassistent stellt er ein „Treeview“ Dienstprogramm, zur Betrachtung der Spur in einem Baumformat, zur Verfügung.

2.1.6 LPSAT

LPSAT [53] wurde an der Universität von Washington von Steven Wolfman [54] entwickelt. Dieser Solver versucht die Vorteile, zweier Systeme in einem Projekt zu vereinen:

- Lineareprogrammierung (LP)
- Boolean Erfüllbarkeit (SAT)

Die lineare Programmierung wird verwendet um Probleme darzustellen, die sich mit numerischen Werten und Einschränkungen befassen. LP Probleme können schnell gelöst werden, und sind für globale Analysen verantwortlich. Boolean Erfüllbarkeit wird verwendet, um logische Probleme darzustellen. SAT kann auch Disjunktionen darstellen, welches LP nicht kann, aber SAT ist ein komplizierterer Solver für aufwendigere Probleme als LP. LPSAT verbindet diese Systeme, um eine neue Darstellungsmethoden zu schaffen und um logische Disjunktionen sowie numerische Einschränkungen effizient zu behandeln. Das LPSAT-Projekt besteht aus:

- einem Solver Programm genannt „enigmatically termed“
- einer Eingangssprache genannt LCNF [52]

- einem Anwendungssystem, das LPSAT verwendet, um metrische Planungsprobleme zu beheben
- einer Ausgabe die SAT, UNSAT, TIME LIMIT EXPIRED zurück gibt

2.1.7 ICS

Integrated Solver und Canonizer (ICS) [35] ist eine ist ein Entscheidungsverfahren, das an der SRI International von Harald Rueß entwickelt wurde. Es wurde in Ocaml implementiert. ICS ist ein Entscheidungsverfahren für eine Teillogik der ersten Ordnung. Das Ziel des Projektes ist, formale Softwarespezifikation und Softwareüberprüfung in die industriellen Softwareentwicklungsprozesse zu integrieren. Bei diesem Projekt wird versucht die Überprüfung so transparent wie möglich zugestalten. ICS unterstützt Gleichungen (G), Ungleichungen (U), rational und ganzzahlig lineare Arithmetik (l.A.), Tupel (Listen), boolesche Konstanten, Arrays und Bitvektoren, Formeln müssen aber quantorenfrei sein. ICS kann als eine eigenständige Anwendung verwendet werden, die Formeln interaktiv liest, und auch als eine Bibliothek in jeder Anwendung eingebunden werden kann. Die Liste von interpretierten Theorien ist nach oben offen, dass heißt es können neue Theorien zu ICS hinzugefügt werden, solange sie canonizable und algebraisch lösbar sind. Der modulare Entwurf von ICS, sowohl die zu Grunde liegenden Algorithmen als auch ihre Implemenierung, unterstützt solche Erweiterungen. ICS entscheidet erfüllbare Satzprobleme mit Literalen in der Kombination von oben beschriebenen Theorien. Der zu grunde liegende Verifikationsalgorithmus verbindet die ICS Kernfähigkeiten mit einem modernen SAT Solver, der Paradigmas des „Lazy“ Lehrsatzbeweisers [26] verwendet. ICS kann über die command shell oder interaktiv verwendet werden.

2.1.8 MathSAT

MathSAT [24] wird von Marco Bozzano [18] entwickelt. Es ist ein auf DPLL beruhendes Entscheidungsverfahren für das SMT Problem. Dieser Solver unterscheidet zwischen vier Theorien:

- Gleichheit und uninterpretierten Funktion (EUF)

- Trennungslogik (SEP)
- Linearearithmetik über den Reals (LA(R))
- Linearearithmetik über den Integer Zahlen (LA(Z))

Das Ausgabeformat ist nicht näher beschrieben.

2.1.9 SAMMY

Sammy [15] wird an der Universität Iowas entwickelt und ist in OCaml und C geschrieben. Der Theorie Solver von Sammy verwendet die kombinierten Entscheidungsverfahren des CVC Lite Solvers. Durch eine Auswahl an den Bibliotheksfunktionen von CVC Lite kann man aus CVC Lite einen Theorie Solver für den DPLL Algorithmus konstruieren. Der DPLL Algorithmus [48] von Sammy beruht auf dem Programm der SATBOX, die auf die SATO Aussagenlogik zurückzuführen ist. Das SATBOX Modul für DPLL Operationen ist für Satzübertragung, lernfähige Lemma, und backjumping verantwortlich. Es ist möglich Sammy mit freien Funktion oder Prädikat Symbolen zu erweitern. Sammy akzeptiert andererseits willkürliche Logik der ersten Ordnung im SMT-LIB Format. CNF freie Formeln werden von einem zusätzlichen Wrapper behandelt, der hautsächlich Basissätze von einer Eingangsformel erstellt. Die Ausgabe besteht aus richtig, falsch (mit Gegenbeispiel) oder timeout.

2.1.10 Simplics

Simplics [44] wird an dem Computer Science Laboratory, SRI International von Bruno Dutertre [12] und Leonardo de Moura entwickelt. Es wird haupsächlich in Ocaml implementiert. Als Eingabe sind beliebige Formeln aus der Sprache der Prädikatenlogik erster Ordnung erlaubt. Simplics ist ein Nachfolger des „Integrated Solver und Canonizer“ (ICS). Dieses Programm hat sich aber im Gegensatz zu ICS auf einen kleinen Teil von Aufgaben spezialisiert. Es ist ein Werkzeug, um die Gültigkeit der Boolean Kombination vom linearen beschränkten arithmetischen Real zu entscheiden. Die beabsichtigte Hauptanwendung von Simplics ist beschränktes Prüfen von Modellen unendlicher Übergangsystemen.

2.1.11 STeP

Der Stanford Temporal Prover (STeP) [46] wird von der REACT Forschungsgruppe entwickelt. Es unterstützt die computergestützte formelle Überprüfung von reaktiven, „real-time“ [11] und hybriden Systemen die auf Basis der temporären Spezifizierung beruhen. Gegenüber den meisten Systemen für die temporäre Verifikation ist STeP nicht auf Endzustandsysteme eingeschränkt. Die Ausgabe gibt entweder proof oder ein Gegenbeispiel zurück. STeP verbindet die Modellüberprüfung mit deduktiven Methoden, die Verifikationen einer breiten Klasse von Systemen, einschließlich parametrisiert (N-Komponenten) „circuit designs“, parametrisiert (N-Prozess) Programme, und Programme mit unendlichen Datenbereichen erlaubt.

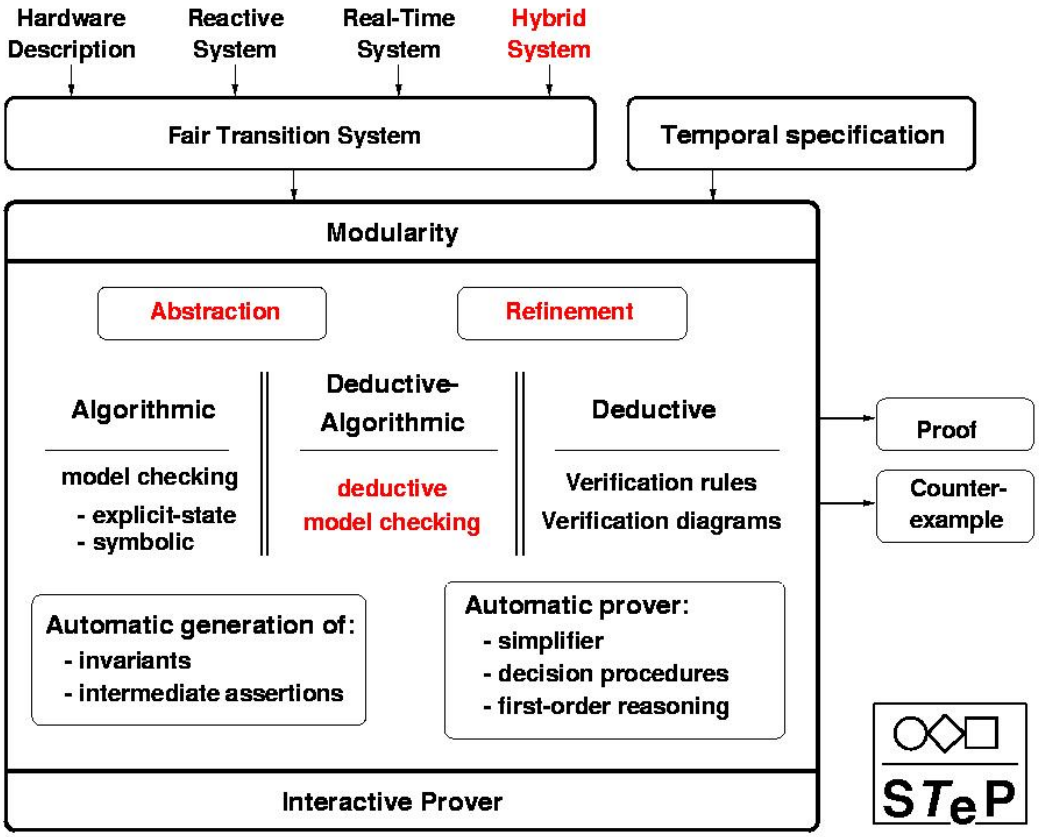

Abbildung 2.5: STEP [46]

STeP stellt eine integrierte Programmgruppe für Vereinfachungs- und Entscheidungsverfahren zur Verfügung, um die Gültigkeit einer großen Klasse der Logik der ersten Ordnung und der temporären Formeln automatisch zu überprüfen. Die-

ser Grad des automatisierten Herleitens ist beabsichtigt, um die meisten Überprüfungszustände rationell zu behandeln, die in der deduktiven Überprüfung entstehen. Die elementaren Eingaben sind ein reaktives System (die eine Hard- oder Softwarebeschreibung sein kann), ausgedrückt als ein Übergangsystem, hergestellt durch ein Transitionsystem, und einer prüfenden Systemeigeigenschaft,
repräsentiert durch eine temporäre logische Formel. Überprüfung kann durch den Modellprüfer oder durch deduktive Mittel durchgeführt werden. Benutzergesteuert kann man Zwischenbehauptungen oder Überprüfungsdiagramme zur Verfügung stellen. In jedem Fall ist das System für das Erzeugen und Prüfen, aller erforderlichen Verifikationszustände verantwortlich. Ein interaktiver "Gentzen-styleLLehrsatzbeweiser und ein auf Lösung basierender Beweisassisent sind verfügbar. Diese verifizieren nicht automatisch hergeleitete Überprüfungszustände. Es gibt Techniken, um Teile der Beweissuche auf höchster Ebene zu automatisieren. Diese können lange oder sich wiederholende Folgen von Beweiskommandos beinhalten.

2.1.12 TSAT++

TSAT++ [14] wurde von Claudio Castellini implementiert. Es versteht als Eingangsformat Formeln der Trennungslogik der Kausal Normalform, darunter gehören arithmetische Operationen (+,-), arithmetische Prädikate (=,! =,<,<=,>,>=) und Booleans (Konjunktion, Disjunktion, Negation). TSAT++ [1] wurde auf der Basis von SAT (ähnlich SL) implementiert. Die Ausgabe wird durch unsat, sat ,timeout und userbreak beschrieben. Zusätzlich wird noch die benötigte Zeit ausgegeben

Die Charakteristiken für mehrere Optimierungen sind:

- (IS) preprocessing
- verallgemeinertes „early-pruning“
- Modellverminderung mittels selbststart, „prime implicants evaluation“
- beste Entdeckung der Lösung
- zwei Literal Beobachtungen, konfliktgeleitetes „backjumping“, „1-UIP“ Lernen, VSIDS Heuristik

2.1.13 UCLID

UCLID („Euklid“) [43] wurde von Shuvendu Lahiri und Sanjit Seshia an der Carnegie Mellon Univesität von Pittsburgh entwickelt. Es ist ein Programm, um die Korrektheit von Hard- und Softwaremodellen zu analysieren. Dieses Programm wird verwendet, um unendliche Zustandsysteme von Variablen des Typs Integer, Boolean, Funktionen und Arrays nachzuprüfen. Es gibt zwei Hauptanwendungsgebiete von UCLID:

- Als ein Verifizierer, für „term-level“ begrenzte Modellprüfung, Ähnlichkeitsüberprüfung, deduktive Überprüfung, und Basis Predikat Abstraktionsprüfung.
- Als ein eigenständiges Entscheidungsverfahren für die Theorien von uninterpretierten Funktionen und Gleichheiten, Integer Linearearithmetik und Arrays. UCLID kann auch beschränkte Quantoren Formen behandeln.

Die Entscheidungsprozedur arbeitet in der Logic von „Counter Arithmetik mit Lambda Ausdrücken und uninterpretierten Funktionen“ (CLUF) [13]. Ausgabe besteht aus „VALID“ oder „COUNTEREXAMPLE“.

2.1.14 Yices

Yices [45] ist ein Entscheidungsverfahren, das am Computer Science Laboratory SRI International von Leonardo de Moura [25] entwickelt wird. Es entscheidet die Erfüllbarkeit von quantorenfreien Formeln und die Theorie der ersten Ordnung, die sowohl uninterpretierte Funktionssymbole als auch interpretierte Symbole und Kombination von diesen einschließt. Die Real und Integerarithmetik, rekursiver Datentypen, Records, extensional Arrays, und Lambda Ausdrücke werden ebenfalls unterstützt. Abhängige Typen werden für die Anzeige von Eigenschaften von uninterpretierten Funktionssymbolen unterstützt. Yices wird in C++ implementiert. Es benutzt die Nelson Oppen Methode um Entscheidungsverfahren zu kombinieren. Es beruht auf einem verallgemeinerten Suchprogramm, welche verschiedene Arten von „case splits“ und Einschränkungsfortpflanzungsregeln unterstützt. Yices sucht, welche Atome irrelevant sind um die Erfüllbarkeit der ganzen Formel zu bestimmen.

2.1.15 Zusammenfassung

Es exisitieren viele automatische Entscheidungsprozeduren, deren Funktionsumfang und Handhabung sich sehr unterscheiden. In der vorangegangen Punkten habe ich Ario, Barcelogic Tools, CVC Lite, HTP, MathSAT, SAMMY, Simplics, Yices, CVC, ICS, LPSAT, STEP, TSAT++ und UCLID näher vorgestellt. Folgende Tabelle fasst die zuvor erwähnten Solver zusammenfassen.

Erklärung der Kürzel

LA(R)	Lineare Arithmetik über den Real
LA(I)	Linear Arithmetik über den Integer
B	Boolean
AR	Arrays
L	Listen
U	uniterpretierte Funktionen
A(I)	Integer Arithmetik
NL	Power Products (nichtlineare Arithmetik)
P	Products
COP	Coproducts (direkte Summen)
ARR	Funktionale Arrays
BV	fixed-sized Bitvektoren
PSET	Propositional Mengen
APP	Funktionale Abstraktion und Applikation
RD	Rekursive Datentypen

Name	Eingabeformat	Theorien	Ausgabe	Referenz
ARIO	PL1 ohne Quantoren	U, LA(I)	SAT, UNSAT	[4]
BarcelogicTools	eigene Logic	LA, U, A(I)	unsatisfiable, satisfiable	[6]
CVC Lite	PL1	Arrays, integer Arithmetic, Listen	VALID, COUNTER-EXAMPLE	[8]
HTP	Boolean Logik	A(I), A, U	SAT, UNSAT	[50]
MathSAT	Separation Logik	U, LA(R), LA(I)	n.a.	[24]
SAMMY	PL1	SMT-LIB	valid, invalid, timeout	[15]
Simplics	PL1	B, LA(R)	n.a.	[44]
Yices	PL1 ohne Quantoren	U, BV, AR, RD	satisfiable, unsatisfiable, unknown	[45]
CVC	PL1 mit Quantoren	A, LA(R), LA(I), Listen, Records, Bitvektoren	VALID, COUNTER-EXAMPLE	[7]
ICS	PL1 ohne Quantoren	U, LA(R), LA(I), NL, P, COP, ARR, BV, PSET, APP	SAT, UNSAT	[35]
LPSAT	PL1	A, A(I), L	SAT, UNSAT, T. L. EXPIRED	[53]
STEP	PL1	SMT-LIB	proof, counterexample	[46]
TSAT++	Separation Logik	SMT-LIB	SAT, UNSAT, timeout	[14]
UCLID	PL1 ohne Quantoren	A(I), B, U, A	VALID, COUNTER-EXAMPLE	[43]

Tabelle 2.1: Zusammenfassung Entscheidungspozeduren

2.2 Interaktive Beweisassistenten

Interaktive Beweisassistenten sind Computerprogramme, die in der Lage sind, Formeln einer bestimmten Logik zu verarbeiten und den Benutzer dabei zu unterstützen, die Gültigkeit der Formel festzustellen. Dies geschieht auf Syntax-Ebene. Hierfür werden Regeln aus einem Kalkül auf die entsprechende Formel angewandt. Dies geschieht Schritt für Schritt, stets in Interaktion mit dem Benutzer. Mit solchen interaktiven Beweisassistenten ist es möglich, Beweise für Formeln zu finden, die in Logiken mit höherer Ordnung geschrieben sind. Außerdem bieten sie oft die Möglichkeit, bereits geführte Beweise mit einzubeziehen, wenn neue Formeln bewiesen werden sollen.
Abhängig von der Basislogik kann das Entscheidungsproblem eines Theorems von trivial bis unlösbar variieren. Für den Fall der Aussagenlogik ist das Problem entscheidbar (d.h. ein Beweis ist immer erzeugbar, wenn das Theorem logisch gültig ist), allerdings NP-vollständig. Dagegen ist Prädikatenlogik erster Stufe lediglich semi-entscheidbar, das heißt unter Verwendung von unbeschränkten Zeit- und Speicherressourcen kann ein gültiges Theorem bewiesen, ein ungültiges allerdings nicht immer als solches erkannt werden. Trotz dieser theoretischen Grenzen sind praktische Theorembeweisassistenten implementiert worden, die viele schwierige Probleme in diesen Logiken lösen können.
Der Vorteil beim Einsatz von interaktiven Beweisen ist die Unterstützung und Kontrolle bei den Umformungen. Interaktive Beweisassistenten formen Formeln durch Anwendung von Regeln aus einem Kalkül um und überprüfen dadurch ihre Gültigkeit. Diese Umformung geschieht durch den Benutzer, der angibt, welche Regeln des Kalküls verwendet werden. Der interaktive Beweisassistent sorgt dann dafür, dass die Regel angewendet wird und zeigt die zu bearbeitende Restformel an. Interaktive Beweisassistenten sind meist leicht auf andere Logiken erweiterbar, indem einfach ein neuer Kalkül als Regelmenge mit eingebunden wird.
Sie erfordern Hinweise für die Beweissuche, die von einem menschlichen Benutzer dem Beweissystem gegeben werden müssen. Abhängig vom Automatisierungsgrad kann dann ein Theorembeweisassistent im Wesentlichen auf einen Beweisprüfer reduziert werden, oder selbstständig bedeutsame Teile der Beweissuche automatisch durchführen. Interaktive Beweisassistenten werden mittlerweile für vielfältige Aufgaben eingesetzt, deren Anwendungsbereich von reiner Mathematik bis zu Problemen der Programmverifikation reicht. Bedeutende mathematische Bewei-

se, die durch einen Theorembeweisassistenten überprüft wurden, sind der Beweis des Vier-Farben-Satzes durch Georges Gonthier, sowie der (zur Zeit unvollendete) formalisierte Beweis der Keplerschen Vermutung durch das „Flyspeckproject“. Aber auch automatische Theorembeweisassistenten haben mittlerweile einige interessante und schwierige Probleme abgearbeitet, deren Lösung in der Mathematik längere Zeit unbekannt war. Leider gibt es auch einige Nachteile. So ist es meist nötig jeden Schritt per Hand auszuführen, wodurch das Finden von komplizierten Beweisen oft viel Zeit benötigt. Oft ist das Bedienen auch nicht ohne tiefere Kenntnisse möglich. Der Benutzer muss stets Kenntnisse über den verwendeten Kalkül haben. Außerdem ist es nötig auch die Implementierung des interaktiven Beweisassistenten zu kennen, da es sonst nur schwer möglich ist, einen bestimmten Beweis damit umzusetzen.
Aus der Natur von interaktiven Beweisassistenten ergibt sich bereits eine gute Lesbarkeit der Beweise. Die Ausdrucksstärke ist im Vergleich zu anderen Methoden sehr hoch. Es ist mit interaktiven Beweisassistenten möglich Beweise in der Logik höher Ordnung zu führen. Dies geht jedoch zu Lasten der Zeit, die für einen Beweis benötigt wird. Die produzierten Beweise sind korrekt (in ihrem Kalkül). Der interaktive Beweisassistent überprüft die Korrektheit, der durchgeführten Schritte während des Führens des Beweises. [28]

Im folgenden Abschitt werden einige interaktiven Beweisassistenten im Detail vorgestellt.

2.2.1 ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) [21] wird von Matt Kaufmann und J Strother Moore an der Universität von Texas implementiert. Es ist sowohl eine Programmiersprache, in der man Computersysteme modellieren kann, als auch Werkzeuge, die die Eigenschaften dieser Modelle prüfen können. ACL2 unterstützt Booleans, Integers, Records, Arrays, Strings und viele mehr. Der Theorembeweisassistent ACL2 erlaubt die Definition abstrakter Funktionen, die durch Axiome beschrieben werden. Theoreme über diese Funktionen können dann durch so genannte funktionale Instanzierung auf andere Funktionen übertragen werden, die die Axiome erfüllen. Programme werden in ACL2 [22] in einer Teilsprache von Common LISP definiert. Die Interaktion basiert auf so genannten Events, die im Commandline Interface eingegeben werden. Das sind Eingaben an den Beweisassistenten, die dessen internen Zustand verändern. Das können beispielsweise Definitionen von Funktionen oder Lemmata sein. Events sind ebenfalls in LISP Schreibweise notiert.

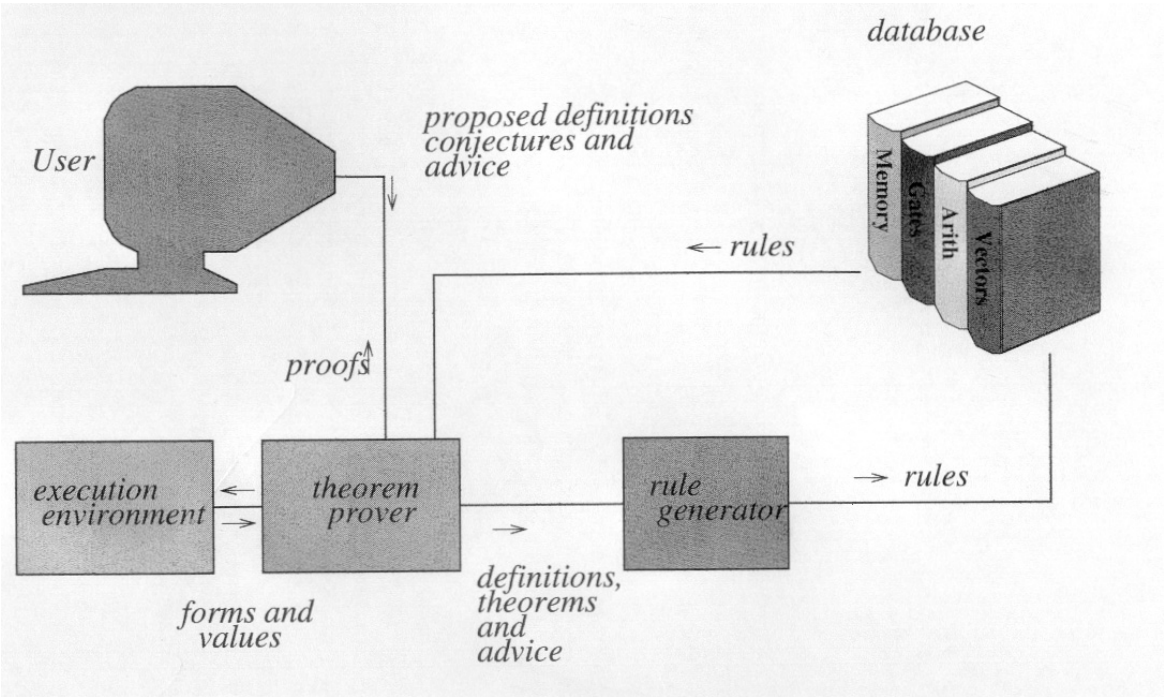

Abbildung 2.6: ACL2 System [21]

2.2.2 coq

Das coq [10] Werkzeug wurde Entwickelt im „LogiCal“ Projekt [33], es handelt sich dabei um ein formelles Beweisverwaltungssystem. Ein mit coq durchgeführter Beweis wird durch den PC automatisch überprüft. Dieses Beweissystem kann zur Entwicklung von mathematischen Theorien und formal zertifizierten Programmen eingesetzt werden und erfolgt in der Sprache Gallina. In ihr wird sowohl die Spezifikation als auch der Beweis formuliert. Es erlaubt die Definition von Funktionen oder Prädikaten, Zustände von mathematischen Lehrsätzen und Softwarespezifikationen, Interaktiv formelle Beweise dieser Lehrsätze durch einen kleinen „Zertifikaten Kern“ zu überprüfen. Coq umfasst einen Mechanismus für die automatische Generation von zertifizierten Beweisprogrammen, von ihren Spezifikationen, ein Dokumentationswerkzeug (coqdoc), ein Makefile Generationswerkzeug (coq_makefile und coqdep), einen Preprozessor für TeX Dateien (coqtex). Coq wird in der objektorientierten Sprache Caml geschrieben und verwendet den Camlp4 „Pre-processor-pretty-printer“ für Caml. Die aktuelle Version von Coq ist die 8.0.

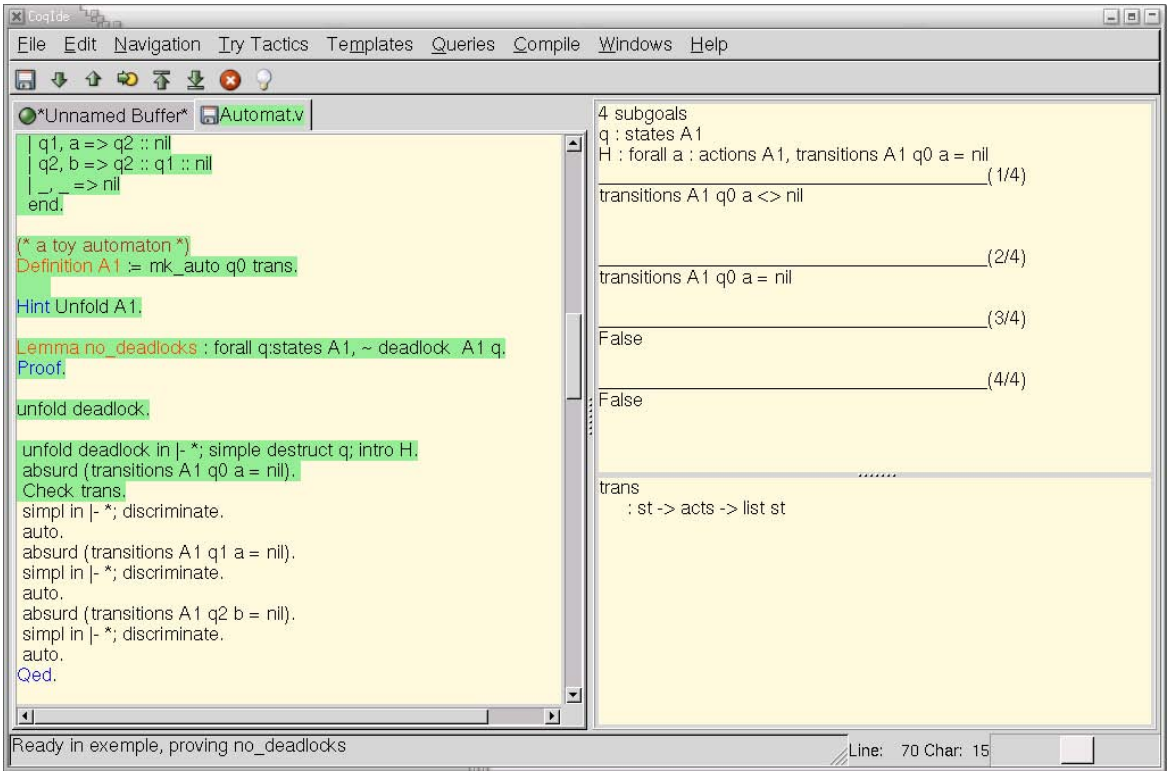

Abbildung 2.7: coq Interface [10]

2.2.3 HOL 4

HOL 4 [20] ist eine Weiterentwicklung auf der Basis von HOL98 das ursprünglich an der Cambridge Universität implementiert wurde. Das HOL System ist eine Umgebung für einen interaktiven Lehrsatzbeweisassistent, der die Logik der höheren Ordnung [2] prüft. Mit den Erweiterungen, dass sich Variablen über Funktionen und Prädikate erstrecken können, die Logik getyped ist und es keine getrennte syntaktische Kategorie von Formeln gibt. Terme können Variablen, Konstante, Funktionen und λ-Abstraktionen enthalten. Die Theorie in HOL 4 unterstützt Boolean, Integers, Reals, arithmetische Ausdrücke, Listen, Sets, Multisets, Rekords, selbst definierte Typen und viele mehr. Seine Besonderheit ist ein hoher Grad der Programmierbarkeit durch die Metasprache ML die ähnlich LISP ist. HOL 4 besitzt nur ein Commandlinie Interface, daher ist der Einstieg in dieses System etwas schwieriger.

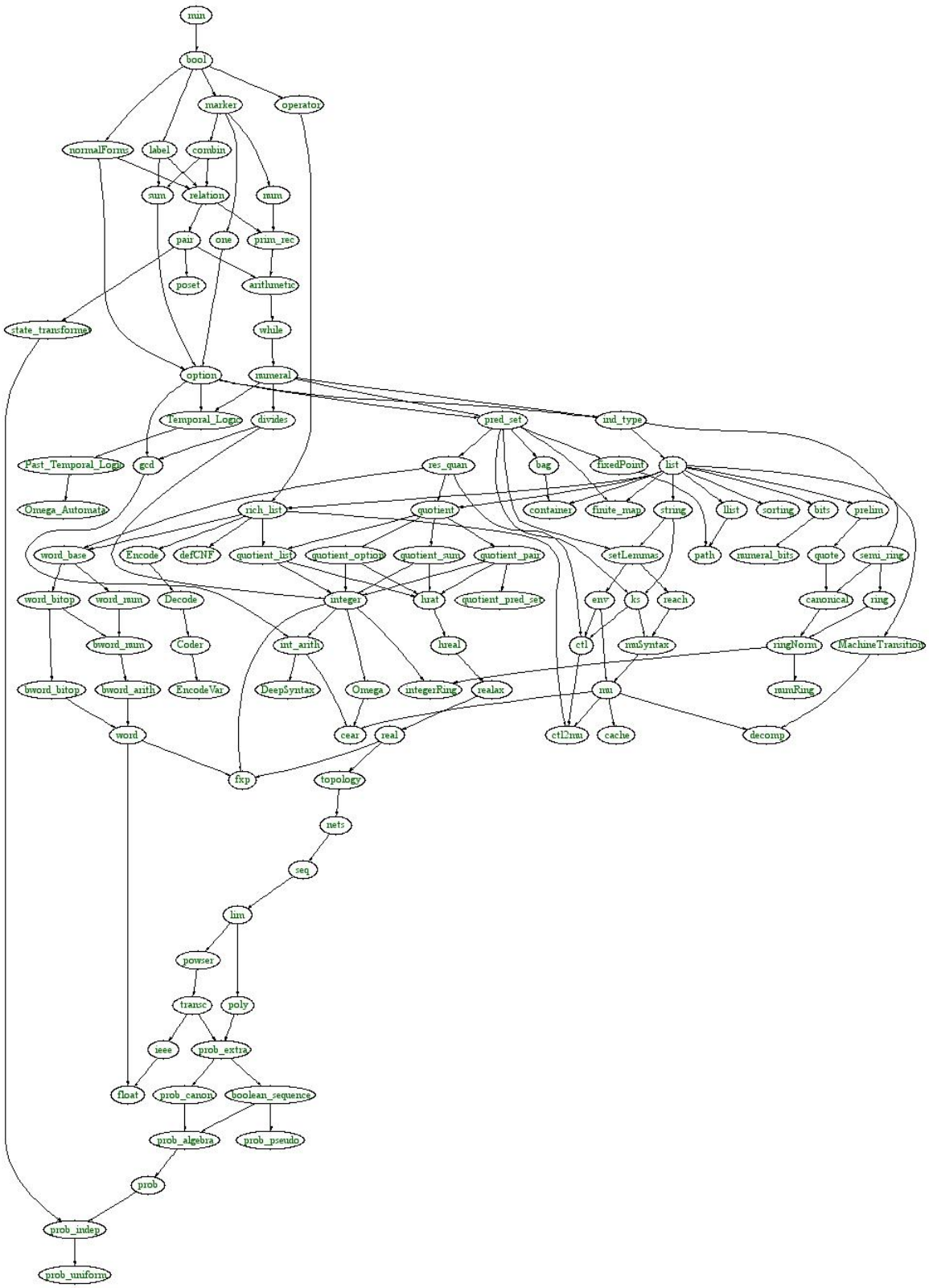

Abbildung 2.8: Theorien in HOL4 [20]

2.2.4 Isabelle/HOL

Isabelle [32] ist ein Kooperationsprojekt zwischen Lawrence C. Paulson (Universität von Cambridge) und Tobias Nipkow (Technische Universität von München). Es ist ein generischer Beweisassistent [51]. Isabelle ist sowohl für Beweise in mathematischen Theorien als auch zur Spezifikation und Verifikation der Korrektheit von Hard- und Softwaresystemen geeignet. Es basiert auf der Programmiersprache ML. Isabelle stellt Booleans, Integers, Listen, Tuple und rekursive Datentypen zur Verfügung. Selbstdefinierte Typen können auch verwendet werden. Es stellt eine eigene arithmetische Entscheidungsprozedur zur Verfügung. Beweise können in einer strukturierten Notation beruhend auf den traditionellen Beweisstil, oder „straightforwardly“ als Sequenz von Befehlen geschrieben werden. Definitionen und Beweise können TeX Quellen umfassen, von denen Isabelle automatisch typeset Dokumente erzeugen kann.

Isabelle vereinigt einige Werkzeuge, um die Produktivität des Benutzers zu verbessern, indem sie einige Teile des Beweisverfahrens automatisiert. Insbesondere durch das klassische logische Schlussfolgern kann Isabelle mit schrittweisem Lösen langer Ketten die Formeln prüfen. Der Simplifier kann mit und über Gleichungen vernünftig urteilen. Lineare arithmetische Ausdrücke werden automatisch bewiesen. Isabelle kann einerseits als ein generisches Framework für das rapid prototyping von deduktiven Systemen dienen. Andererseits stellt die vorhandene Hauptlogik von Isabelle/HOL eine Lehrsatzprüfungsumgebung zur Verfügung. Die Logik ist nicht fest in Isabelle verdrahtet, aber formuliert in Isabells Meta Logik. Es gibt für Isabelle viele syntaktische und deduktive Werkzeuge. Diese können neue Logiken wie folgt definieren:

- Deklaration von konkreter Syntax
- Deklaration von abstrakter Syntax
- Deklaration von deduktiven Regeln
- Instantiation von automatischen generischen Prüfprozeduren
- Codieren von manuellen speziellen Beweisverfahren

Isabelle Proof General: Fib.thy

File Edit View Cmds Tools Options Buffers Proof-General X-Symbol Isabelle Help

State Context Retract Undo Next Use Goto Find Command Stop Restart Info Help

Fib.thy *isabelle*

```
text{*We disable @{text fib.Suc_Suc} for simplification ...*}
declare fib.Suc_Suc [simp del]

text{*...then prove a version that has a more restrictive pattern.*}
lemma fib_Suc3: "fib (Suc (Suc (Suc n))) = fib (Suc n) + fib (Suc (Suc n))"
  by (rule fib.Suc_Suc)


text {* \medskip Concrete Mathematics, page 280 *}

lemma fib_add: "fib (Suc (n + k)) = fib (Suc k) * fib (Suc n) + fib k * fib n"
  apply (induct n rule: fib.induct)
    prefer 3
    txt {* simplify the LHS just enough to apply the induction hypotheses *}
    apply (simp add: fib_Suc3)
    apply (simp_all add: fib.Suc_Suc add_mult_distrib2)
    done

lemma fib_Suc_neq_0: "fib (Suc n) ≠ 0"
  apply (induct n rule: fib.induct)
    apply (simp_all add: fib.Suc_Suc)
  done

lemma fib_Suc_gr_0: "0 < fib (Suc n)"
  by (insert fib_Suc_neq_0 [of n], simp)
```

-----XEmacs: Fib.thy (Isar script XS:isabelle/s PenDel Font Abbrev; Scriptin

```
proof (prove): step 4

fixed variables: n, k

goal (lemma (fib_add), 3 subgoals):
 1. ⋀x. ⟦fib (Suc (Suc (x + k))) =
         fib (Suc k) * fib (Suc (Suc x)) + fib k * fib (Suc x);
         fib (Suc (x + k)) = fib (Suc k) * fib (Suc x) + fib k * fib x⟧
        ⟹ fib (Suc k) * fib (Suc x) + fib k * fib x +
           (fib (Suc k) * fib (Suc (Suc x)) + fib k * fib (Suc x)) =
           fib (Suc k) * (fib (Suc x) + fib (Suc (Suc x))) +
           fib k * fib (Suc (Suc x))
 2. fib (Suc (0 + k)) = fib (Suc k) * fib (Suc 0) + fib k * fib 0
 3. fib (Suc (Suc 0 + k)) =
    fib (Suc k) * fib (Suc (Suc 0)) + fib k * fib (Suc 0)
```

Abbildung 2.9: Isabelle Umgebung [32]

2.2.5 PVS (Spezifikations und Verifikations System)

PVS [31] wird am SRI International entwickelt. Es ist ein interaktiver Beweisassistent für Logik höherer Ordnung (HOL). Die Spezifikationssprache von PVS kann man sich als funktionale Programmiersprache, erweitert um Existenz- und Allquantoren sowie die Möglichkeit, Axiome und Theoreme zu formulieren vorstellen. Die Grundtypen umfassen uninterpretierte Typen die vom Benutzter definiert werden und „built-in“ Typen wie Booleans, Integers, Reals, und die Ordnungszahlen bis zu epsilon_0. Die Typekonstrukte umfassen Funktionen, Mengen, Tuple, Records, Enumerationen und rekursiv definierte abstrakte Datentypen, wie Listen und binäre Bäume. Es können Prädikatsubtypen [41] und abhängige Typen verwendet werden, um beschränkte Typen wie Primzahlen einzuführen. Diese Typen können Abhängigkeiten während des „typecheckings“ für den Beweis übernehmen, aber sie vergrößern leider die Spezifikation ungemein. In der Praxis werden die meisten Abhängigkeiten automatisch durch den Lehrsatzbeweisassistent abgearbeitet. PVS Spezifikationen werden in parametrisierte Theorien organisiert die Annahmen, Definitionen, Axiome und Lehrsätze enthalten können. Definitionen garantieren, dass Erweiterungen zur Verfügung gestellt werden. Dies wird durch Erzeugen von rekursiven abhängigen Funktionsdefinitionen verwirklicht. Induktiv definierte Beziehungen werden auch unterstützt. PVS Ausdrücke stellen die gewöhnlichen arithmetischen und logischen Operatoren, Funktionsanwendung, Lambda Abstraktionen und Quantoren innerhalb einer natürlichen Syntax zur Verfügung. Namen können einschließlich derjenigen der „built-in“ Operatoren wie UND und + frei überladen werden. Tabellarische Spezifikationen werden, durch automatisiertes Prüfen von Disjunktionen und „coverage“ abgedeckten Zuständen unterstützt. Der PVS Beweisassistent stellt eine Sammlung von Entscheidungsverfahren bereit, die interaktiv vom Benutzer innerhalb einer Sequenz von berechenbaren „Frameworks“ angewandt werden können [3]. Die primitiven Schlussfolgerungen umfassen Satz- und Quantorenregeln, Induktionen, „rewriting“, und Entscheidungsverfahren für lineare Arithmetik. Die Erfüllungen dieser primitiven Schlussfolgerungen werden für große Beweise optimiert. PVS umfasst ein auf BDD beruhendes Entscheidungsverfahren für das „Relationen μ-Kalkül“ und stellt dadurch eine experimentelle Integration zwischen dem Lehrsatz Prüfer und dem CTL Modellprüfer zur Verfügung. Die PVS Benutzeroberfläche verwendet Gnu oder X Emacs, welche eine einheitliche Schnittstelle in der Spezifikationssprache

und des Beweisassistenten zur Verfügung stellt. Befehle können entweder durch Pulldown-Menüs oder durch erweiterte Befehle von Emacs ausgewählt werden.

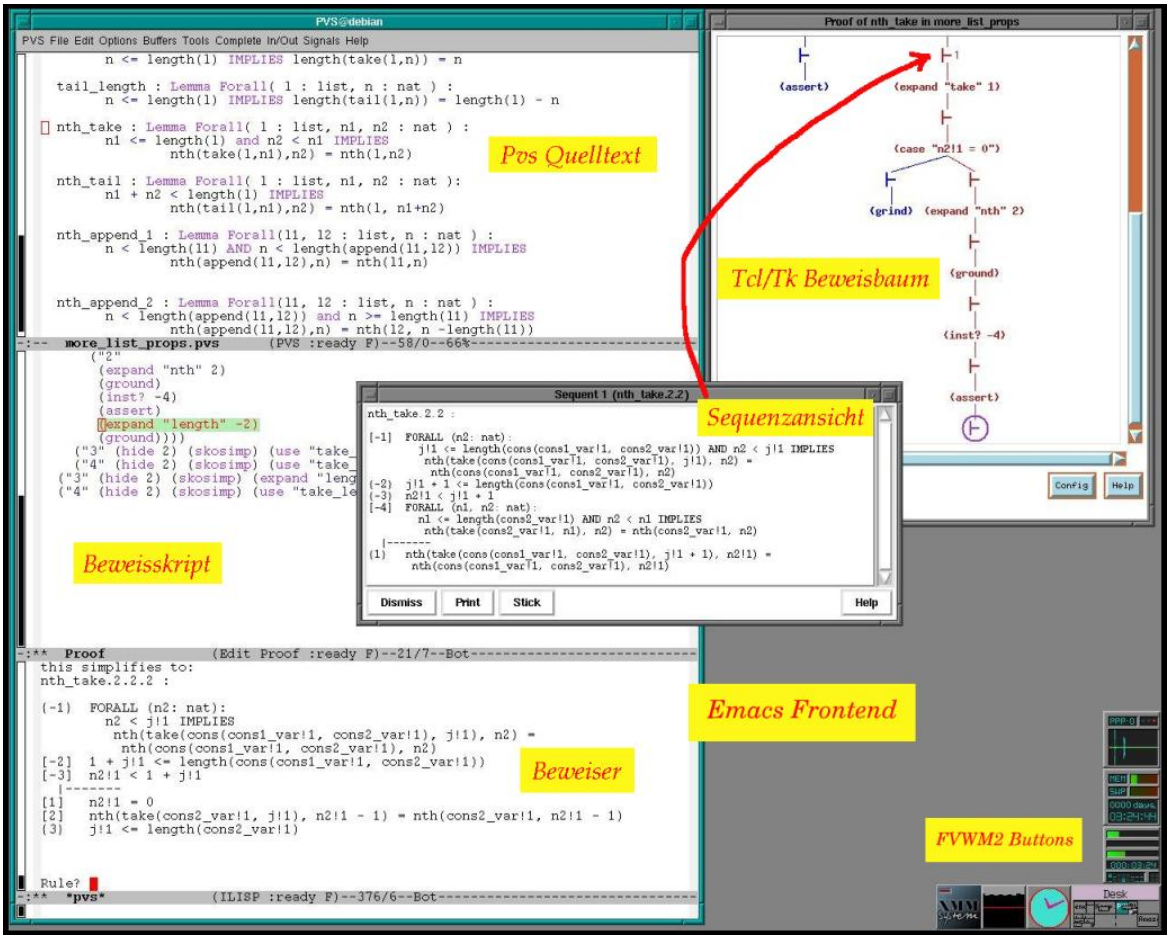

Abbildung 2.10: Die PVS Umgebung [31]

2.2.6 Zusammenfassung

In der vorangegangen Punkten habe ich PVS, Isabelle/HOL, HOL, ACL2 und coq näher vorgestellt. Folgende Tabelle fast die zuvor erwähnten interaktiven Beweisassistenten zusammenfassen.

Beweisassistent	**Sprache**	**Entscheidungs-verfahren**	**Referenz**
PVS	funktionale Programmiersprache	YICES	[31]
Isabelle/HOL	ML ähnlich LISP	eigenes	[32]
HOL	ML ähnlich LISP	SATO, GRASP, ZCHAFF, MINISAT	[20]
ACL2	Teilsprache von Common LISP	eigenes	[21]
coq	Gallina	Omega	[10]

Tabelle 2.2: Zusammenfassung Beweisassistent

2.3 Integration von Entscheidungsprozeduren in Beweisassistenten

Die wichtigsten mathematischen Beweisassistenten, die auf klassischer Logik höherer Ordnung, aufbauen sind HOL, ISABELLE/HOL, und PVS. Die Integration verschiedener Spezialsysteme in ein heterogenes Gesamtsystem hat seit Mitte der 90er Jahre stark an Bedeutung gewonnen. Das EU Netzwerk CALCULEMUS [27] wurde mit dem Ziel gegründet, bessere mathematische Assistenzsysteme durch die Integration von symbolischem Schließen (Deduktion) und symbolischer Berechnung (Computer Algebra) zu entwickeln. Die möglichen Verfahren sind:

1. Kooperation von Deduktionssystem und Computeralgebrasystem auf gleicher Ebene
2. Einbettung eines Deduktionssystems in ein Computeralgebrasystem, bzw. umgekehrt
3. die Einbettung eines Computeralgebrasystems in ein Deduktionssystem.

Geht man von einem Beweissystem als Basissystem wie in Punkt 3 aus, gibt es grundsätzlich zwei verschiedene Möglichkeiten zur Integration:

1. (Integration auf Kalkülebene) Direkte domänenspezifische Erweiterung des Kalküls eines Beweissystem durch Spezialregeln; d.h. durch den Einbau einer Theorie. Diese Methode wird vornehmlich für Systeme erster Ordnung angewendet und Beispiele sind die Erweiterung des Superpositionsansatzes durch den Einbau einer Shostak Theorie [19], der Theorem Proving Modulo Ansatz [16], oder die historische T-Resolution und die T-Unifikation [5]
2. (Integration auf Systemebene) Die Alternative ist eine Integration domänenabhängiger Spezialsysteme. Ein Beispiel ist die am SRI entwickelte Verifikationsumgebung PVS [31], die viele domänenspezifische Entscheidungsprozeduren integriert. Vorteile im Vergleich zu Punkt 1 liegen vor allem in der Flexibilität des Ansatzes. Eine zusätzliche Herausforderung ist die Korrektheit der Integration, die z.B. durch eine Transformation der externen Repräsentation in den Repräsentationsformalismus des Basissystems erreicht werden kann, um dort eine Verifikation durchzuführen. Ein Beispiel ist das Elan System das solche Transformationen in coq unterstützt [29].

Der vom Wolfgang Schreiner entwickelte ProofNavigator [38] ist ein interaktiver Beweisassistent. Es wird in Java implementiert und verwendet als Entscheidungsprozedur CVCL 2.0. Um den Beweisassistenten noch effizienter zu gestalten, sollen einige der oben beschriebenen Entscheidungsverfahren modular in den interaktiven Beweisassistent implementiert werden. Um dies zu verwirklichen, muss die Logik der höheren Ordung, die der ProofNavigator versteht, in die Logik der ersten Ordung, die die einzelen Entscheidungsverfahren verstehen, geparst werden. Die formale Beschreibung diese Problems folgt im nächsten Kapitel.

3 Formale Problembeschreibung

Prädikatenlogik erster Stufe, PL1 oder auch FOL nach „first order logic“, ist eine Erweiterung der Aussagenlogik. Die konzeptuelle Grundlage ist die Annahme, dass die Welt aus Objekten besteht, die bestimmte Eigenschaften besitzen. Objekte können miteinander über Relationen in Beziehung stehen. Diese werden mittels Prädikat- und Funktionssymbolen ausgedrückt, die über die ihnen zugeordnete Interpretation ihre Bedeutung bekommen. Mit Hilfe von Quantoren können Eigenschaften ganzer Objektmengen beschrieben werden. Weitere logische Konnektoren $(\neg, \vee, \wedge, \rightarrow)$ dienen (ähnlich zur Aussagenlogik) zur induktiven Konstruktion von komplexen Formeln. Die Semantik der Symbole ist durch die Sprachdefinition genau festgelegt.
Die meisten Strategien zum Schlussfolgern können jedoch im Allgemeinen nur mit exponentieller Komplexität realisiert werden. So können Theorembeweiser schon bei kleinen trickreichen Problemen versagen, die nur wenige Formeln umfassen. Zudem ist die Prädikatenlogik erster Stufe in seiner klassischen Form nur semientscheidbar, d.h. falls ein Theorem nicht bewiesen werden kann, stoppt der Algorithmus nicht. Des weiteren darf die Wissensbasis keine Widersprüche enthalten, da sonst prinzipiell jede Formel aus der Wissensbasis abgeleitet werden kann.
Die Prädikatenlogik zweiter Ordnung, oder oft auch „Higher Order Logic“(HOL) genannt, erweitert die „First Order Logic“ um einige Punkte: Zum einen können Quantoren nicht mehr nur über Variablen, sondern auch über Prädikate gelegt werden. Andererseits können Prädikate in HOL andere Prädikate als Argument erhalten. Ein „Higher Order Prädikat“ der Ordnung n hat also ein oder mehr Prädikate der Ordnung $n - 1$ als Argument (wobei $n > 1$).

3.1 Logik der ersten Ordnung

Die Basis dieses Kapitels bildet die Syntax und Semantik der Logik der ersten Ordnung oder Prädikatenlogik der ersten Stufe (PL1), um Erfüllbarkeit, Gültigkeit und Allgemeingültigkeit zu definieren. Die Definitionen werden zunächst für die PL1 formuliert, und dann im Laufe diese Kapitels auf getypte PL1 erweitert. Dann wird die PL1 auf Logik höhere Ordnung (HOL) übergeführt, um die Gültigkeit von Formeln von PL1 auf HOL zu erhalten.

3.1.1 Basis Definitionen

Dieser Abschnitt definiert die einzelnen Symbole und Sonderzeichen für die Syntax der PL1.

Bezeichnungskonventionen

(,) Klammern
w Wahr
f Falsch
$\neg$ Negation
$\wedge$ und
$\vee$ oder
$\rightarrow$ Implikation
$\leftrightarrow$ Äquivalenz
$\forall$ für alle
$\exists$ es existiert

$V = \{v_0, v_1, v_2, ...v_i, ...\}$ heißt Menge der Variablen (Inidividuenvariablen)
$K = \{k_0, k_1, k_2, ...k_i, ...\}$ heißt Menge der Konstantensymbole
$F = \{f_0, f_1, f_2, ...f_i, ...\}$ heißt Menge der Funktionssymbole
$P = \{p_0, p_1, p_2, ...p_i, ...\}$ heißt Menge der Prädikatensymbole

3.1.2 Syntax von PL1

Definition (Signatur)

Eine Signatur S bezeichnet das Tripel (α, β, γ) mit den Eigenschaften

- α, β sind höchstens abzählbare unendliche Mengen
- $\alpha \cap \beta = \{\,\}$
- $\gamma : \alpha \cup \beta \rightarrow \mathbb{N}$

Ist $F \in \alpha$ und $\gamma(F) = n$, so heißt F n-stelliges Funktionsymbol.
Im Falle n = 0 spricht man auch von Konstanten.
Ist $P \in \beta$ und $\gamma(P) = n$, so heißt P n-stelliges Prädikatensymbol.
Im Falle n=0 spricht man auch von Aussagenvariablen.

Bemerkung Die Funktion γ gibt also die Stelligkeit von Funktoren und Prädikaten an, während α und β neben den Variablen die Grundsymbole der Sprache stellen.

Definition (Terme)

Ist S eine Signatur, so ist T die Menge der Terme mit den Eigenschaften:

- $V \subseteq T$
- $k \in T$ für alle Konstanten $k \in \alpha$
- Wenn $F \in \alpha, \gamma(F) = n \geq 1, T_1, ..., T_n \in T$ so gilt $F(T_1, ..., T_n) \in T$

Beispiele für Terme

Ausdruck	**Operator**	**Typ**	**Präfix-Syntax**
$v_i + v_j$	$+$	Infix	$Plus[v_i, v_j]$
$v_i - v_j$	$-$	Infix	$Minus[v_i, v_j]$
$-v_i$	$-$	Präfix	$Minus[v_i]$
$v_i!$	$!$	Postfix	$Factorial[v_i]$
$\frac{v_i}{v_j}$	$\frac{\circ}{\circ}$	Infix	$Divide[v_i, v_j]$
$(v_i)_n$	$\circ_n$	Subscript	$Subscript[v_i, n]$
$(v_i)^n$	$\circ^n$	Superscript	$Power[v_i, n]$

Tabelle 3.1: Terme

Definition (Atome)

Ist S eine Signatur so ist A die Menge der atomaren Formeln mit den Eigenschaften

- Wenn $P \in \beta, \gamma(P) = 0$ so gilt $P \in A$
- Wenn $P \in \beta, \gamma(P) = n \geq 1, T_1, ..., T_n \in T$ so gilt $P(T_1, ..., T_n) \in A$

Beispiele für Atome

Ausdruck	**Operator**	**Typ**	**Präfix-Syntax**
$p_i = p_j$	$=$	Infix	$Equal[p_i, p_j]$
$p_i \leq p_j$	$\leq$	Infix	$LessEqual[p_i, p_j]$
$p_i \geq p_j$	$\geq$	Infix	$GreaterEqual[p_i, p_j]$
$p_i \in A$	$\in$	Infix	$Element[p_i, A]$

Tabelle 3.2: Atome

Definition (Formeln)

Fo ist die Menge der Formeln über der Signatur S wenn gilt

- $A \subseteq Fo$
- Wenn $p_i, p_j \in Fo$ und $v_i \in V$ dann gilt

  1. $\neg p_i \in Fo$ (Negation)
  2. $p_i \wedge p_j \in Fo$ (Konjunktion)
  3. $p_i \vee p_j \in Fo$ (Disjunktion)
  4. $p_i \rightarrow p_j \in Fo$ (Implikation)
  5. $p_i \leftrightarrow p_j \in Fo$ (Äquivalenz)
  6. $\forall v_i.p_i \in Fo$ (Generalisation)
  7. $\exists v_i.p_i \in Fo$ (Partikulation)

wobei $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ als Junktoren und $\forall, \exists$ als Quantoren bezeichnet werden. Die Hierarchie bei den Junktoren und Quantoren wird wie folgt definiert.
$\neg \lhd \wedge \lhd \vee \lhd \rightarrow \lhd \leftrightarrow \lhd \forall \lhd \exists$.
Dadurch können Klammern in Formeln weggelassen werden.

Notation (λ-Kalkül)

- Das Lambda-Kalkül wird als geringfügige notationelle Erweiterung der Logik erster Stufe aufgefasst, die es uns gestattet, Variablen zu binden, indem wir den Operator λ benutzen. Durch λ gebundene Variablen sind „Platzhalter“ für fehlende Information.
- Steht eine Lambda-Ausdruck vor einem anderen Ausdruck (den Argumentausdruck), dann ist das eine Anweisung, den Argumentausdruck in die Platzhalter einzusetzen. Das wird Funktionalapplikation genannt.

Definition (λ-Operator)

Der Lambda-Operator kennzeichnet fehlende Information, indem er Variablen bindet.

- Es folgt ein einfacher Lambda-Ausdruck: $\lambda v_i.P(v_i)$
- Der Präfix λv_i bindet das Auftreten des v_i in $P(v_i)$.
- $\lambda v_i.P(v_i)$ kann wie folgt gelesen werden:
  Das einstellige Prädikat P, sucht nach einem Term, um seine Argumentstelle zu füllen.

3.1.3 Semantik von PL1

Im vorigen Abschnitt wurde erklärt, wie man aus gewissen Grundsymbolen größere syntaktische Einheiten wie Terme, Formeln oder Sequenzen bildet. Was man sich unter diesen Zeichenreihen vorzustellen hat, das heißt wie man die Erfüllbarkeit und Gültigkeit einer Formel definiert wird in diesem Punkt beschrieben.

Definition (Interpretation(D,I))

Sei S eine Signatur, dann ist eine Interpretation ein Paar (D, I) mit den folgenden Eigenschaften.

- $D \neq \{\,\}$ (D heist domain, „universe of discourse“ von (D,I))
- 1. I ordnet jeder Konstanten $k \in \alpha$ ein Element $I(k) \in D$ zu.
  2. I ordnet jedem n-stelligen Funktionssymbol $(n \geq 1)F \in \alpha$ eine Funktion $I(F) : D^n \to D$ zu.
  3. I ordnet jedem 0-stelligen Prädikatensymbol P einen der Wahrheitswerte w oder f zu.
  4. I ordnet jedem n-stelligen Prädikatensymbol P, für $n \geq 1$ eine n-stellige Relation über D zu, es ist also $I(P) \subseteq D^n$.

Definition (Termauswertung ($[[T]]_{(D,I,\epsilon)} \in D$))

Sei S eine Signatur und (D, I) eine Interpretation. Jede Funktion $\epsilon : V \rightarrow D$ heißt dann eine Variablenbelegung.
Dazu definiert man eine Termauswertung $[[T]]_{(D,I,\epsilon)} : T \rightarrow D$ wie folgt.

- Ist $T \in V$ so gilt $[[T]]_{(D,I,\epsilon)} := \epsilon(T)$.
- Ist $T \in \alpha$ mit $\gamma(T) = 0$, dann ist $[[T]]_{(D,I,\epsilon)} := I(T)$.
- Ist $F \in \alpha$ mit $\gamma(T) = n, n \geq 1$ und seien $T_1, ..., T_n \in \beta$, dann ist $[[F(T_1, ..., T_n)]]_{(D,I,\epsilon)} := I(F)([[T_1]]_{(D,I,\epsilon)}, ..., [[T_n]]_{(D,I,\epsilon)})$

Term	**Bedeutung**
Konstanten	Die Interpretation der Konstanten
Variablen	Das durch die Belegung der Variablen zugeordnete Objekt
$\epsilon_I(f(t_1, ..., t_n))$	Die Interpretation von ϵ angewendet auf die Interpretationen der t_i

Tabelle 3.3: Terme Sematik

Definition (Atomauswertung)

Sei S eine Signatur und (D,I) eine Interpretation, dann ist die Atomauswertung definiert als.

- $[[P(T_1, ..., T_n)]]_{(D,I,\epsilon)} := \begin{cases} w & : ([[T_1]]_{(D,I,\epsilon)}, ..., [[T_n]]_{(D,I,\epsilon)}) \in I(P) \\ f & : \text{sonst} \end{cases}$

Definition (Formelauswertung ($[[Fo]]_{(D,I,\epsilon)} \in \{w, f\}$))

Sei S eine Signatur, (D, I) eine Interpretation und ϵ eine Variablenbelegung, dann wird die Formelauswertung $[[Fo]]_{(D,I,\epsilon)} : F \to w, f$ definiert als.

- $[[\neg p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = f \\ f & : [[p_i]]_{(D,I,\epsilon)} = w \end{cases}$
- $[[p_i \wedge p_j]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} = w \\ f & : \text{sonst} \end{cases}$
- $[[p_i \vee p_j]]_{(D,I,\epsilon)} := \begin{cases} f & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} = f \\ w & : \text{sonst} \end{cases}$
- $[[p_i \to p_j]]_{(D,I,\epsilon)} := \begin{cases} f & : [[p_i]]_{(D,I,\epsilon)} = w \text{ und } [[p_j]]_{(D,I,\epsilon)} = f \\ w & : \text{sonst} \end{cases}$
- $[[p_i \leftrightarrow p_j]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} \\ f & : \text{sonst} \end{cases}$
- $[[\forall v_i . p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : \text{für alle } d \in D \text{ gilt} : [[p_i]]_{(D,I,\epsilon[v_i \to d]} = w \\ f & : \text{sonst} \end{cases}$
- $[[\exists v_i . p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : \text{für mindestens ein } d \in D \text{ gilt} : [[p_i]]_{(D,I,\epsilon[v_i \to d]} = w \\ f & : \text{sonst} \end{cases}$

Formel	Bedeutung
$\neg p_i$	Interpretiere p_i und verwende Wahrheitswerte für $\neg$
$p_i \vee p_j$	Interpretiere p_i und p_j und verwende Wahrheitswerte für $\vee$
$p_i \wedge p_j$	Interpretiere p_i und p_j und verwende Wahrheitswerte für $\wedge$
$p_i \rightarrow p_j$	Interpretiere p_i und p_j und verwende Wahrheitswerte für $\rightarrow$
$p_i \leftrightarrow p_j$	Interpretiere p_i und p_j und verwende Wahrheitswerte für $\leftrightarrow$

Tabelle 3.4: Formeln

Definition (Gebundene und freie Variablen)

Ein Vorkommen der Variable v_i ist *gebunden* in der Formel Fo, falls v_i in einer Teilformel von Fo vorkommt, die die Gestallt $\exists v_i.P(p_i)$ oder $\forall v_i.P(p_i)$ hat. In diesem Fall wird v_i durch den Existenzquantor ($\exists$) bzw. den Allquantor ($\forall$) *gebunden*.
Andernfalls ist das Vorkommen von v_i *frei* in Fo.
Eine Formel ohne freies Vorkommen von Variablen heißt *geschlossen*. Geschlossene Formeln werden auch als Aussagen bezeichnet.

Beispiele für gebundene und freie Variablen

Formel	nur gebunden	nur frei
$\exists v_i.P(v_i, F(v_i, v_j))$	v_i	v_j
$\neg\exists v_j \forall v_i.P(v_j, F(v_i, v_k))$	v_i, v_j	v_k
$\exists v_i.P(v_i, F(v_i, v_j)) \vee \neg\exists v_j \forall v_i.Q(v_j, G(v_i, v_j))$	v_i	v_j

Tabelle 3.5: Gebundene und Freie Variablen

Definition (Modell, Erfüllbarkeit, Allgemeingültigkeit)

Sei S eine Signatur und (D, I) eine zugehörige Interpretation. Gilt $\omega_{D,I}(p_i) = w$ für eine Formel $p_i \in F$, so heißt (D, I) ein *Modell* von p_i.
Ist $M \subseteq Fo$ und für alle $p_i \in M$ gilt $\omega_{D,I}(p_i) = w$, so heißt (D, I) ein *Modell* von M.
Existiert für eine Formel aus Fo ein *Modell*, so nennt man die Formel auch *erfüllbar*, andernfalls *unerfüllbar*.
Ist andererseits jede beliebige Interpretation (D, I) ein *Modell* einer Formel aus Fo, so nennt man die Formel *allgemeingültig*.

Beispiele für Erfüllbarkeit, Unerfüllbarkeit und Gültigkeit

Erfüllbar: $M = \forall v_i.P(p_i)$ ist erfüllbar wegen:

- $\forall v_i.P(p_i) \leftrightarrow P(p_i)$
  v_i kommt in $P(p_i)$ nicht vor
- p_i kann jeden Wert aus dem Definitionsbereich annehmen
- Es gibt immer ein p_i, sodas $P(p_i)$ wahr wird

Gültig: $M = \exists v_i(\neg P(v_i) \vee P(p_i))$ ist gültig wegen:

- $\exists v_i(\neg P(v_i) \vee P(p_i)) \leftrightarrow \exists v_i(\neg P(v_i)) \vee P(p_i)$
  v_i kommt in $P(p_i)$ nicht vor
- wenn $P(p_i)$ wahr ist, ist M wahr
- wenn $P(p_i)$ falsch ist, dann gibt es ein $v_i(v_i = p_i)$, sodass $\neg P(v_i)$ und M wahr sind

Unerfüllbar: $M = \forall v_i.P(p_i) \vee \neg\exists v_j.P(j)$ ist unerfüllbar wegen:

- wenn $\forall v_i.P(v_i)$ wahr ist, dann ist P für alle Objekte aus dem Definitionsbereichs wahr. Dann ist $\neg\exists v_j.P(v_j)$ falsch. (Die Aussage, es gäbe kein Objekt aus dem Definitionsbereich, für das P gilt, ist falsch).
- wenn $\forall v_i.P(v_i)$ wahr ist, dann ist P für alle Objekte aus dem Definitionsbereich falsch. Damit ist $\neg\exists v_j.P(v_j)$ wahr. (Es gibt tatsächlich kein Objekt aus dem Definitionsbereich, für das P gilt).

3.2 Getypte Prädikatenlogik der ersten Stufe

Die Erweiterung von PL1 um Typen, im Folgenden getypten Prädikatenlogik der ersten Stufe genannt, erweitert nicht die Ausdrucksfähigkeit von PL1. Erwartungsgemäß bleibt derselbe berechenbarkeitstheoretische Status erhalten. Die getypte Logik der 1.Stufe ist wie PL1 unentscheidbar, aber rekursiv aufzählbar.
Da in den einzelnen Teiltermen der getypten Prädikatenlogik der ersten Stufe durch ihre Typzugehörigkeit mehr Information vorhanden ist, hat man in getypter Prädikatenlogik der ersten Stufe meist den Vorteil, dass sich derselbe Sachverhalt in kompakteren Formeln beschreiben lässt. In getypten Prädikatenlogik der ersten Stufe besteht die Möglichkeit nur über eine ausgezeichnete Teilmenge des Diskursbereiches eine Aussage zu machen, für andere Individuen ist das entsprechende Prädikat dann einfach nicht definiert, da die entsprechende Aussage für diese Individuen sinnlos wäre. Hat man zum Beispiel den Diskursbereich aller natürlichen Zahlen, so gibt es viele Aussagen, welche nur gerade oder nur ungerade natürliche Zahlen betreffen.

3.2.1 Syntax

Datentypen

Die Typen sind mit ihren Operationen, Prädikaten und Funktionen, fix integrierte Datentypen.

Notation:

$So \in$ Sorten
$P \in$ Prädikate
$F \in$ Funktionen
$Ty \in$ Typ (Sorten)
$D \in$ Domain
$V \in$ Variablen
$K \in$ Konstanten

$Ty ::= So_1|So_2|...|So_n.$

Aus der Notation der Sorten, Prädikaten und Funktionen kann jetzt die Signatur und die Syntax der Terme und Formeln definiert werden.

Definition (Signatur)

Eine Signatur S bezeichnet das Quartupel $(So, \alpha, \beta, \gamma)$, wobei $So = \{So_1, ..., So_n\}$, $\alpha = \{F \in \text{Funktionen }\}$ und $\beta = \{P \in \text{Prädikate }\}$ ist. Jedes Element von β wird S-Formel genannt. Außerdem definieren wir die Stelligkeit für jedes Prädikat $\gamma_P : \beta \rightarrow \{Ty_1 x ... x Ty_n\}$, für jede Funktion $\gamma_F : \alpha \rightarrow \{Ty_1 x ... x Ty_n \rightarrow Ty\}$, für jedes $v \in$ Variablen mit $\gamma_v(v) = Ty_i$ für ein $Ty_i \in$ Typ und für jede $k \in$ Konstanten mit $\gamma_k(k) \in$ Konstanten, mit den Eigenschaften:

- α, β sind höchstens abzählbare unendliche Mengen
- $\alpha \cap \beta = \{\,\}$

Nun wird die Syntax von Termen, Atomen und Formeln der getypten PL1 definiert.

Syntax (Terme)

Ist S eine Signatur dann ist die Menge T der getypten Terme wie folgt definiert

- für jede Variable $v \in V$ mit $\gamma_V(v) = Ty$ gilt $(v : Ty) \in T$
- für jede Konstante $k \in K$ mit $\gamma_K(k) = Ty$ gilt $(k : Ty) \in T$
- für jede Funktion $F \in \alpha$ mit $\gamma_\alpha(F) = (Ty_1, ...Ty_n) \rightarrow Ty$ und $T_1 : Ty_1, ..., T_n : Ty_n \in T$ so gilt
  $F(T_1 : Ty_1, ..., T_n : Ty_n) \in T$

Syntax (Atome)

Ist S eine Signatur dann ist die Menge A der getypten atomaren Formeln wie folgt definiert

- Wenn $P \in \beta, \gamma(P) = ()$ so gilt $P \in A$
- Wenn $P \in \beta$ mit $\gamma_\beta(P) = (Ty_1 x ... x Ty_n)$ und $T_1 : Ty_1, ..., T_n : Ty_n \in T$ so gilt
  $P(T_1 : Ty_1, ..., T_n : Ty_n) \in A$

Syntax (Formeln)

Fo ist die Menge der Formeln über der *Sigantur* S wenn gilt

- $A \subseteq Fo$
- Wenn $F_1 \in Fo$, $F_2 \in Fo$ und $v_i \in V$ dann gilt

  1. Negation: $\neg F_1 \in Fo$
  2. Konjunktion: $F_1 \wedge F_2 \in Fo$
  3. Disjunktion: $F_1 \vee F_2 \in Fo$
  4. Implikation: $F_1 \rightarrow F_2 \in Fo$
  5. Äquivalenz: $F_1 \leftrightarrow F_2 \in Fo$
  6. Generalisation: $\forall v_i.F_1 \in Fo$
  7. Partikulation: $\exists v_i.F_1 \in Fo$

3.2.2 Semantik

Definition ((D,I) ist eine S-Interpretation)

Eine S-Interpretation ist ein Paar (D,I) sodass:

1. $I(So_i) \in D$, für alle $So_i \in Ty$
2. Wenn $Ty_1, ..., Ty_n, Ty \in D$, dann ist auch $Ty_1 x ... x Ty_n \in D$ und $Ty_1 x ... x Ty_n \rightarrow Ty \in D$
3. $K \in \alpha$ mit $\gamma_F(K) = Ty \Rightarrow I(K) \in I(Ty)$
4. $F \in \alpha$ mit $\gamma_F(F) = Ty_1 x ... x Ty_n \rightarrow Ty \Rightarrow I(F) \in I(Ty_1) x ... x I(Ty_n) \rightarrow I(Ty)$
5. $P \in \beta$ mit $\gamma_P(P) = () \Rightarrow I(P) \in \{w, f\}$
6. $P \in \beta$ mit $\gamma_P(P) = (Ty_1 x ... x Ty_n) \Rightarrow I(P) \subseteq I(Ty_1) x ... x I(Ty_n)$

Bemerkung:
D ist damit die Menge der Domains, in denen die Interpretation der Prädikate und Funktionen von S enthalten sind.

Definition (Termauswertung ($[[T]]_{(D,I,\epsilon)} \in D$)

Sei S eine Signatur und (D, I) eine Interpretation. Jede Funktion ϵ heißt dann eine Variablenbelegung, wenn für jedes $v \in V$ gilt $\epsilon(v) \in I(\gamma_V(v))$
Dazu definiert man eine Termauswertung $[[T : Ty]]_{(D,I,\epsilon)} : T \rightarrow D$ wie folgt.

- Ist $T \in V$ so gilt $[[T : Ty]]_{(D,I,\epsilon)} := \epsilon(T)$.
- Ist $T \in \alpha$ mit $\gamma(T : Ty) = ()$, dann ist $[[T : Ty]]_{(D,I,\epsilon)} := I(T)$.
- Ist $F \in \alpha$ mit $\gamma_F(F) = (Ty_1 x ... x Ty_n) \rightarrow T$ und seien $T_1, ..., T_n \in T$ sodass $I(T_1) \in I(Ty_1), ..., I(T_n) \in I(Ty_n)$, dann ist
  $[[F(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D,I,\epsilon)} := I(F)([[T_1 : Ty_1]]_{(D,I,\epsilon)}, ..., [[T_n : Ty_n]]_{(D,I,\epsilon)})$

Definition (Atomauswertung)

Sei S eine Signatur und (D, I) eine Interpretation, dann ist die Atomauswertung definiert als.

- $[[P()]]_{(D,I,\epsilon)} := \begin{cases} w & : () \in I(P) \\ f & : \text{sonst} \end{cases}$
- $[[P(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D,I,\epsilon)}$
  $:= \begin{cases} w & : ([[T_1 : Ty_1]]_{(D,I,\epsilon)}, ..., [[T_n : Ty_n]]_{(D,I,\epsilon)}) \in I(P) \\ f & : \text{sonst} \end{cases}$

Definition (Formelauswertung ($[[Fo]]_{(D,I,\epsilon)} \in \{w, f\}$))

Sei S eine Signatur, (D, I) eine Interpretation, ϵ eine Variablenbelegung und $p_i, p_j \in Fo$, dann wird die Formelauswertung $[[Fo]]_{(D,I,\epsilon)} : F \to \{w, f\}$ definiert als.

- $[[\neg p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = f \\ f & : [[p_i]]_{(D,I,\epsilon)} = w \end{cases}$
- $[[p_i \wedge p_j]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} = w \\ f & : \text{sonst} \end{cases}$
- $[[p_i \vee p_j]]_{(D,I,\epsilon)} := \begin{cases} f & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} = f \\ w & : \text{sonst} \end{cases}$
- $[[p_i \to p_j]]_{(D,I,\epsilon)} := \begin{cases} f & : [[p_i]]_{(D,I,\epsilon)} = w \text{ und } [[p_j]]_{(D,I,\epsilon)} = f \\ w & : \text{sonst} \end{cases}$
- $[[p_i \leftrightarrow p_j]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} \\ f & : \text{sonst} \end{cases}$
- $[[\forall v_i . p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : \text{für alle } d \in D \text{ gilt} : [[p_i]]_{(D,I,\epsilon[v_i \to d])} = w \\ f & : \text{sonst} \end{cases}$

- $[[\exists v_i.p_i]]_{(D,I,\epsilon)} := \begin{cases} w & \text{: für mindestens ein } d \in D \text{ gilt : } [[p_i]]_{(D,I,\epsilon[v_i \to d])} = w \\ f & \text{: sonst} \end{cases}$

3.2.3 Benutzerdefinierte Typen

Um umfangreichere Datentypen nicht auf einmal und vielleicht jedesmal neu spezifizieren zu müssen, sondern bulit-in Datentypen wie nat, bool und bereits spezifizierte Datentypen als Bausteine verwenden können, erweitert man vorhandene Typen um neue Typen, neue Operatoren, neue Prädikate und neue Funktionen.
Dazu definieren wir:

Definition ($S^{'}$ ist Erweiterung von S durch Se)

Sei $S = (So, \alpha, \beta, \gamma)$ eine Signatur und sei $S_e = (So_e, \alpha_e, \beta_e, \gamma_e)$ so dass
$(So \cup So_e, \alpha_e, \beta_e, \gamma_e)$ eine Signatur ist, dann ist
$S^{'} = (So^{'}, \alpha^{'}, \beta^{'}, \gamma^{'})$ eine *Erweiterung* von S durch S_e genau dann wenn gilt:
$So^{'} = So \cup So_e$,
$\alpha^{'} = \alpha \cup \alpha_e$,
$\beta^{'} = \beta \cup \beta_e$ und
$\gamma^{'} = \gamma \cup \gamma_e$.
Somit ist $S^{'} = (So^{'}, \alpha^{'}, \beta^{'}, \gamma^{'})$ auch eine Signatur.

Bemerkung:
S_e muß nicht zwingend auch eine Signatur sein.

Bemerkung:
S wird als *Einschränkung* von $S^{'}$ durch ein S_e bezeichnet.

Definition ($(D, I^{'})$ ist eine Erweiterung von (D, I) zu $S^{'}$)

Sei $S = (So, \alpha, \beta, \gamma)$ eine Signatur, (D, I) eine S - Interpretation und sei $S^{'} = (So^{'}, \alpha^{'}, \beta^{'}, \gamma^{'})$ eine *Erweiterung* von S. Dann heißt $(D, I^{'})$ eine *Erweiterung* von (D, I) zu $S^{'}$ wenn gilt:

- $(D, I^{'})$ ist eine Interpretation von $S^{'}$
- 1. $I^{'}(S) = I(S)$ mit $S \in So$
  2. $I^{'}(K) = I(K)$ mit $K \in \alpha$
  3. $I^{'}(F) = I(F)$ mit $F \in \alpha$
  4. $I^{'}(P) = I(P)$ mit $P \in \beta$

Definition (relativ Gültigkeit)

Seien $S, S^{'}$ Signaturen so dass $S^{'}$ die *Erweiterung* von S durch ein S_e ist. Sei ein Fo eine $S^{'}$-Formel und (D, I) eine S - Interpretation.
Dann ist Fo *gültig relativ* zu (D,I) genau dann, wenn für jede *Erweiterung* $(D, I^{'})$ von (D, I) zu $S^{'}$ und für jede Variablenbelegung $\epsilon^{'}$ auf D gilt
$[[Fo]]_{(D,I^{'},\epsilon^{'})} = w$.

Definition (Modell)

Sei S eine Signatur und (D, I) eine zugehörige Interpretation. Gilt $[[Fo]]_{(D,I,\epsilon)} = w$ für eine Formel $Fo \in F$ und allen Variablenbelegungen ϵ, so heißt (D, I) ein *Modell* von Fo.

Definition (Entscheidungsprozedur)

Seien S eine Signatur und (D, I) ein *Modell* von S.
Sei S_p eine *Untersignatur* von S und sei (D, I_p) die *Einschränkung* von (D, I) auf S_p (d.h. (D, I_p) ist ein Modell von S_p)
Dann ist P eine *Entscheidungsprozedur* für S_p mit Interpretation (D, I_p) genau dann wenn für alle *Erweiterungen* von S_p zu einem $S^{'}$ durch ein S_e und für jede Formel Fo in $S^{'}$ gilt:
Wenn die Anwendung von P auf Fo die Antwort „gültig" liefert dann ist Fo gültig relativ zu (D, I_p)

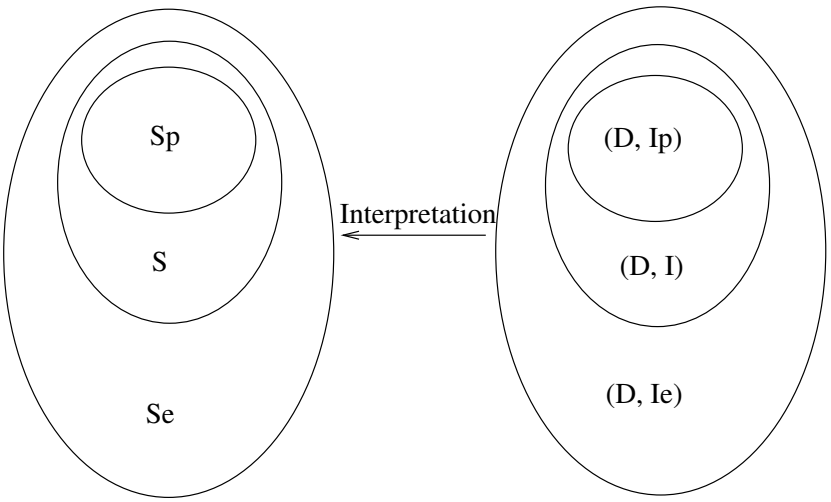

Abbildung 3.1: Interpretation

(D, I_p) Interpretation der Entscheidungsprozedur
(D, I) Interpretation
(D, I_e) Erweiterte Interpretation
S_p Signatur der Entscheidungsprozedur
S Signatur
S_e Erweiterung einer Signatur

Satz (*Fo* gültig relativ zu)

Seien S_p und S Signaturen sodass S_p die *Erweiterung* von S_p durch ein S_e ist. Sei Fo eine S-Formel und (D, I) eine S Interpretation und I_p, sodass (D, I_p) die Einschränkung von (D, I) auf S_p ist.
Dann gilt:
Fo ist gültig relativ zu $(D, I_p) \Rightarrow Fo$ gültig in (D, I).

Beweis:
Annahme: Fo ist gültig relativ zu (D, I_p)
Zu zeigen: Fo gültig in (D, I)
Zu zeigen: Für jede Variablenbelegung ϵ auf D gilt $[[Fo]]_{(D,I,\epsilon)} = w$.
Sei ϵ eine Variablenbelegung auf D.
Zu zeigen: $[[Fo]]_{(D,I,\epsilon)} = w$.
Wir wissen auf Grund der Annahme dass für jede Erweiterung (D, I) von (D, I_p) zu S und für jede Variablenbelegung ϵ auf D gilt $[[Fo]]_{(D,I_p,\epsilon_p)} = w$.
Es bleibt zu zeigen dass (D, I) eine Erweiterung von (D, I_p) zu S ist.
Also (D, I) eine Interpretation von S ist (Definition Erweiterung).
Wir wissen dass:
$I(So) = I_p(So)$ mit $So \in So$,
$I(K) = I_p(K)$ mit $K \in \alpha$,
$I(F) = I_p(F)$ mit $F \in \alpha$,
$I(P) = I_p(P)$ mit $P \in \beta$
für alle $(So, K, F, P) \in S_p$
Daher gilt der Satz.

3.3 Logik höherer Ordnung

Die meisten Entscheidungsprozeduren verstehen Prädikatenlogik der ersten Stufe (PL1), wo hingegen interaktive Beweissysteme „Higher Order Logic“ (HOL) beherrschen, da sie leichter zu handhaben ist. Im nächsten Punkt wird geklärt wie man von PL1 auf HOL gelangt.

3.3.1 Von PL1 zu HOL

Die Prädikatenlogik 1.Stufe ist trotz ihrer relativ starken Ausdrucksfähigkeit bei weitem nicht mächtig genug, um alle Sachverhalte beschreiben zu können. Schon die natürlichen Zahlen lassen sich in PL1 nicht formulieren, da über Prädikate quantifiziert werden müsste. Die Prädikatenlogik zweiter Stufe, PL2, stellt daher zusätzlich zu den Individuenvariablen v_i auch Variablen für Funktoren v_F^k und für Prädikate v_P^k beliebiger Stelligkeit k = 0, 1, 2, . . . zur Verfügung, über die wie über Individuenvariablen quantifiziert werden darf.

$P ::= Ind \rightarrow [Ind \rightarrow Bool]$....Prädikatensymbol 2.Stufe
$F ::= Ind \rightarrow [Ind \rightarrow Ind]$.... Funktionensymbol 2.Stufe

Eine Variablenbelegung ϵ bildet nun die Individuenvariablen aus v_i wie gewohnt auf Individuen der Domain D ab. Prädikatenvariablen der Stelligkeit n werden von ϵ auf Teilmengen von D^n abgebildet. Eine Funktionsvariable $f \in v_F^k$ wird von ϵ wie erwartet auf eine Funktion $D^k \rightarrow D$ abgebildet. Ist ferner $p \in v_p^n$, so gilt $\omega_{(D,I,\epsilon)}(p(t_1, ..., t_n)) = w$ dann und nur dann wenn $(\epsilon(t_1), ..., \epsilon(t_n)) \in \epsilon(p)$ ist. Mit den Variablen aus v_P und v_F ist es möglich, Aussagen über Funktionen (z.B. deren Existenz) oder über Teilmengen der Domain D zumachen. Die Tatsache, dass von diesen überabzahlbar viele existieren, deutet schon darauf hin, dass man jetzt auch überabzahlbare Strukturen eindeutig beschreiben kann. Betrachtet man nun eine Formel $p_i \in PL2$ mit den freien Variablen $v_1, ..., v_n \in v_i \cup v_F \cup v_P$, so konnte man durch sie ein Prädikat $P(v_1, ..., v_n)$ des Typs $\alpha(P) = 2^{(\alpha(v_1)v...v\alpha(v_n))}$ definieren. P ist dann ein Prädikatensymbol zweiter Stufe. Interpretationen zweiter Stufe müssen solchen Prädikaten Teilmengen des entsprechenden Typs zuordnen, ist z.B. P

ein Prädikat zweiter Stufe über einem n-stelligem Prädikat P erster Stufe (ist also $\alpha(P) = 2^{\alpha(p)}$, so muss P eine Menge von Teilmengen von D^n, also ein Element von 2^{D^n} zugeordnet werden. PL3 stellt schließlich auch Variable für Prädikate zweiter Stufe zur Verfügung, über die man wiederum quantifizieren darf, usw. Erlaubt man schließlich eine beliebige Stufe, so spricht man von der Stufenlogik $PL\infty$.

P ::= $Ind \rightarrow [Ind \rightarrow ....[Ind \rightarrow Bool]...]....$Prädikatensymbol n.Stufe
F ::= $Ind \rightarrow [Ind \rightarrow ....[Ind \rightarrow Ind]...]....$ Funktionensymbol n.Stufe

In ihr kann man schließlich alle mathematischen Sachverhalte formulieren. Dies ist Logik Höhere Ordnung (HOL) die in den meisten interaktiven Beweissystemen verwendet wird.

3.3.2 HOL mit Typen

Datentypen

Notation:

$So \in$ Sorten
$P \in$ Prädikate
$F \in$ Funktionen
$Ty \in$ Typ

$Ty ::= So_1|So_2|...|So_n|F|P.$
$F ::= Ty_1x...xTy_n. \rightarrow Ty.$
$P ::= Ty_1x...xTy_n.$

Definition (Signatur)

Eine Signatur S bezeichnet das Quartupel $(So, \alpha, \beta, \gamma)$, wobei $So = \{So_1, ..., So_n\}$, $\alpha = \{F \in$ Funktionen $\}$ und $\beta = \{P \in$ Prädikate $\}$ ist. Jedes Element von β wird S-Formel genannt. Außerdem definieren wir die Stelligkeit für jedes Prädikat $\gamma_P : \beta \rightarrow \{Ty_1x...xTy_n\}$, für jede Funktion $\gamma_F : \alpha \rightarrow \{Ty_1x...xTy_n \rightarrow F\}$, für jedes $v \in$ Variablen mit $\gamma_v(v) = Ty_i$ für ein $Ty_i \in$ Typ und für jede $k \in$ Konstanten mit $\gamma_k(k) \in$ Konstanten, mit den Eigenschaften:

- α, β sind höchstens abzählbare unendliche Mengen
- $\alpha \cap \beta = \{\ \}$

Nun wird die Syntax von Termen, Atomen und Formeln der getypten Logik der höheren Ordnung definiert.

Syntax (Terme)

Ist S eine Signatur dann ist die Menge A der getypten Terme wie folgt definiert

- für jede Variable $v \in V$ mit $\gamma_V(v) = Ty$ gilt $(v : Ty) \in A$
- für jede Konstante $k \in K$ mit $\gamma_K(k) = Ty$ gilt $(k : Ty) \in A$
- für jedes $F \in \alpha$ mit $\gamma_\alpha(F) = (Ty_1, ..., Ty_n) \rightarrow Ty$ gilt
  $(F : Ty_1, ..., Ty_n) \in A$
- für jedes $T \in \alpha$ mit $\gamma_\alpha(T) = (Ty_1, ..., Ty_n) \rightarrow Ty$ und $T_1 : Ty_1, ..., T_n : Ty_n \in T$ gilt
  $(T : Ty_1 x ... x Ty_n \rightarrow T)(T_1 : Ty_1, ..., T_n : Ty_n) \in A$

Syntax (Atome)

Ist S eine Signatur dann ist die Menge A der getypten atomaren Formeln wie folgt definiert

- Wenn $P \in \beta, \gamma(P) = ()$ so gilt $P \in A$
- Wenn $P \in \beta$ mit $\gamma_\beta(P) = (Ty_1 x ... x Ty_n)$ gilt
  $(P : Ty_1, ..., Ty_n) \in A$
- Wenn $T \in \beta$ mit $\gamma_\beta(T) = (Ty_1 x ... x Ty_n)$ und $T_1 : Ty_1, ..., T_n : Ty_n \in T$ gilt
  $(T : Ty_1 x ... x Ty_n)(T_1 : Ty_1, ..., T_n : Ty_n) \in A$

Syntax (Formeln)

Fo ist die Menge der Formeln über der *Sigantur* S wenn gilt

- $A \subseteq Fo$
- Wenn $F_1 \in Fo$, $F_2 \in Fo$ und $v_i \in V$ dann gilt

  1. Negation: $\neg F_1 \in Fo$
  2. Konjunktion: $F_1 \wedge F_2 \in Fo$
  3. Disjunktion: $F_1 \vee F_2 \in Fo$
  4. Implikation: $F_1 \rightarrow F_2 \in Fo$
  5. Äquivalenz: $F_1 \leftrightarrow F_2 \in Fo$
  6. Generalisation: $\forall v_i.F_1 \in Fo$
  7. Partikulation: $\exists v_i.F_1 \in Fo$

Semantik

Definition ((D,I) ist eine S Interpretation)

Eine S-Interpretation ist ein Paar (D,I) sodass:

1. $I(So_i) \in D$, für alle $So_i \in Ty$
2. Wenn $Ty_1, ..., Ty_n, Ty \in D$, dann ist auch $Ty_1 x ... x Ty_n \in D$ und $Ty_1 x ... x Ty_n \rightarrow Ty \in D$
3. $K \in \alpha$ mit $\gamma_F(K) = Ty \Rightarrow I(K) \in I(Ty)$
4. $F \in \alpha$ mit $\gamma_F(F) = Ty_1 x ... x Ty_n \rightarrow Ty \Rightarrow I(F) \in I(Ty_1) x ... x I(Ty_n) \rightarrow I(Ty)$
5. $P \in \beta$ mit $\gamma_P(P) = () \Rightarrow I(P) \in \{w, f\}$
6. $P \in \beta$ mit $\gamma_P(P) = (Ty_1 x ... x Ty_n) \Rightarrow I(P) \subseteq I(Ty_1) x ... x I(Ty_n)$

Bemerkung:
D ist damit die Menge der Domains, in denen die Interpretation der Prädikate und Funktionen von S enthalten sind.

Definition (Termauswertung)

Sei S eine Signatur und (D, I) eine Interpretation. Jede Funktion ϵ heißt dann eine Variablenbelegung, wenn für jedes $v \in$Variablen gilt $\epsilon(v) \in I(\gamma_V(v))$
Dazu definiert man eine Termauswertung $[[T : Ty]]_{(D',I',\epsilon')} : T \rightarrow D$ wie folgt.

- Ist $T \in V$ so gilt $[[T : Ty]]_{(D',I',\epsilon')} := \epsilon(T)$.
- Ist $T \in \alpha$ mit $\gamma(T) = ()$, dann ist $[[T : Ty]]_{(D',I',\epsilon')} := I(T)$.
- Ist $T \in \alpha$ mit $\gamma_\alpha(T) = (Ty_1 x ... x Ty_n) \rightarrow T$ und seien $T_1, ..., T_n \in T$ sodass $I(T_1) \in I(Ty_1), ..., I(T_n) \in I(Ty_n)$, dann ist
  $(T_0 : Ty_1 x ... x Ty_n \rightarrow T)(T_1 : Ty_1, ..., T_n : Ty_n) := [[T_0 : Ty_1 x ... x Ty_n \rightarrow T]]_{(D',I',\epsilon')}([[T_1 : Ty_1]]_{(D',I',\epsilon')}, ..., [[T_n : Ty_n]]_{(D',I',\epsilon')})$

Definition (Atomauswertung)

Sei S eine Signatur und (D, I) eine Interpretation, dann ist die Atomauswertung definiert als.

- $[[P()]] := \begin{cases} w & : w = I(P) \\ f & : \text{sonst} \end{cases}$
- $[[T : Ty_1 x ... x Ty_n]][[T(T_1 : Ty_1, ..., T_n : Ty_n)]]$
  $:= \begin{cases} w & : ([[T_1 : Ty_1]], ..., [[T_n : Ty_n]]) \in [[T : Ty_1 x ... x Ty_n]] \\ f & : \text{sonst} \end{cases}$

Definition (Formelauswertung ($[[Fo]]_{(D,I,\epsilon)} \in \{w, f\}$))

Sei S eine Signatur, (D, I) eine Interpretation, ϵ eine Variablenbelegung und $p_i, p_j \in Fo$, dann wird die Formelauswertung $[[Fo]]_{(D,I,\epsilon)} : F \rightarrow \{w, f\}$ definiert als.

- $[[\neg p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = f \\ f & : [[p_i]]_{(D,I,\epsilon)} = w \end{cases}$
- $[[p_i \wedge p_j]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} = w \\ f & : \text{sonst} \end{cases}$
- $[[p_i \vee p_j]]_{(D,I,\epsilon)} := \begin{cases} f & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} = f \\ w & : \text{sonst} \end{cases}$
- $[[p_i \rightarrow p_j]]_{(D,I,\epsilon)} := \begin{cases} f & : [[p_i]]_{(D,I,\epsilon)} = w \text{ und } [[p_j]]_{(D,I,\epsilon)} = f \\ w & : \text{sonst} \end{cases}$
- $[[p_i \leftrightarrow p_j]]_{(D,I,\epsilon)} := \begin{cases} w & : [[p_i]]_{(D,I,\epsilon)} = [[p_j]]_{(D,I,\epsilon)} \\ f & : \text{sonst} \end{cases}$
- $[[\forall v_i . p_i]]_{(D,I,\epsilon)} := \begin{cases} w & : \text{für alle } d \in D \text{ gilt} : [[p_i]]_{(D,I,\epsilon[v_i \rightarrow d]} = w \\ f & : \text{sonst} \end{cases}$

- $[[\exists v_i.p_i]]_{(D,I,\epsilon)} := \begin{cases} w & \text{: für mindestens ein } d \in D \text{ gilt : } [[p_i]]_{(D,I,\epsilon[v_i \to d]} = w \\ f & \text{: sonst} \end{cases}$

3.3.3 Übersetzung von der Logik der höheren Ordnung mit Typen in die Prädikatenlogik der ersten Stufe

Jeder Formel F in HOL (Higher Order Logic) mit Typen entspricht eine Formel $\bar{F}$ in PL_n (Prädikaten Logik n-ter Stufe) für ein $n \in N$. Man kann diese Formel $\bar{F}$ durch eine Funktion T von PL_n auf PL_1 in eine neue Formel $F^{'} = T(F)$ übersetzen, die genau dann gültig ist, wenn F gültig ist.
Für die Übersetzung von Typen verwenden wir eine Funktion T, welches Typen in HOL in Typen in FOL übersetzt. Für die Übersetzung von atomaren Formeln verwenden wir eine Funktion TF, welche atomare Formeln in HOL in atomare Formeln in FOL übersetzt. Für die Übersetzung von Termen verwenden wir eine Funktion TT, welche Terme in HOL in Terme, in FOL übersetzt.
Für jeden Funktionstyp $A_1x...xA_n \rightarrow B$ gibt es eine Sorte $So_{A_1x...xA_n \rightarrow B}$.
Für jeden Prädikattyp $A_1x...xA_n$ gibt es eine Sorte $So_{A_1x...xA_n}$.
Zu Beginn möchte ich ein Beispiel für die Übersetzungsfunktion Anw bringen, wie man von HOL auf PL_1 mittels der Übersetzungsfunktion Anw gelangt.
Eine Funktion in HOL:
$\forall f : A \rightarrow B, x : A \exists y : B : f(x) = y$
wird durch die Funktionskonstante $Anw()$ auf die für PL_1 verständliche Form
$\forall f : So_{A \rightarrow B}, x : A \exists y : B : Anw_{So_{A \rightarrow B}}(f, x) = y$
gebracht.

Sorten für Funktionen

I sei eine Interpretation und $So_{A_1x...xA_n \to B}$ eine Sorte für Funktionen.
Dann ist $I[[So_{A_1x...xA_n \to B}]] = I[[A_1]]x...xI[[A_n]] \to I[[B]]$
Dies nennt man eine mengentheoretische Funktion

Sorten für Prädikate

I sei eine Interpretation und $So_{A_1x...xA_n}$ eine Sorte für Prädikate.
Dann ist $I[[So_{A_1x...xA_n}]] = I[[A_1]]x...xI[[A_n]]$
Dies nennt man ein mengentheoretisches Prädikat

Interpretation von Prädikaten

Sei $Gilt_{A_1x...xA_n}$ das Ergebnis der Übersetzungsfunktion für Prädikate.
Dann ist die dazugehörige Interpretation I definiert als
$I[[Gilt_{A_1x...xA_n}]](p, a_1...a_n) = w$ genau dann wenn $p(a_1...a_n)$ (das heißt $(a_1...a_n) \in p$)
für $p \in I[[A_1x...xA_n]]$
und für $a_1...a_n \in I[[A_1x...xA_n]]$.

Interpretation von Funktionen

$Anw_{A_1x...xA_n \to B}$ ist eine neue $n+1$-stellige Funktionskonstante, die die Anw einer Funktion $f : A_1x...xA_n \to B$ auf n Argumente mit Typ $A_1, ..., A_n$ abbildet. Die dazugehörige Interpretation I ist definiert als
$I[[Anw_{A_1x...xA_n \to B}]](f, a_1...a_n) = f(a_1, ..., a_n, b)$ (das heißt solch ein Wert b so dass $\langle a_1, ..., a_n, b \rangle \in f$)
für $f \in I[[A_1x...xA_n \to B]]$
und für $a_1...a_n \in I[[A_1x...xA_n]]$ ist.

Konstanten

- Sei K_F ein bestimmter Typ so dass $\gamma(F) = A_1 x ... x A_n \rightarrow B$ ist, dann ist $\gamma(K_F) = So_{A_1 x ... x A_n \rightarrow B}$.
- Sei K_P ein bestimmter Typ so dass $\gamma(P) = A_1 x ... x A_n$ ist, dann ist $\gamma(K_P) = So_{A_1 x ... x A_n}$.

Übersetzung von Typen

Sei T die Übersetzungsfunktion für Typen und $So \in$ Sorten, dann gilt für die Übersetzung:

- $T[[So]] = So$
- $T[[A_1 x ... x A_n \rightarrow B]] = So_{A_1 x ... x A_n \rightarrow B}$
- $T[[A_1 x ... x A_n]] = So_{A_1 x ... x A_n}$

Übersetzungsfunktion für die Interpretation

Sei S eine Signatur, (D, I) eine S Interpretation in HOL mit der Eigenschaft aus der Definition der Interpretation (Seite 61), ϵ ein dazugehörige Variablenbelegung (Seite 61).

Wir definieren die Übersetzungsfunktion $TI(D, I, \epsilon) = (D^{'}, I^{'}, \epsilon^{'})$ wobei mit $(D^{'}, I^{'})$ eine $S^{'}$ Interpretation in FOL mit Typen (Seite 49) ist und $\epsilon^{'}$ eine dazugehörige Variablenbelegung (Seite 49) ist.
Dann ist $D^{'}, I^{'}, \epsilon^{'}$ definiert als:

1. $D^{'} = D$
2. $I^{'}(k) = I(k)$ mit $k \in \alpha$
3. $I^{'}(F) = I(F)$ mit $F \in \alpha$
4. $I^{'}(P) = I(P)$ mit $p \in \beta$
5. $\epsilon^{'}(v) = \epsilon(v)$ mit $v \in \alpha$
6. $I^{'}(Gilt_{A_1 x ... x A_n})(p, a_1 ... a_n) = w \Leftrightarrow (a_1 ... a_n) \in p$ für $p \in I[[A_1 x ... x A_n]]$
7. $I^{'}(Anw_{A_1 x ... x A_n \rightarrow B})(f, a_1 ... a_n)$ = solch ein Wert b sodass $\langle a_1, ..., a_n, b \rangle \in f$ für $f \in I[[A_1 x ... x A_n \rightarrow B]]$

Übersetzung von Termen

Sei TT die Übersetzungsfunktion für Terme dann gilt für $v \in$ Variablen, $k \in$ Konstanten, $F \in$ Funktionen, $P \in$ Prädikate und $(T_1, ..., T_n) \in$ Terme:

- $TT[[v : Ty]] = v$
- $TT[[k : Ty]] = k$
- $TT[[F : Ty_1 x ... x Ty_n \rightarrow Ty]] = K_F \in$ Konstanten
- $TT[[P : Ty_1 x ... x Ty_n]] = K_P \in$ Konstanten

- $TT[[(T_0 : Ty_1x...xTy_n \rightarrow T)(T_1 : Ty_1, ..., T_n : Ty_n)]] =$
  $Anw_{T_1x...xT_n \rightarrow T} : (Ty_1x...xTy_n \rightarrow T)(TT[[T : Ty_1x...xTy_n \rightarrow T]], TT[[T_1 : Ty_1]], ..., TT[[T_n : Ty_n]])$

Übersetzung von Formeln

Sei TF die Übersetzungsfunktion für Formeln, dann gilt für $Fo, Fo_1, Fo_2 \in$ Formeln, $x \in$ Variablen und $(T_1, ..., T_n) \in$ Terme:

- $TF[[\neg Fo]] = \neg TF[[Fo]]$
- $TF[[Fo_1 \wedge Fo_2]] = TF[[Fo_1]] \wedge TF[[Fo_2]]$
- $TF[[Fo_1 \vee Fo_2]] = TF[[Fo_1]] \vee TF[[Fo_2]]$
- $TF[[Fo_1 \rightarrow Fo_2]] = TF[[Fo_1]] \rightarrow TF[[Fo_2]]$
- $TF[[Fo_1 \leftrightarrow Fo_2]] = TF[[Fo_1]] \leftrightarrow TF[[Fo_2]]$
- $TF[[\forall x : T.Fo]] = \forall x : T[[T : Ty_1x...xTy_n]].TF[[Fo]]$
- $TF[[\exists x : T.Fo]] = \exists x : T[[T : Ty_1x...xTy_n]].TF[[Fo]]$
- $TF[[(T : Ty_1x...xTy_n)(T_1 : Ty_1, ..., T_n : Ty_n)]] =$
  $Gilt_{T_1x...xT_n}(TT[[T : Ty_1x...xTy_n]], TT[[T_1 : Ty_1]], ..., TT[[T_n : Ty_n]])$

Äquivalenz der Interpretation der Terme

Für alle Terme T in HOL mit Typen gilt:
Sei (D, I, ϵ) eine Interpretation einschließlich einer Interpretation der Sorten $So_{A_1x...xA_n \rightarrow B}$ für Funktionen,
der Übersetzungsfunktion $Anw_{A_1x...xA_n \rightarrow B}$ für Funktionen.
Dann gilt:
$[[T]]_{(D,I,\epsilon)} = [[TT[T]]]_{(D',I',\epsilon')}$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.

Beweis:

- Fall $TT[[v : Ty]] = v$:
  Zu zeigen:
  $[[v : Ty]]_{(D,I,\epsilon)} = TT[[v : Ty]]_{(D',I',\epsilon')}$
  Aus der 1. Eigenschaft der Übersetzung von Termen ($TT[[v : Ty]] = v$) zu zeigen:
  $[[v : Ty]]_{(D,I,\epsilon)} = [[v : Ty]]_{(D',I',\epsilon')}$
  Aus der Termauswerung von HOL zu zeigen:
  $\epsilon'(v) = [[v : Ty]]_{(D,I,\epsilon)}$
  Aus der Termauswertung von $PL1$ zu zeigen:
  $\epsilon'(v) = \epsilon(v)$
  Gilt wegen $\epsilon'(v) = \epsilon(v)$ aus Definition Übersetzungsfunktion für die Interpretation (Seite 67)

- Fall $TT[[k : Ty]] = k$:
  Zu zeigen:
  $[[k : Ty]]_{(D,I,\epsilon)} = TT[[k : Ty]]_{(D',I',\epsilon')}$
  Aus der 2. Eigenschaft der Übersetzung von Termen ($TT[[k : Ty]] = k$) zu zeigen:
  $[[k : Ty]]_{(D,I,\epsilon)} = [[k]]_{(D',I',\epsilon')}$
  Aus der Termauswerung von HOL zu zeigen:
  $I'(k) = [[k]]_{(D,I,\epsilon)}$
  Aus der Termauswertung von $PL1$ zu zeigen:
  $I'(k) = I(k)$
  Gilt wegen $I'(k) = I(k)$ aus Definition Übersetzungsfunktion für die Interpretation (Seite 67)

- Fall $TT[[F : Ty_1 x ... x Ty_n \rightarrow Ty]] = K_F \in$ Konstanten:
  Zu zeigen:
  $[[F : Ty_1 x ... x Ty_n \rightarrow Ty]]_{(D,I,\epsilon)} = TT[[F : Ty_1 x ... x Ty_n \rightarrow Ty]]_{(D',I',\epsilon')}$
  Aus der 3. Eigenschaft der Übersetzung von Termen ($TT[[F : Ty_1 x ... x Ty_n \rightarrow Ty]] = K_F \in$ Konstanten) zu zeigen:
  $[[F : Ty_1 x ... x Ty_n \rightarrow Ty]]_{(D,I,\epsilon)} = [[K_F]]_{(D',I',\epsilon')}$
  Aus der Termauswerung von HOL zu zeigen:
  $I'(K_F) = [[K_F]]_{(D',I',\epsilon')}$
  Aus der Termauswertung von $PL1$ zu zeigen:
  $I'(K_F) = I(K_F)$

Gilt wegen $I^{'}(F) = I(F)$ aus Definition Übersetzungsfunktion für die Interpretation (Seite 67)

- Fall $TT[[P : Ty_1x...xTy_n]] = K_P \in$ Konstanten:
  Zu zeigen:
  $[[P : Ty_1x...xTy_n]]_{(D,I,\epsilon)} = TT[[P : Ty_1x...xTy_n]]_{(D^{'},I^{'},\epsilon^{'})}$
  Aus der 4. Eigenschaft der Übersetzung von Termen ($TT[[P : Ty_1x...xTy_n]] = K_P \in$ Konstanten) zu zeigen:
  $[[P : Ty_1x...xTy_n]]_{(D,I,\epsilon)} = [[K_P]]_{(D^{'},I^{'},\epsilon^{'})}$
  Aus der Termauswerung von HOL zu zeigen:
  $I^{'}(K_P) = [[K_P]]_{(D^{'},I^{'},\epsilon^{'})}$
  Aus der Termauswertung von $PL1$ folgt:
  $I^{'}(K_P) = I(K_P)$
  Gilt wegen $I^{'}(P) = I(P)$ aus Definition Übersetzungsfunktion für die Interpretation (Seite 67)

- Fall $\overline{T} = (T_0 : Ty_1x...xTy_n \rightarrow T)(T_1 : Ty_1, ..., T_n : Ty_n)$
  Zu zeigen:
  $[[\overline{T}]]_{(D,I,\epsilon)} = [[TT[\overline{T}]]]_{(D^{'},I^{'},\epsilon^{'})}$
  Zu zeigen:
  $[[(T_0 : Ty_1x...xTy_n \rightarrow T)(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D,I,\epsilon)} = [[TT[\overline{T}]]]_{(D^{'},I^{'},\epsilon^{'})}$
  Wir wissen aus der Definition der Termauswertung:
  $[[(T_0 : Ty_1x...xTy_n \rightarrow T)(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D,I,\epsilon)} = [[T_0 : Ty_1x...xTy_n \rightarrow T]]_{(D^{'},I^{'},\epsilon^{'})}([[T_1 : Ty_1]]_{(D^{'},I^{'},\epsilon^{'})}, ..., [[T_n : Ty_n]]_{(D^{'},I^{'},\epsilon^{'})})$
  Wir wissen aus der Definition der Übersetzungsfunktion:
  $[[TT[\overline{T}]]]_{(D^{'},I^{'},\epsilon^{'})} = Anw_{T_1x...xT_n \rightarrow T} : (Ty_1x...xTy_n \rightarrow T)(TT[[T : Ty_1x...xTy_n \rightarrow T]]_{(D^{'},I^{'},\epsilon^{'})}, TT[[T_1 : Ty_1]]_{(D^{'},I^{'},\epsilon^{'})}, ..., TT[[T_n : Ty_n]]_{(D^{'},I^{'},\epsilon^{'})})$
  Wir wissen aus der Definition der Termauswertung:
  $Anw_{T_1x...xT_n \rightarrow T} : (Ty_1x...xTy_n \rightarrow T)(TT[[T : Ty_1x...xTy_n \rightarrow T]], TT[[T_1 : Ty_1]], ..., TT[[T_n : Ty_n]])_{(D^{'},I^{'},\epsilon^{'})} = Anw_{T_1x...xT_n \rightarrow T} : (Ty_1x...xTy_n \rightarrow T)_{(D^{'},I^{'},\epsilon^{'})}$
  $(TT[[T : Ty_1x...xTy_n \rightarrow T]]_{(D^{'},I^{'},\epsilon^{'})},$
  $TT[[T_1 : Ty_1]]_{(D^{'},I^{'},\epsilon^{'})}, ..., TT[[T_n : Ty_n]]_{(D^{'},I^{'},\epsilon^{'})}) \overset{I.A.}{=}$
  $Anw_{T_1x...xT_n \rightarrow T} : (Ty_1x...xTy_n \rightarrow T)_{(D^{'},I^{'},\epsilon^{'})}$
  $(TT[[T : Ty_1x...xTy_n \rightarrow T]]_{(D,I,\epsilon)}, TT[[T_1 : Ty_1]]_{(D,I,\epsilon)}, ..., TT[[T_n : Ty_n]]_{(D,I,\epsilon)})$
  Wir wissen aus der Interpretation von Funktionen:
  $Anw_{T_1x...xT_n \rightarrow T} : (Ty_1x...xTy_n \rightarrow T)_{(D^{'},I^{'},\epsilon^{'})}(TT[[T : Ty_1x...xTy_n \rightarrow T]]_{(D,I,\epsilon)},$

$$TT[[T_1 : Ty_1]]_{(D,I,\epsilon)}, ..., TT[[T_n : Ty_n]]_{(D,I,\epsilon)}) =$$
$$[[T_0 : Ty_1 x ... x Ty_n \rightarrow T]]_{(D',I',\epsilon')}([[T_1 : Ty_1]]_{(D',I',\epsilon')}, ..., [[T_n : Ty_n]]_{(D',I',\epsilon')})\bullet$$

Äquivalenz der Interpretation der Formeln

Für alle Formeln F in HOL mit Typen gilt:
Sei (D, I, ϵ) eine Interpretation einschließlich einer Interpretation der Sorten
$So_{A_1 x ... x A_n \rightarrow B}$ für Funktionen,
der Sorten $So_{A_1 x ... x A_n}$ für Prädikate,
der Übersetzungsfunktion $Anw_{A_1 x ... x A_n \rightarrow B}$ für Funktionen und
der Übersetzungsfunktion $Gilt_{A_1 x ... x A_n}$ für Prädikate.

Dann gilt:
$[[F]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[F]]]_{(D',I',\epsilon')} = w$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.

Beweis:
Über strukurelle Induktion

Fall: $F = (T : Ty_1 x ... x Ty_n)(T_1 : Ty_1, ..., T_n : Ty_n)$
Für alle (D, I, ϵ) zu zeigen:
$[[F]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[F]]]_{(D',I',\epsilon')} = w$
Zu zeigen:
$[[(T : Ty_1 x ... x Ty_n)(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D,I,\epsilon)} \Leftrightarrow [[TF[F]]]_{(D',I',\epsilon')} = w$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.
Wir wissen aus der Definition der Atomauswertung:
$[[(T : Ty_1 x ... x Ty_n)(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D,I,\epsilon)} = w \Leftrightarrow$
$[[TT[[T : Ty_1 x ... x Ty_n]]_{(D',I',\epsilon')}, ([[T_1 : Ty_1]]_{(D',I',\epsilon')}, ..., TT[[T_n : Ty_n]]_{(D',I',\epsilon')}) = w$
Wir wissen aus der Definition der Übersetzungsfunktion für Formeln:
$TF[[(T : Ty_1 x ... x Ty_n)(T_1 : Ty_1, ..., T_n : Ty_n)]]_{(D',I',\epsilon')} = w \Leftrightarrow Gilt_{T_1 x ... x T_n}(TT[[T : Ty_1 x ... x Ty_n]], TT[[T_1 : Ty_1]], ..., TT[[T_n : Ty_n]])_{(D',I',\epsilon')} = w$
$Gilt_{T_1 x ... x T_n}(TT[[T : Ty_1 x ... x Ty_n]], TT[[T_1 : Ty_1]], ..., TT[[T_n : Ty_n]])_{(D',I',\epsilon')} = w \Leftrightarrow$
$Gilt_{T_1 x ... x T_n}$
$(TT[[T : Ty_1 x ... x Ty_n]]_{(D',I',\epsilon')}, TT[[T_1 : Ty_1]]_{(D',I',\epsilon')}, ..., TT[[T_n : Ty_n]]_{(D',I',\epsilon')}) = w$
$\overset{I.A.}{\leftrightarrow} Gilt_{T_1 x ... x T_n} : (Ty_1 x ... x Ty_n)_{(D',I',\epsilon')}$
$(TT[[T : Ty_1 x ... x Ty_n]]_{(D,I,\epsilon)}, TT[[T_1 : Ty_1]]_{(D,I,\epsilon)}, ..., TT[[T_n : Ty_n]]_{(D,I,\epsilon)}) = w$
Wir wissen aus der Interpretation von Prädikaten:

$Gilt_{T_1x...xT_n} : (Ty_1x...xTy_n)_{(D',I',\epsilon')}$
$(TT[[T : Ty_1x...xTy_n]]_{(D,I,\epsilon)}, TT[[T_1 : Ty_1]]_{(D,I,\epsilon)}, ..., TT[[T_n : Ty_n]]_{(D,I,\epsilon)}) = w \Leftrightarrow$
$[[T : Ty_1x...xTy_n]]_{(D',I',\epsilon')}([[T_1 : Ty_1]]_{(D',I',\epsilon')}, ..., [[T_n : Ty_n]]_{(D',I',\epsilon')}) = w\bullet$

Fall: $F = \neg Fo$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[F]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[\neg Fo]]]_{(D',I',\epsilon')} = w$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[\neg Fo]]_{(D',I',\epsilon')} = w \Leftrightarrow [[Fo]]_{(D',I',\epsilon')} = f$
$\overset{I.A.}{\Leftrightarrow} [[Fo]]_{(D,I,\epsilon)} = f \Leftrightarrow [[\neg Fo]]_{(D,I,\epsilon)} = w\bullet$

Fall: $F_1 \wedge F_2 = Fo_1 \wedge Fo_2$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[F_1 \wedge F_2]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[Fo_1 \wedge Fo_2]]]_{(D',I',\epsilon')} = w$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[Fo_1 \wedge Fo_2]]_{(D',I',\epsilon')} = w \Leftrightarrow [[Fo_1]]_{(D',I',\epsilon')} = [[Fo_2]]_{(D',I',\epsilon')} = w$
$\overset{I.A.}{\Leftrightarrow} [[Fo_1]]_{(D,I,\epsilon)} = [[Fo_2]]_{(D,I,\epsilon)} = w \Leftrightarrow [[Fo_1 \wedge Fo_2]]_{(D,I,\epsilon)} = w\bullet$

Fall: $F_1 \vee F_2 = Fo_1 \vee Fo_2$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[F_1 \vee F_2]]_{(D,I,\epsilon)} = f \Leftrightarrow [[TF[Fo_1 \vee Fo_2]]]_{(D',I',\epsilon')} = f$

wobei $(D^{'}, I^{'}, \epsilon^{'}) = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[Fo_1 \vee Fo_2]]_{(D^{'},I^{'},\epsilon^{'})} = f \Leftrightarrow [[Fo_1]]_{(D^{'},I^{'},\epsilon^{'})} = [[Fo_2]]_{(D^{'},I^{'},\epsilon^{'})} = f$
$\overset{I.A.}{\leftrightarrow} [[Fo_1]]_{(D,I,\epsilon)} = [[Fo_2]]_{(D,I,\epsilon)} = f \Leftrightarrow [[Fo_1 \vee Fo_2]]_{(D,I,\epsilon)} = f \bullet$

Fall: $F_1 \rightarrow F_2 = Fo_1 \rightarrow Fo_2$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[F_1 \rightarrow F_2]]_{(D,I,\epsilon)} = f \Leftrightarrow [[TF[Fo_1 \rightarrow Fo_2]]]_{(D^{'},I^{'},\epsilon^{'})} = f$
wobei $(D^{'}, I^{'}, \epsilon^{'}) = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[Fo_1 \rightarrow Fo_2]]_{(D^{'},I^{'},\epsilon^{'})} = f \Leftrightarrow [[Fo_1]]_{(D^{'},I^{'},\epsilon^{'})} = w$ und $[[Fo_2]]_{(D^{'},I^{'},\epsilon^{'})} = f$
$\overset{I.A.}{\leftrightarrow} [[Fo_1]]_{(D,I,\epsilon)} = w$ und $[[Fo_2]]_{(D,I,\epsilon)} = f \Leftrightarrow [[Fo_1 \rightarrow Fo_2]]_{(D,I,\epsilon)} = f \bullet$

Fall: $F_1 \leftrightarrow F_2 = Fo_1 \leftrightarrow Fo_2$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[F_1 \leftrightarrow F_2]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[Fo_1 \leftrightarrow Fo_2]]]_{(D^{'},I^{'},\epsilon^{'})} = w$
wobei $(D^{'}, I^{'}, \epsilon^{'}) = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[Fo_1 \leftrightarrow Fo_2]]_{(D^{'},I^{'},\epsilon^{'})} = w \Leftrightarrow [[Fo_1]]_{(D^{'},I^{'},\epsilon^{'})} = [[Fo_2]]_{(D^{'},I^{'},\epsilon^{'})}$

$\overset{I.A.}{\leftrightarrow} [[Fo_1]]_{(D,I,\epsilon)} = [[Fo_2]]_{(D,I,\epsilon)} \Leftrightarrow [[Fo_1 \leftrightarrow Fo_2]]_{(D,I,\epsilon)} = w\bullet$

Fall: $\forall v_i.F = \forall v_i.Fo$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[\forall v_i.F]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[\forall v_i.Fo]]_{(D',I',\epsilon')} = w$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[\forall v_i.Fo]]_{(D',I',\epsilon')} = w \Leftrightarrow [[Fo]]_{(D',I',\epsilon'_{v_i \to d})} = w$ für jedes $d \in D$
$\overset{I.A.}{\leftrightarrow} [[Fo]]_{(D,I,\epsilon_{v_i \to d})} = w \Leftrightarrow [[\forall v_i.Fo]]_{(D,I,\epsilon)} = w$ für jedes $d \in D\bullet$

Fall: $\exists v_i.F = \exists v_i.Fo$
Zu zeigen:
Für alle (D, I, ϵ) gilt
$[[\exists v_i.F]]_{(D,I,\epsilon)} = w \Leftrightarrow [[TF[\exists v_i.Fo]]_{(D',I',\epsilon')} = w$
wobei $(D', I', \epsilon') = TI(D, I, \epsilon)$ ist.

Sei (D, I, ϵ) beliebig.

Wir wissen:
$[[\exists v_i.Fo]]_{(D',I',\epsilon')} = w \Leftrightarrow [[Fo]]_{(D',I',\epsilon'_{v_i \to d})} = w$ für ein $d \in D$
$\overset{I.A.}{\leftrightarrow} [[Fo]]_{(D,I,\epsilon_{v_i \to d})} = w \Leftrightarrow [[\exists v_i.Fo]]_{(D,I,\epsilon)} = w$ für ein $d \in D\bullet$

Lemma: Gültigkeit

Sei Ty die Menge der Sorten und seien $A := \{Anw_{T_1 x ... x T_n \rightarrow T} | T_1, ..., T_n, T \in Ty\}$ und $G := \{Gilt_{T_1 x ... x T_n} | T_1, ..., T_n \in Ty\}$ Mengen von Funktionskonstanten und Prädikatenkonstanten. Sei F eine Formel aus HOL mit Typen, in der die Sorten aus Ty, die Funktionskonstanten aus A und die Prädikatenkonstanten aus G nicht vorkommen. Sei $TF[[F]]$ eine Übersetzungsfunktion für Formeln. Dann gilt für alle F aus HOL mit Typen:
Ist $TF[[F]]$ gültig in PL1 mit Typen, dann ist F ist gültig in HOL mit Typen.

Beweis:

Annahme: $TF[[F]]$ ist gültig in PL1 mit Typen
Zu zeigen: F ist gültig in HOL mit Typen.
Aus $TF[[F]]$ gültig in PL1 mit Typen folgt:
$TF[[F]]_{(D,I,\epsilon)} = w$ für alle (D, I, ϵ).
Sei (D, I, ϵ) eine beliebige Interpretation und sei $(D^{'}, I^{'}, \epsilon^{'}) = TI(D, I, \epsilon)$.
Aufgrund der Definition von TI wissen wir, dass
$D = D^{'}$
$I = I^{'} \setminus (\{\langle So, i\rangle : So \in Ty\} \cup \{\langle f, i\rangle : f \in A\} \cup \{\langle p, i\rangle : p \in G\})$
$\epsilon = \epsilon^{'}$ ist.
Aus der Äquivalenz der Interpretation der Formeln wissen wir, dass
$[[F]]_{(D^{'},I^{'},\epsilon^{'})} = w$
Da die Sorten aus Ty, die Funktionskonstanten und die Prädikatenkonstanten aus A und G nicht in F vorkommen, wissen wir aus der Definition von TI, der Äquivalenz der Interpretation der Formeln und der Definition von $[[]]$, dass
$[[F]]_{(D,I,\epsilon)} = w$ ist.•

4 Software-Schnittstelle

Im ersten Abschnitt dieses Kapitels wird der ProofNavigator [38], ein interaktives Beweissystem, vorgestellt. Dieses System benötigt automatische Beweisverfahren, um die Gültigkeit von Aussagen zu prüfen.
Die beiden in den ProofNavigator eingebundenen automatischen Entscheidungsprozeduren sind CVC Light 2.0 und CVC3, welche im Abschnitt 4.2 Entscheidungsprozeduren näher beschrieben werden. Das interaktive Beweissystem ist ein System, das auf einer Logik der höheren Ordnung beruht. CVC Light 2.0 unterstützt ebenfalls eine Logik höherer Ordnung, im Gegensatz dazu beschränkt sich die Unterstützung von CVC3 auf eine Logik erster Stufe, sodass eine Übersetzung in die Logik (Kapitel 3) notwendig ist. Im Anschnitt 4.3 wird die Implementierung der entsprechenden Übersetzer beschrieben. Im Abschnitt 4.4 werden spezielle Anpassungen an die Logik der höheren Ordnung evaluiert.

4.1 Der RISC ProofNavigator

Der ProofNavigator [39] wird als Interaktives Beweissystem an der Kepler Universität Linz von Wolfgang Schreiner [37] entwickelt. In diesem System können verschiedene Kombinationen von Methoden ausgewählt werden, um mathematische Thesen zu beweisen. Die Eingabe in dieses Beweissystem erfolgt über die Logik der Höheren Ordnung. Der Kern der Software ist in Java programmiert, CVC Lite 2.0 und CVC3 sind die beiden automatische Beweisassistenten. Zusätzlich werden für die Implementierung und Benutzung des Beweisers folgende Software benutzt:

- CVC Lite 2.0
- RIACA OPenMath Library 2.0
- General Purpose Hash Function Algorithmus Library
- ANTLR 2.7.6b2
- Eclipse Standard Widget Toolkit 3.3
- Mozilla Firefox 2.0.X or SeaMonkey 1.1.X
- GIMP Toolkit GTK+ 2.x
- Sun JDK 5.0
- Tango Icon Library

Die GUI (grafische Benutzeroberfläche) wurde im Eclipse Standard Widget Toolkit 3.3 implementiert.

4.1.1 Benutzerinterface

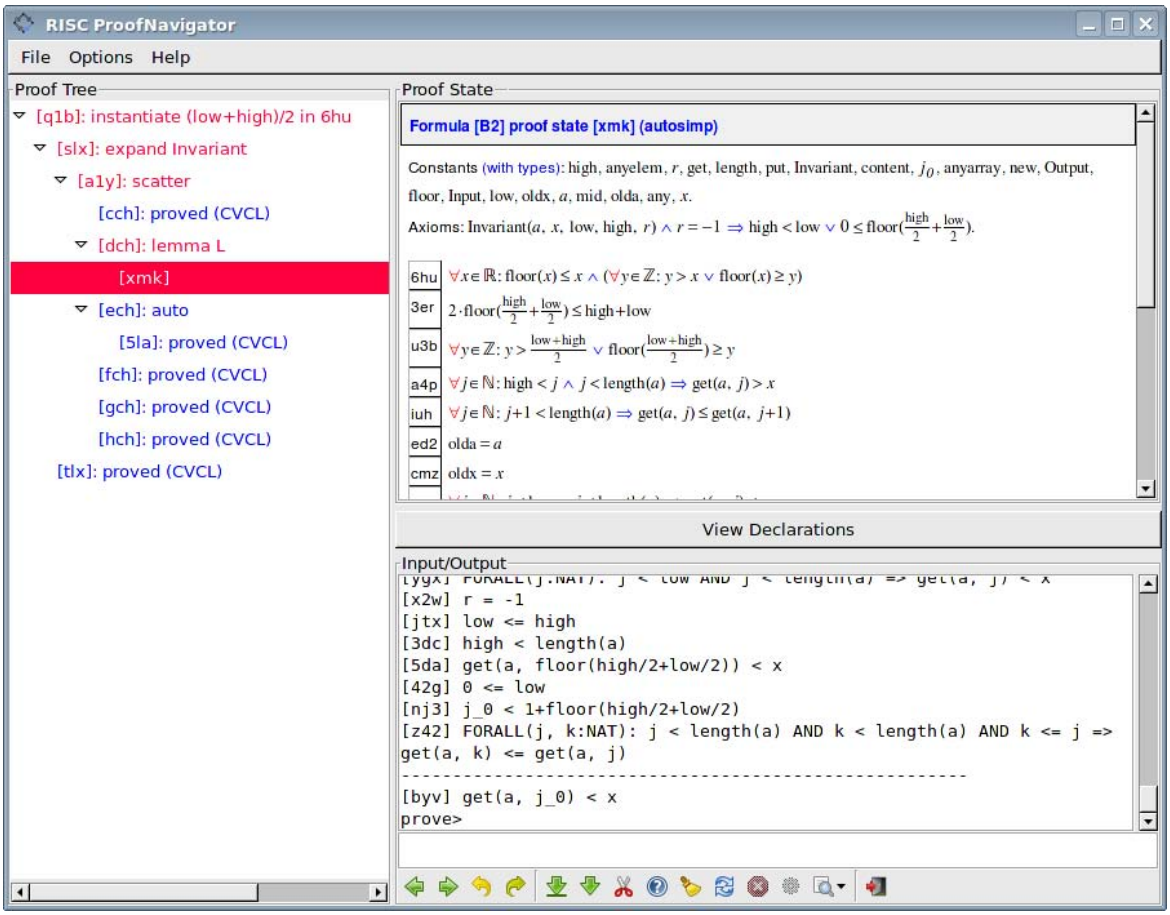

Abbildung 4.1: ProofNavigator

Wie aus der Abbildung 4.1 ersichtlich ist, wird ist die grafische Benutzeroberfläche unterteilt in *Proof Tree*, *Declarations* und *Input/Output* Durch den *Proof Tree* wird die Struktur des Beweises visualisiert, der *Declarations* Bereich enthält die Deklarationen in „Pretty Print“ und im *Input/Output* Bereich stehen die Ein und Ausgaben des Benutzers.

Die Menüleiste besteht aus den Menüpunkte, *File*, *Options* und *Help*, wobei der Menüpunkt *File* weiters unterteilt ist in *Read File*, *Restart Session* und *Quit*. Der Menüpunkt *Options* ist unterteilt in einen Schalter zwischen No Auto Simplification / Auto Simplification, einen Schalter zwischen Bigger Font / Smaller Fonts und einen Schalter zwischen CVCL / CVC3.

4.1.2 Eingabesprache

In diesem Abschnitt wird die Eingabesprache des RISC ProofNavigators durch Beispiele gezeigt. Die genaue Definition der einzelnen Sprachelemente findet man im Manual des ProofNavigators im Anhang A. [**?**] Als Erstes werden einige grundsätzliche Regeln der Sprache des Navigators vorgestellt:

- Die Eingabe von Schlüsselwörtern erfolgt ausschließlich in Grossbuchstaben.
- Durch das „%“ Zeichen werden Kommentare deklariert, welche bis zum Ende der Zeile gelten.
- Mit einem Strichpunkt wird die Eingabe bestätigt.

Das Typensystem versteht die Logik der höheren Ordnung mit einer Gleichbehandlung von Funktionen und Prädikaten, wobei die Resultate bei Prädikaten mit „TRUE“ und „FALSE“ definiert werden. Bei den atomaren Typen wird zwischen benutzerdefinierten Typen und fixen Typen (vordefinierte Typen) unterschieden.

Beispiele für fixe Typen:

t1: BOOLEAN;
t2: NAT;
t3: INT;
t4: REAL;

BOOLEAN besteht aus den Konstanten „TRUE“ und „FALSE“
NAT ist die Menge der natürlichen Zahlen
INT ist die Menge der ganzen Zahlen
REAL ist die Menge der reelle Zahlen

Im Gegensatz zu den fixen Typen müssen die benutzerdefinierten atomaren Typen deklariert werden.

Beispiel:

t1: TYPE = SUBTYPE(LAMBDA (x:INT): x >= 0 AND x <= 10);
t2: TYPE t1;

Der Typ t1 ist aus der Menge der ganzen Zahlen wobei x zwischen 0 und 10 liegt und t2 ist vom Typ t1.

Untertypen werden deklariert durch das Signalwort *SUBTYPE*.
t5: SUBTYPE(LAMBDA(x:INT): x >= 0);

Dieser Untertyp ist äquivalent zu dem Typ der natürlichen Zahlen NAT.

Beispiel für einen Array:
INDEX: TYPE = NAT;
ELEM: TYPE = INT;
ARR: TYPE = [INDEX, ARRAY INDEX OF ELEM];

Dieser Arraytyp ARR hat einen Indextyp aus den natürlichen Zahlen und seine Elemente sind aus der Menge der ganzen Zahlen.

Beispiel für Tupel:
t1: TYPE = NAT;
t2: TYPE = REAL;
t3: TYPE = INT;
tupel: [t1, t2, t3];
Dies ist einTupel bestehend aus NAT,REAL und INT Typen.

Die Deklaration der Funktionstypen wird definiert als:
TA -> TR
(T1, T2, ... , Tn) -> TR

TA ist der Argument Typ
$(T1, T2, ..., Tn)$ sind die Argumenten Typen
TR ist der Ergebnis Typ

Ein Beispiel mit einem Argument und einem Ereignis Typ in einer Funktion ist:

t1: NAT -> NAT;

Beispiele mit Argumenten und einem Ereignis Typ in einer Funktion ist:

t1: TYPE = INT;
t2: TYPE = NAT;
t3: (t1, t2)-> NAT;

oder

t4:(REAL,INT,NAT,BOOLEAN)->NAT;

Mit dem Schlüsselwort FORMULA wird eine zu beweisende Formel deklariert.

Beispiel:

c: FORMULA FORALL(x:NAT,y:NAT):(x+y)^ 2 = x^ 2 + 2*x*y + y^ 2;

Durch das Schlüsselwort AXIOM wird ein Axiom definiert.

Beispiel:

a: AXIOM b(10) = 10;

4.2 Entscheidungsprozeduren

Der ProofNavigator [40] hat zurzeit CVC Lite 2.0 als automatische Entscheidungsprozedur eingebunden. Dieses Programm wird über die leicht veränderte Logik der höheren Ordnung angesprochen, sodass keine Übersetzung in die Logik der ersten Ordnung nötig ist. Die Entscheidungsprozedur wertet die Eingaben aus, um Gültigkeitsaussagen über die Eingangsdaten zu treffen. Die Ausgaben werden an den interaktiven Beweisassistenten zurückgesendet und in den interaktiven Beweisbaum eingebunden.

4.2.1 CVC Lite 2.0

CVC Lite 2.0 ist ein automatischer Beweiser für die SATisfiability Modulo Theories (SMT).
CVC Lite ist die überarbeitete Neuimplementierung des
„Cooperating Validity Checker“. Der Begriff *Lite* steht für einen schlankeren Aufbau als für das ursprüngliche CVC.
Der CVC Lite versteht eine Logik der höheren Ordnung. Außerdem werden interaktive Bibliotheksschnittstellen für C und C++, Beweis- und Modellgenerierungsfähigkeiten, Prädikatuntertypen und Quantoren unterstützt.

4.2.2 Beispiel

Hier wird die Funktionsweise des RISC ProofNavigators im Zusammenspiel mit der automatischen Entscheidungsprozedur CVCL anhand eines Beispiels näher erklärt.

Beweis einer Summenformel

Die Formel

$$\sum_{i=0}^{n} i = \frac{(n+1) * n}{2}$$

soll bewiesen werden. Dazu werden zwei Axiome

$$\sum_{i=0}^{0} i = 0$$

und

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i (n > 0)$$

zur eindeutigen Charakterisierung der Summenformel benötigt.

Deklaration

Die Eingaben der Deklarationen in den ProofNavigator lauten:

```
sum: NAT -> NAT;                    .... Summenfunktion von i
S1: AXIOM sum(0) = 0;               .... 1. Axiom
S2: AXIOM FORALL(n:NAT): n > 0 =>
sum(n) = n+sum(n-1);                .... 2. Axiom
S: FORMULA FORALL(n:NAT):
sum(n) = (n+1)*n/2;                 .... die zu beweisende Formel
```

Der ProofNavigator transformiert die Eingabe in die für CVCL verständlichen Kommandos.

```
sum__:(INT->INT);
ASSERT FORALL(x0_2__:INT): x0_2__ >= 0 => sum__(x0_2__) >= 0;
ASSERT sum__(0) = 0;
PUSH;
QUERY FORALL(n_2__:INT): n_2__ >= 0 => n_2__ > 0 => n_2__ >= 1;
POP;
ASSERT FORALL(n_3__:INT): n_3__ >= 0 => n_3__ >
0 => sum__(n_3__) = n_3__+sum__(n_3__-1);
PUSH;
QUERY 2 /= 0;
```

POP;

„PUSH“ ist das Signalwort um einen Sicherungspunkt zu setzen.
ÄSSERT FORALL(x0_2__:INT): x0_2__ >= 0 => sum__(x0_2__) >= 0“ fügt das erste Axiom in den logischen Kontext hinzu.
„ FORALL(n_2__:INT): n_2__ >= 0 => n_2__ > 0 => n_2__ >= 1“ prüft ob die Formel gültig ist.
„POP“ ist das Signalwort um zum letzten Sicherungspunkt zurückzukehren.

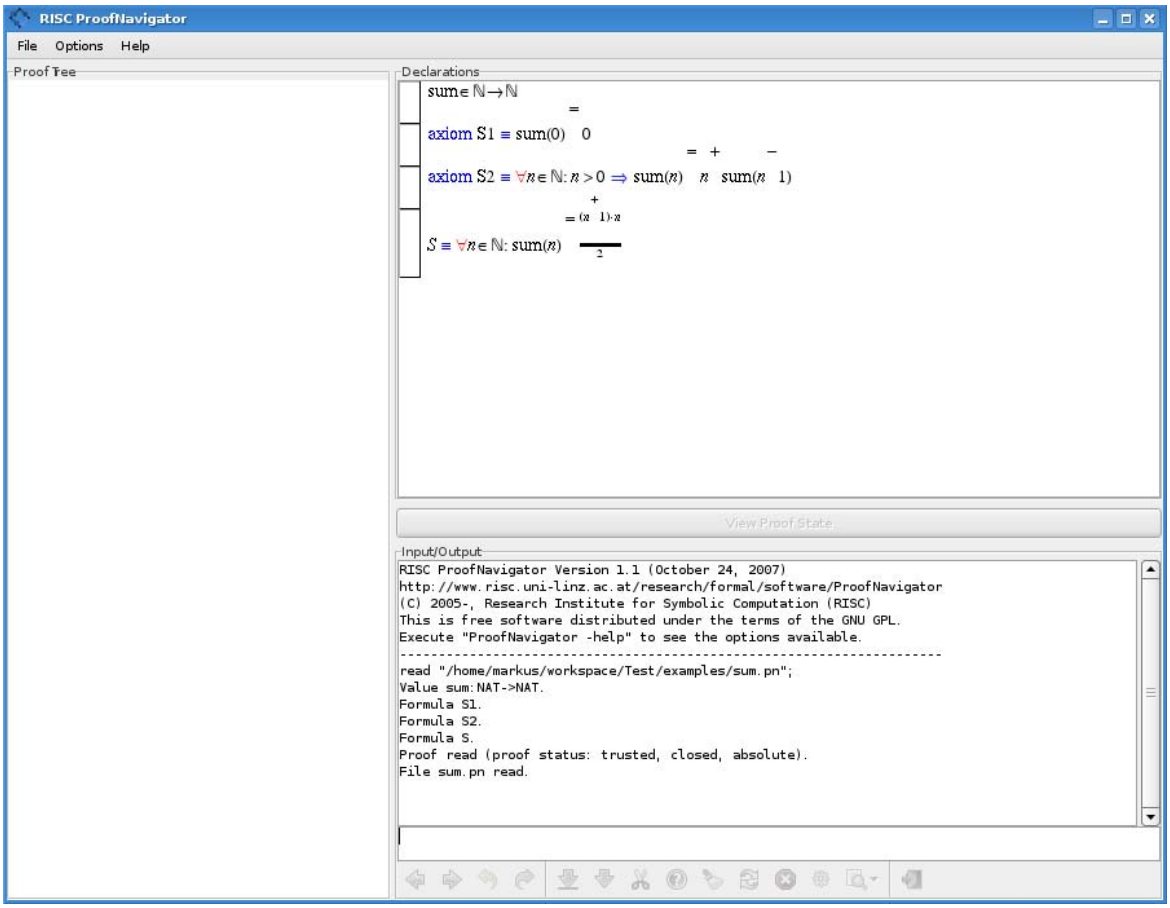

Abbildung 4.2: Deklaration

Nach der Eingabe der Deklaration wandelt der ProofNavigator die Daten in eine schönere Ausgabe (siehe Abbildung 4.2) um und schreibt sie zurück in das Deklarationsfenster.

Wenn die Maus über eine Formel im Deklarationsbereich positioniert wird, wird ein Auswahlmenü geöffnet. Durch die Auswahl von

- „proof S“ wird ein bereits durchgeführter Beweis der Summenformel angezeigt,
- „prove S“ wird das System in den Beweismodus versetzt und
- „formula S“ wird die gewählte Formel im Eingabefenster ausgegeben.

Induktion

Das System wird in den Beweismodus geschalten. Dann wird die markierte Formel mit der Maus ausgewählt, wo das Fenster (siehe Abbildung 4.3) sichtbar wird. Wenn „expand [] in lxe“ benutzt wird, wird die Definition der Eingabe [] in die Formel mit der Markierung „lxe“ kopiert.
Durch einen klick auf „induction [] in byu“ wird der Beweisschritt mittels Induktion gestartet. Wenn „flip lxe“ auf die Formel angewendet wird, wird die Formel durch die Negation der Formel ersetzt.

Wir wenden bei diesem Beweis „induction n in byu“ auf die Formel an.

Hier sind Auszüge der Kommunikation zwischen CVCL und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Induktion von n

```
sum__:(INT->INT);
_MASK0__:(((INT->INT))->BOOLEAN);
ASSERT _MASK0__(sum__);
PUSH;
PUSH;
_MASK1__:(((INT->INT))->BOOLEAN);
QUERY _MASK1__(sum__);
.
.
.
```

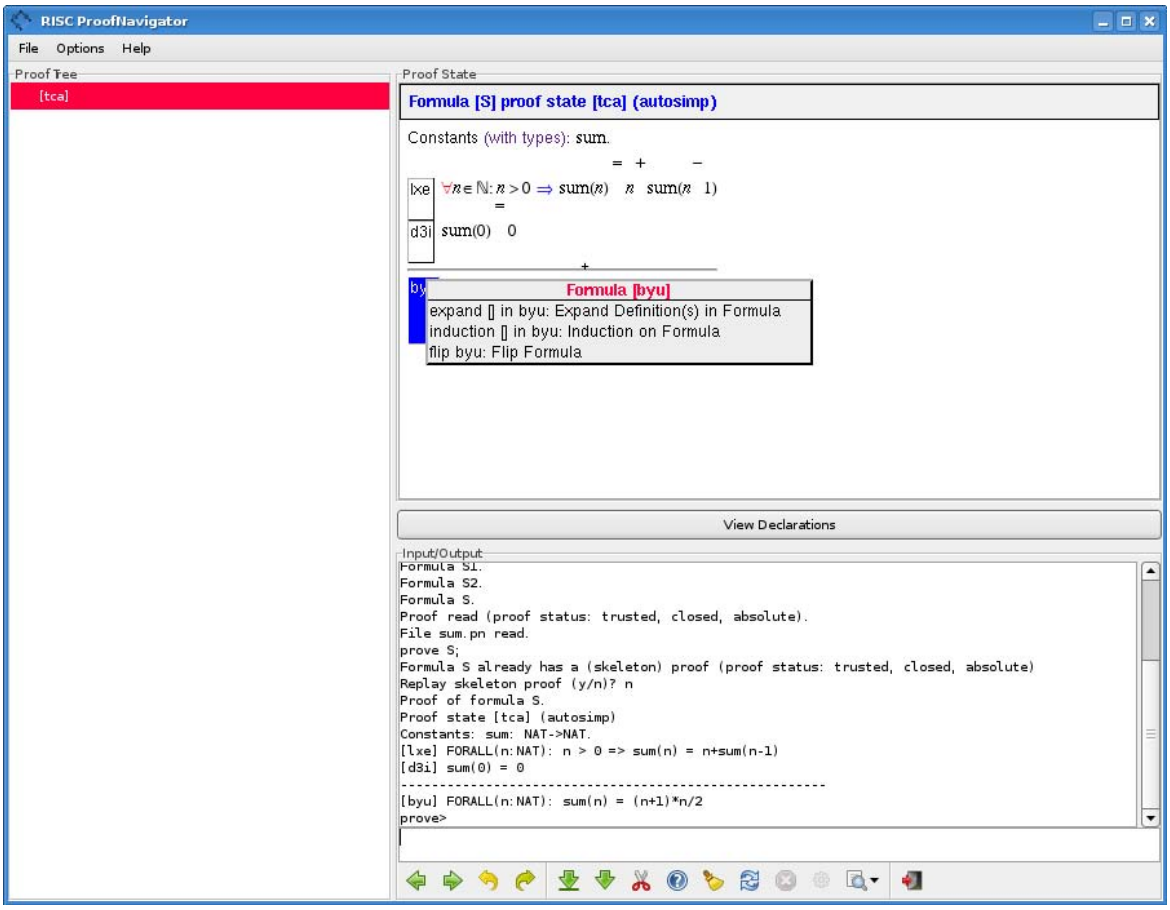

Abbildung 4.3: Induktion

„MASK0“ maskiert die Aussage „lxe“ als atomare Formel. Da CVCL die Logik der höheren Ordnung ,mit Funktionstypen als normalen Typen beherrscht, müssen Konstrukte wie „_MASK0__:(((INT->INT))->BOOLEAN)“ nicht vom ProofNavigator übersetzt werden und können direkt an CVCL gesendet werden.

Instanziierung

Wenn die Maus über die noch zu beweisende Formel bewegt wird, öffnet sich das Fenster (siehe Abbildung 4.4). Mit „auto lxe“ wird die Formel automatisch instanziiert.
Durch „instantiate [] in lxe“ kann mittels Instanziierung von $n_0 + 1$ der Beweis abgeschlossen werden.
Aus der Auswahl von „goal lxe“ wird die Formel durch die Negation der Formel ersetzt und wird zum einzigen Beweisziel.

In diesem Beweis wird entweder „auto lxe“ oder „instantiate $n_0 + 1$ in lxe“ verwendet um den Beweis abzuschließen.

Hier sind Auszüge der Kommunikation zwischen CVCL und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Instanziierung von n_0+1

```
sum__:(INT->INT);
_MASK0__:(((INT->INT))->BOOLEAN);
ASSERT _MASK0__(sum__);
n_0__:INT;
ASSERT n_0__ >= 0;
PUSH;
PUSH;
_MASK1__:(((INT->INT))->BOOLEAN);
QUERY _MASK1__(sum__);
.
.
.
```

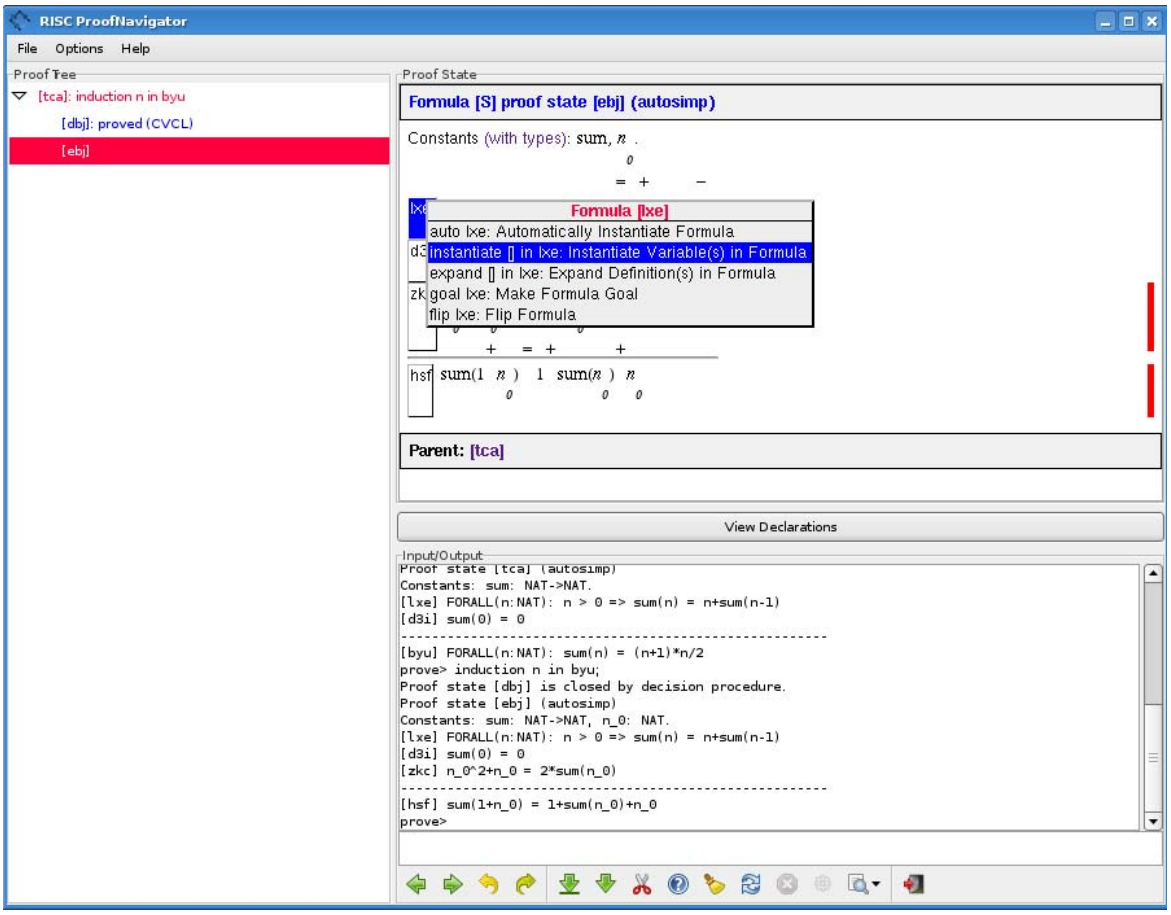

Abbildung 4.4: Instanziierung

Nach der Durchführung dieses Beweisschrittes ist der Beweis abgeschlossen (siehe Abbildung 4.5) und kann abgespeichert werden.

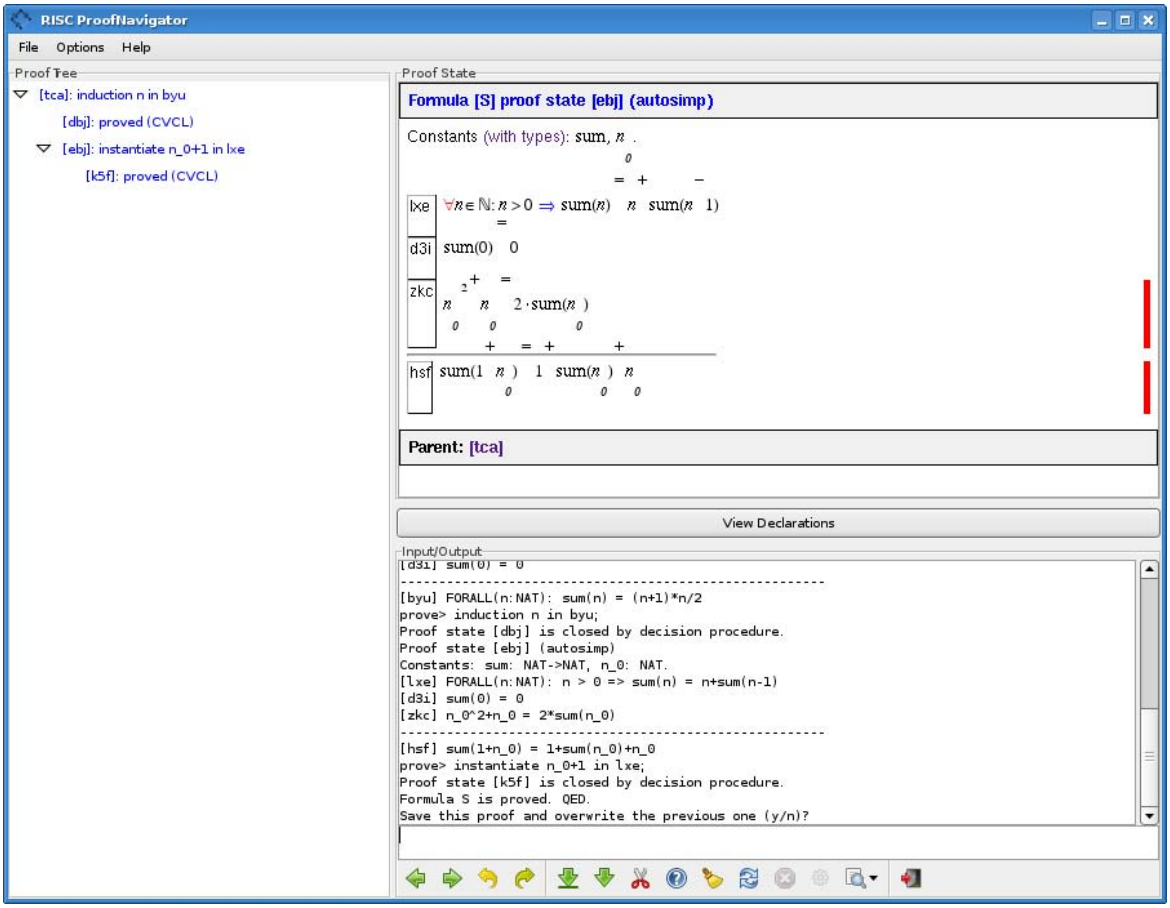

Abbildung 4.5: Abgeschlossener Beweis

4.2.3 CVC 3

In diesem automatischen Beweisassistent für die Logik der ersten Ordnung mit Quantoren, Funktionen und Prädikaten sind folgende Theorien eingebunden:

- Gleichheit über freie Funktionen und Prädikaten Symbole
- REAL und INT Arithmetik
- Bit Vektoren
- ARRAYS

- TUPLES
- RECORDS
- Benutzerdefinierte induktive Datentypen

CVC3 benutzt für das „Typ Checking“ die Regeln der Logik der ersten Ordnung mit Sorten. Die Regeln für das Typcheckingsystem sind:

- Jede Variable hat einen zugehörigen Typ.
- Jedes Konstanten-Symbol hat einen zugehörigen Typ.
- Variablen und Konstanten-Symbole dürfen keine Funktionstypen sein
- Jeder Funktionstyp hat einen oder mehrere zugehörige Funktionstypen
- Der Term Typ besteht nur aus Variablen- und Konstanten- Symbolen.
- Eine Funktion $f\ :\ (t1 : T1,\ t2 : T2,\ ...\ tn : Tn) \rightarrow T$ ist nur dann gültig, wenn jedes t_i einen Typ T_i besitzt.
- Im Arraytyp (ARRAY T_1 OF $T_2, T_1) \rightarrow T_2$ dürfen in T_1 und T_2 keine Funktionen oder Prädikate vorkommen.

Terme und Formeln

Zusätzlich zu den Typen besitzt CVC3 Ausdücke für Terme und Formeln. Diese Terme werden aufgebaut aus:

- getypen Variablen
- vordefinierten Operatoren
- frei vorkommenden Funktionssymbolen
- Quantoren
- if-then-else Operatoren
- Lambdaabstraktionen
- lokalen Symboldeklarationen

Beispiele:

Konstante:
$a, b, c : INT;$
$x, y, z : REAL;$

Variablen:
$P, Q; BOOLEAN;$

Funktionen:
$f1 : REAL -> REAL;$
$f2 : (REAL, INT) -> REAL;$
$f3 : [INT, REAL] -> BOOLEAN;$

Prädikaten:
$p : (INT, REAL) -> BOOLEAN;$

Diese Deklarationen müssen eindeutig sein.

Beispiele für die Definition von neuen freien Symbolen:
$c : INT;$
$i : INT = 5 + 3 * c;$
$j : REALv = 3/4;$
$t : [REAL, INT] = (2/3, -4);$
$r : [\#key : INT, value : REAL\#] = (\#key := 4, value := (c + 1)/2\#);$
$f : BOOLEAN = FORALL\,(x : INT) : x <= 0\,OR\,x > c;$

Logische Symbole

Die logischen Ausdrücke in CVC3 repräsentieren die Vergleichsoperationen wie Gleichheit, Ungleichheit, kleiner als, kleiner gleich als, größer als und größer gleich

als, die logischen Konstanten wahr und falsch und die logischen Operationen wie Negation, Konjunktion, Disjunktion, Implikation, Äquivalenz und Antivalenz mit Quantoren.
= und /=
TRUE und FALSE
NOT, AND, OR, XOR, =>, <=>

Beispiele:

$A, B : TYPE;$
$quant : BOOLEAN = FORALL(x, y : A, i, j, k : B) : i = j \, AND \, i / = k \, => EXISTS \, (z : A) : x / = z \, OR \, z / = y;$

$A : TYPE;$
$i, j : INT;$
$f : BOOLEAN = i = j \, AND \, FORALL \, (i, j : A) : i = j \, OR \, i / = j;$

Schlüsselwörter

Kommandos in CVC 3 sind:

- ASSERT
- QUERY
- PUSH
- POP
- TRANSFORM

ASSERT fügt eine Formeln in den logischen Kontext hinzu.
QUERY prüft ob die Formel gültig ist.
PUSH setzt einen Sicherungspunkt.
POP geht zum letzten Sicherungspunkt zurück.

TRANSFORM vereinfacht Ausdrücke.

Ein Beispiel wie eine zu überprüfende Formel eingegeben wird:
$PUSH;$
$QUERY\ <formula>;$
$POP;$

4.2.4 CVC Lite 2.0 versus CVC 3

CVC Lite wurde im Kapitel 6 Seite 9 beschrieben. CVC 3 ist eine Weiterentwicklung von CVCL, aber eingeschränkt auf eine Logik der ersten Ordnung, da Gültigkeitsaussagen nur in der Logik der ersten Ordnung getroffen werden können.
CVC Lite versteht im Gegensatz zu CVC 3 Sprachelemente der Logik der höheren Ordnung, daher kann man, obwohl CVC Lite und CVC 3 eine ähnliche Syntax benutzen, den Übersetzer von CVC Lite nicht für CVC 3 benutzten, um die Anweisungen des ProofNavigators richtig auszuwerten.
So kann CVC Lite geschachtelte Funktions- oder Prädikatentypen interpretieren was aber CVC 3 nicht möglich ist. „Boolean“ Werte in den Argumenten von Funktionen und Prädikaten wurden durch das Sprachkonstrukt „Bitvector(1)“ ersetzt. Logischen Konstanten in solchen Bitvektoren wurden von wahr und falsch umgestellt auf die Konstanten „0bin0“ und „0bin1“.

4.3 Übersetzer

Da der ProofNavigator die Logik der höheren Ordnung versteht, aber die Entscheidungsprozedur CVC 3 nur die Logik der ersten Ordnung, wird eine Übesetzungsfunktion benötigt, um Aussagen im ProofNavigator in entsprechende Aussagen von CVC 3 zu übersetzen. Die Grundlagen dieser Übersetzung wurden in Kapitel 3 erläutert. Außer diesem Problem gibt es aber noch sprachspezifische Eigenheiten der beiden Programme, die ebenfalls übersetzt werden müssen. Die Behandlung dieser Eigenheiten wird in diesem Abschnitt beschrieben.

4.3.1 Built-In Typen NAT

Es gibt keinen Built-In Typen NAT in CVC 3, daher muss
$sum: NAT->NAT;$
übersetzt werden in:
$sum: INT->INT;$
$FORALL(x:INT): x>=0 => sum(x)>=0;$

4.3.2 Geschachtelte Funktionstypen

In CVC 3 sind keine geschachtelten Funktionstypen erlaubt, daher muss
$sum: INT->INT;$
$m1: (INT->INT)->BOOLEAN;$
übersetzt werden in:
$sum: INT->INT;$
$type1: TYPE;$
$m1: (type1)->BOOLEAN;$

Ein weiteres Beispiel für geschachtelte Funktionstypen ist,
$s: INT->INT;$
$s1: (INT->INT)->INT;$
$s2: ((INT->INT)->INT)->INT;$

das übersetzt wird in:
$type0 : TYPE;$
$sum : INT- > INT;$
$type1 : TYPE;$
$sum1 : type0- > INT;$
$type2 : TYPE;$
$sum2 : type1- > INT;$

4.3.3 FORALL Anweisung

Bei den natürlichen Zahlen muss auch die $FORALL$ Anweisung übersetzt werden, sodass kein Sprachelement der Logik der höheren Ordnung auftritt.

Beispiel:
$s : NAT- > NAT;$
$s1 : (NAT- > NAT)- > NAT;$
$s2 : ((NAT- > NAT)- > NAT)- > NAT;$
wird übersetzt in:
$type0 : TYPE;$
$s : INT- > INT;$
$ASSERT\ FORALL(x : INT) : x >= 0 => s(x) >= 0;$
$type1 : TYPE;$
$s1 : type0- > INT;$
$ASSERT\ FORALL(x : (type0)) : type0 = TYPE => s1(x) >= 0;$
$type2 : TYPE;$
$sum2 : type1- > INT;$
$ASSERT\ FORALL(x : (type1)) : type0 = TYPE => s2(x) >= 0;$

4.3.4 Indirekt geschachtelte Funktionstypen

Es müssen auch indirekt geschachtelte Funktionstypen von der Logik der höheren Ordnung in die Logik der ersten Ordnung übersetzt werden.

Beispiel:
$l1: ([INT, ARRAY\ INT\ OF\ ELEM] -> INT);$
$l2: l1 -> BOOLEAN;$
wird übersetzt in:
$l1: ([INT, ARRAY\ INT\ OF\ ELEM] -> INT);$
$type1: TYPE;$
$l2: type1 -> BOOLEAN;$

4.3.5 Sprachelemente

Der Typ $BOOLEAN$ wird zu $BITVECTOR(1)$ als Argument in einer Funktion.

Beispiel:
$s: (BOOLEAN, BOOLEAN) -> BOOLEAN;$
wird zu:
$s: (BITVECTOR(1), BITVECTOR(1)) -> BOOLEAN;$

Operationen:
Die $BITVECTOR$ Konstanten $TRUE$ und $FALSE$ werden durch die Konstanten $0bin0$ und $0bin1$ ersetzt.

4.4 Einschränkung

Die Typdeklarationen für einen Beweis müssen im ProofNavigator schrittweise aufgebaut werden, da sonst CVC 3 die Eingaben falsch interpretiert.

Beispiel:
$s: (INT->INT)->INT;$

Solche Deklarationen sollen schrittweise eingegeben werden:
$s: INT->INT;$
$s1: (INT->INT)->INT;$

CVC 3 kann keine geschachtelten Funktionen verarbeiten. Einen Funktion muss als Typ abgespeichert werden, so dass der Übersetzter die Logik der Höheren Ordnung korrekt in eine Logik der Ersten Ordnung umwandeln kann.

5 Beispiele

In diesem Kapitel wird die Funktionsweise des RISC ProofNavigators im Zusammenspiel mit der automatischen Entscheidungsprozedur anhand von 2 Beispielen näher erklärt. Im ersten Beispiel wird eine Summenformel mittels einer Induktion bewiesen. Im zweiten Beispiel wird eine polynomiale Gleichheit gezeigt.

5.1 Beweis einer Summenformel

Die Formel

$$\sum_{i=0}^{n} i = \frac{(n+1) * n}{2}$$

soll bewiesen werden. Dazu werden zwei Axiome

$$\sum_{i=0}^{0} i = 0$$

und

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i (n > 0)$$

zur eindeutigen Charakterisierung der Summenformel benötigt.

5.1.1 Deklaration

Die Eingaben der Deklarationen in den ProofNavigator lauten:

```
sum: NAT -> NAT;                    .... Definitionsbereich von i
S1: AXIOM sum(0) = 0;               .... 1. Axiom
S2: AXIOM FORALL(n:NAT): n > 0 =>
sum(n) = n+sum(n-1);                .... 2. Axiom
S: FORMULA FORALL(n:NAT):
sum(n) = (n+1)*n/2;                 .... die zu beweisende Formel
```

Der Übersetzer transformiert die Eingangsdaten des ProofNavigator in die für CVC 3 verständlichen Kommandos:

```
type0: TYPE;
sum:INT->INT;
ASSERT FORALL(x:INT): x >= 0 => sum(x) >= 0;
ASSERT sum(0) = 0;
PUSH;
QUERY FORALL(n:INT): n >= 0 => n >= 1;
VALID
POP;
ASSERT FORALL(n:INT): n >= 0 => n > 0 => sum(n) = n+sum(n-1);
PUSH;
QUERY 2 /= 0;
POP;
```

„sum:INT->INT“ wird von NAT zu INT da CVC 3 keinen Built-In Typen NAT besitzt.
„PUSH“ ist das Signalwort zum setzen eines Sicherungspunktes.
„ASSERT sum(0) = 0“ fügt die Formel in den logischen Kontext hinzu.
„QUERY FORALL(x:INT): x >= 0 => sum(x) >= 0“ prüft ob die Formel gültig ist.
„VALID“ gibt CVC 3 an den ProofNavigator zurück um zu bestätigen, dass die Formel gültig ist.

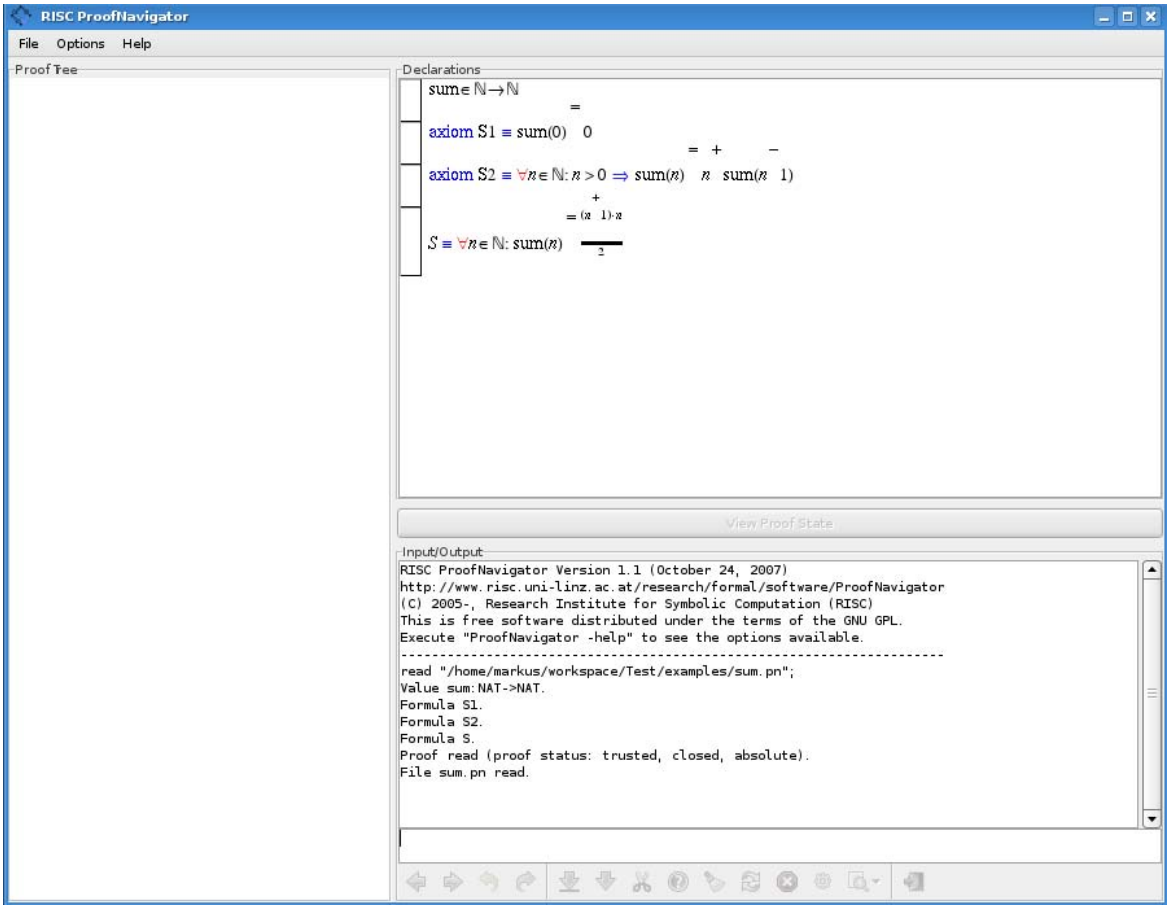

Abbildung 5.1: Deklaration

Nach der Eingabe der Deklaration (siehe Abbildung 5.1) wandelt der ProofNavigator die Daten in eine schönere Ausgabe um und schreibt sie zurück in das Deklarationsfenster.

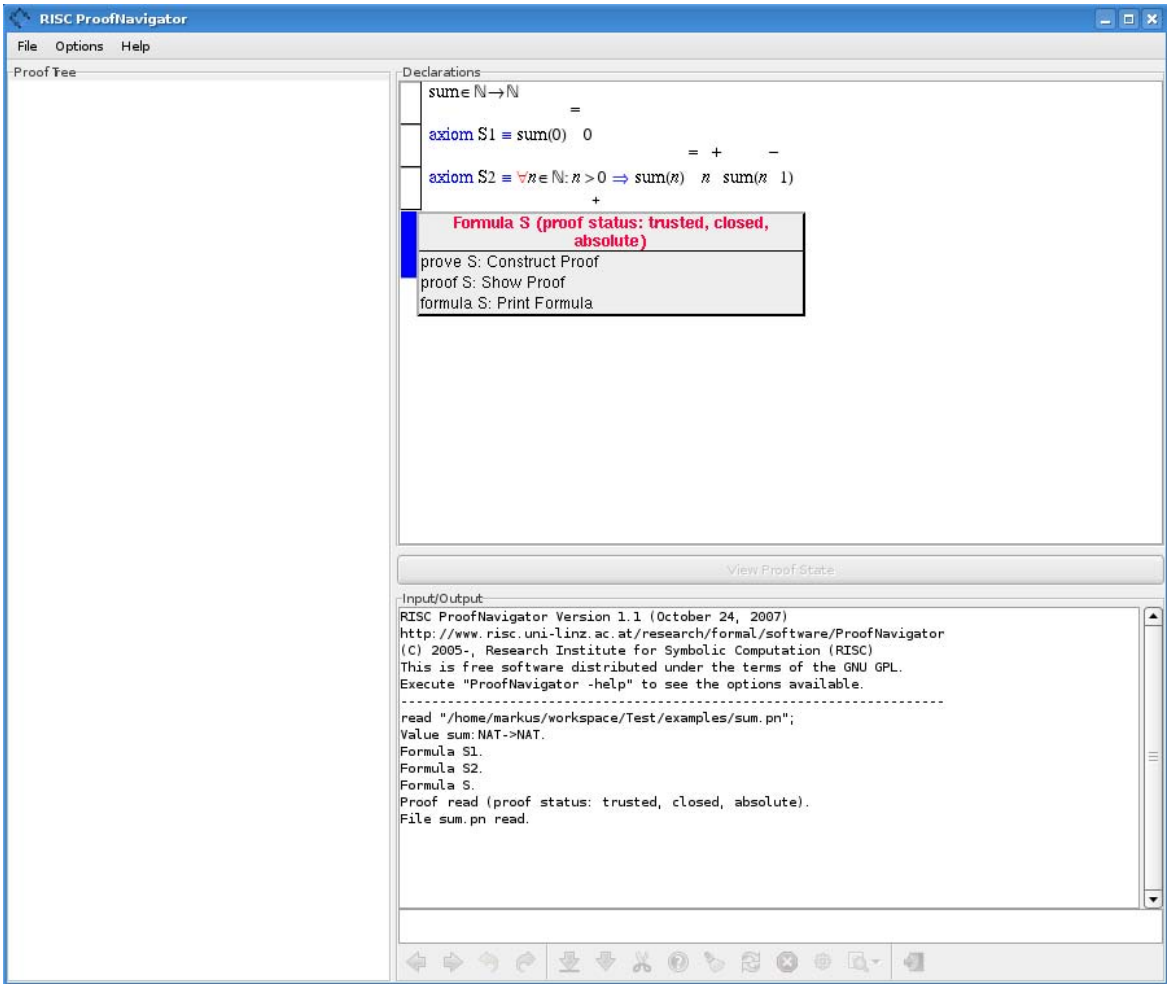

Abbildung 5.2: Auswahl

Wenn die Maus über eine Formel im Deklarationsbereich positioniert wird, wird ein Auswahlmenü (siehe Abbildung 5.2) geöffnet. Durch die Auswahl von

- „proof S“ wird ein bereits durchgeführter Beweis der Summenformel angezeigt,
- „prove S“ wird das System in den Beweismodus versetzt und
- „formula S“ wird die gewählte Formel im Eingabefenster ausgegeben.

5.1.2 Induktion

Das System wird in den Beweismodus geschalten. Dann wird die markierte Formel mit der Maus ausgewählt, wo das Fenster (siehe Abbildung 5.3) sichtbar wird. Wenn „expand [] in lxe“ benutzt wird, wird die Definition der Eingabe [] in die Formel kopiert.
Durch einen klick auf „induction [] in byu“ wird der Beweisschritt mittels Induktion gestartet. Wenn „flip lxe“ auf die Formel angewendet wird, wird die Formel durch die Negation der Formel ersetzt.

Wir wenden bei diesem Beweis „induction n in byu“ auf die Formel an.

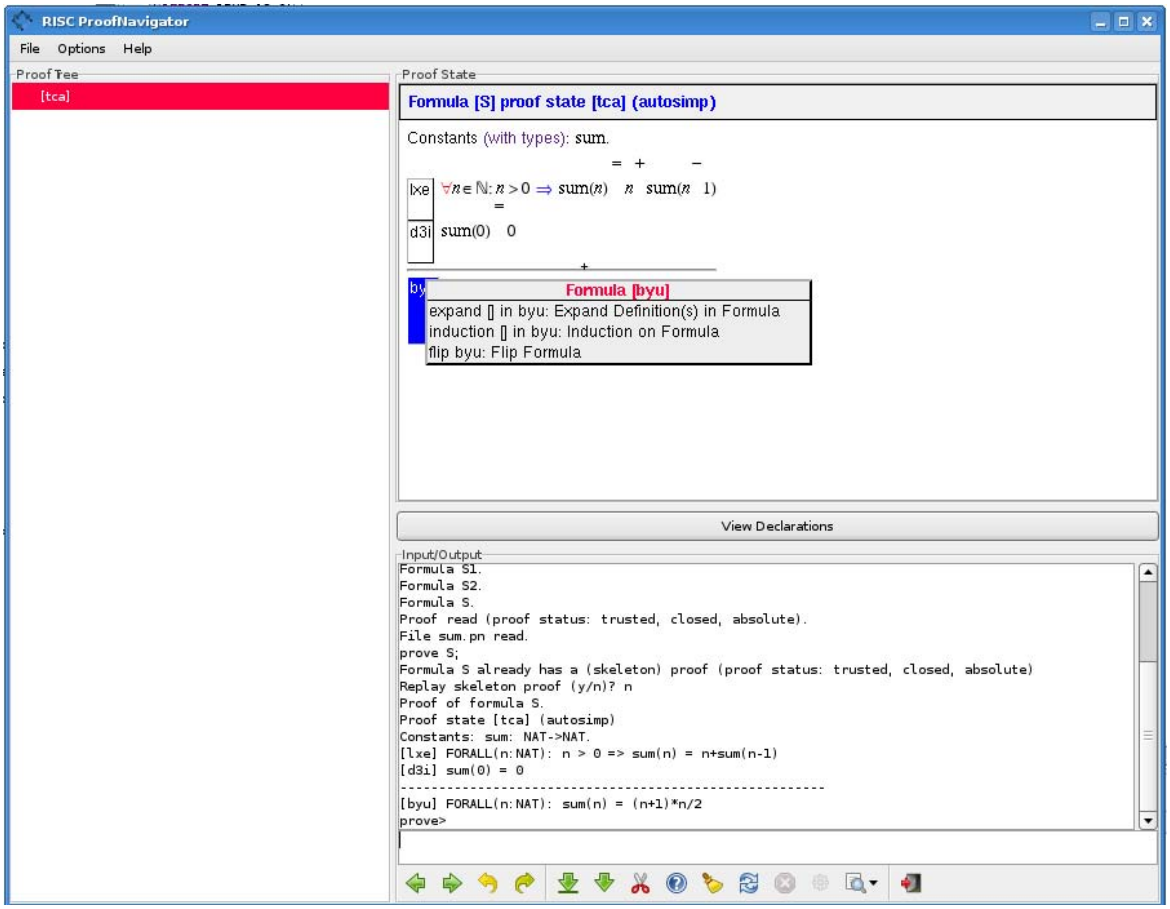

Abbildung 5.3: Induktion

Hier sind Auszüge der Kommunikation zwischen CVC3 und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Induktion von n

```
type0__: TYPE;
sum__:(INT->INT);
_MASK0__:(((type0__))->BOOLEAN);
ASSERT _MASK0__(temp0__);
PUSH;
PUSH;
_MASK1__:(((type0__))->BOOLEAN);
QUERY _MASK1__(temp0__);
.
.
.
```

Der Typ INT->INT von sum wird in MASK0 durch type0 ersetzt, so dass keine geschachtelten Funktionen und somit keine Logik der höheren Ordnung entsteht.

5.1.3 Instanziierung

Wenn die Maus über die noch zu beweisende Formel bewegt wird, öffnet sich das Fenster (siehe Abbildung 5.4). Mit „auto lxe“ wird die Formel automatisch instaziiert.
Durch „instantiate [] in lxe“ kann mittels Instanziierung von $n_0 + 1$ der Beweis abgeschlossen werden.
Aus der Auswahl von „goal lxe“ wird die Formel durch die Negation der Formel ersetzt und wird zum einzigen Beweisziel.

In diesem Beweis wird entweder „auto lxe“ oder „instantiate $n_0 + 1$ in lxe“ verwendet um den Beweis abzuschließen.

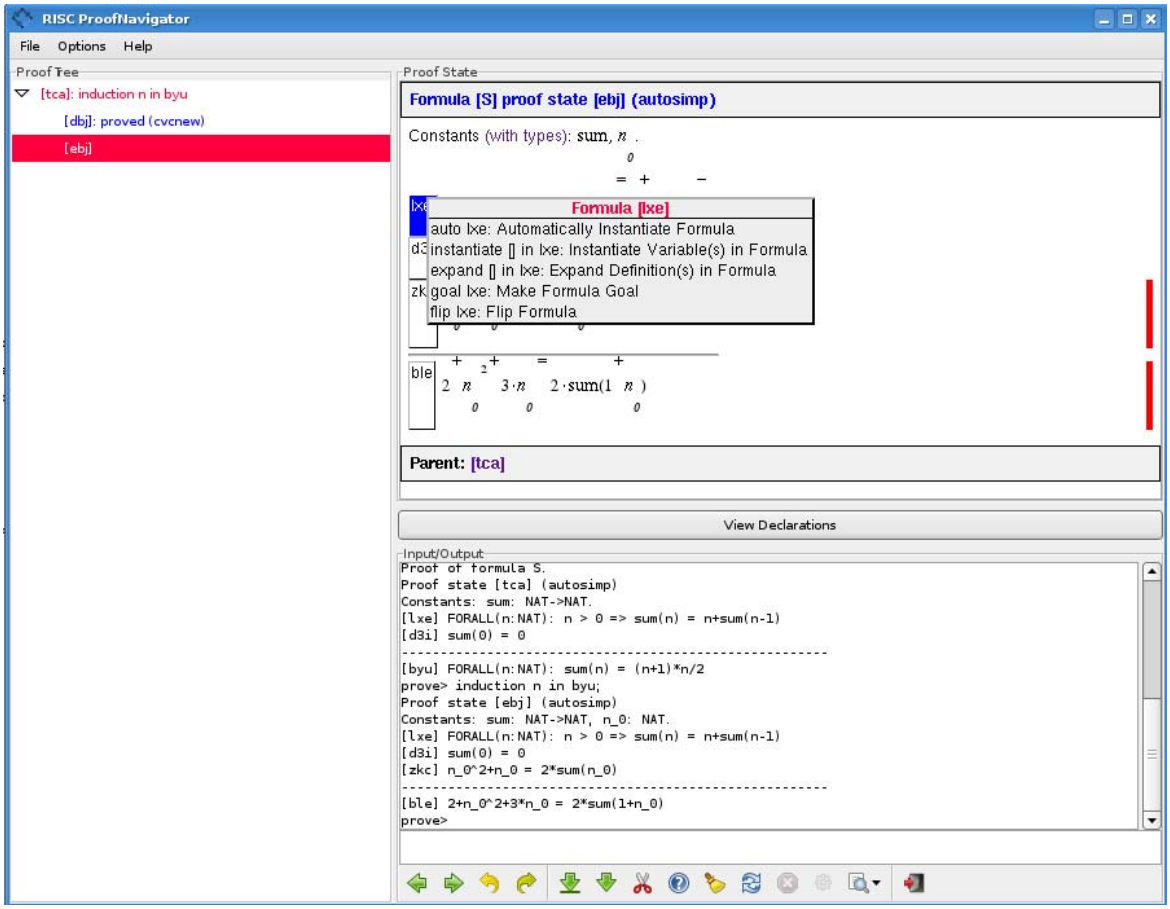

Abbildung 5.4: Instanziierung

Hier sind Auszüge der Kommunikation zwischen CVC3 und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Instanziierung von n_0+1

```
sum__:(INT->INT);
_MASK0__:(((type0__))->BOOLEAN);
ASSERT _MASK0__(temp0__);
n_0__:INT;
ASSERT n_0__ >= 0;
PUSH;
PUSH;
_MASK1__:(((type0__))->BOOLEAN);
QUERY _MASK1__(temp0__);
.
.
.
```

Nach der Durchführung diese Beweisschrittes ist der Beweis abgeschlossen (siehe Abbildung 5.5) und kann abgespeichert werden.

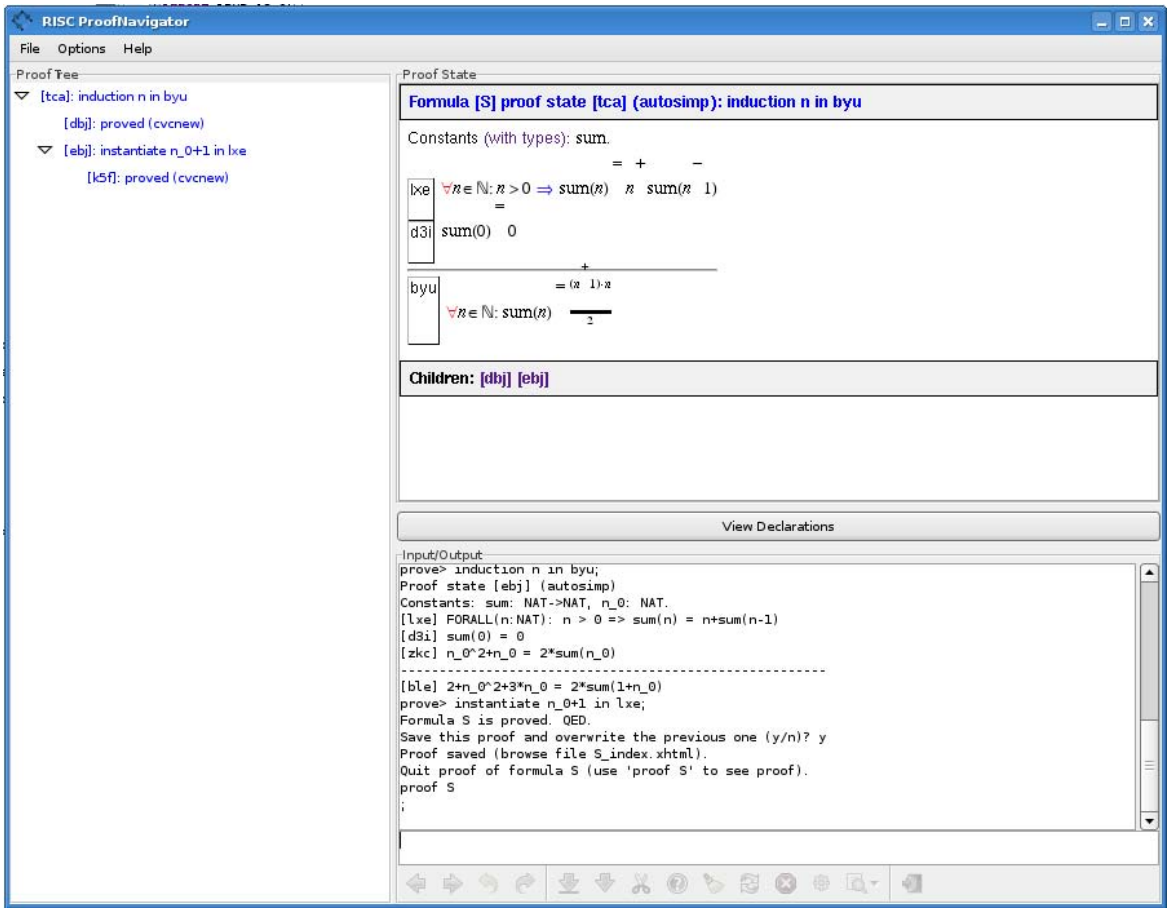

Abbildung 5.5: Abgeschlossener Beweis

5.2 Beweis einer polynomialen Gleichheit

Die Gleichheit

$$(x+y)^3 = x^3 + 3 * x^2 * y + 3 * x * y^2 + y^3$$

soll bewiesen werden.
Dazu werden zwei Axiome

$$x^3 = x * x * x$$

und

$$(x+1)^3 = x^3 + 3 * x^2 + 3 * x + 1$$

benötigt.

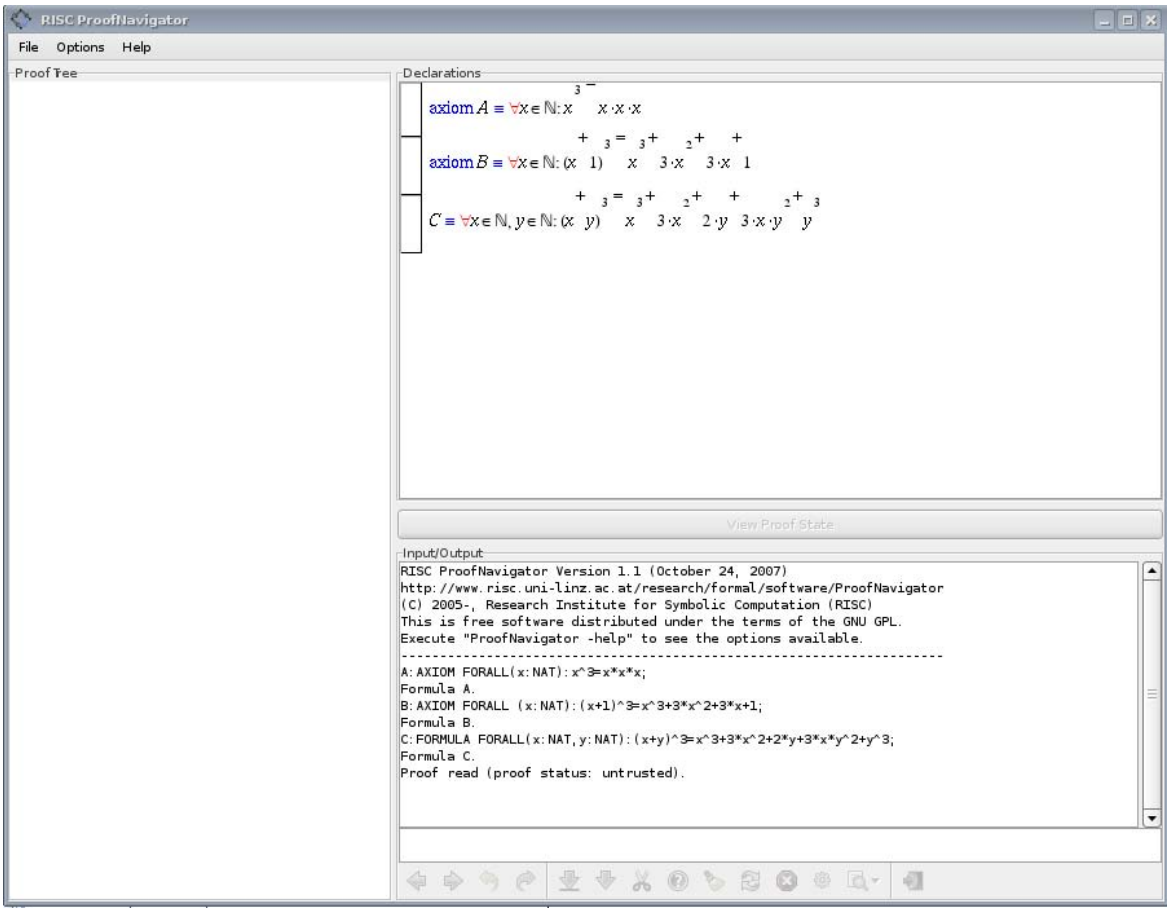

Abbildung 5.6: Eingabe

5.2.1 Deklaration

Die Eingaben der Deklarationen in den ProofNavigator lauten:

A:AXIOM FORALL(x:NAT):x^ 3=x*x*x;.. 1. Axiom
B:AXIOM FORALL(x:NAT):
(x+1)^ 3=x^ 3+3*x^ 2+3*x+1; .... 2. Axiom
C:FORMULA FORALL(x:NAT,y:NAT):
(x+y)^ 3=x^ 3+3*x^ 2*y+3*x*y^ 2+y^ 3; .... die zu beweisende Formel

Der Übersetzer transformiert die Eingangsdaten des ProofNavigator in die für CVC 3 verständlichen Kommandos:

ASSERT FORALL(x_1__:INT): x_1__ >= 0 => x_1__^ 3 = x_1__*x_1__*x_1__;
ASSERT FORALL(x_3__:INT): x_3__ >= 0 => (x_3__+1)^ 3
= x_3__^ 3+3*x_3__^ 2+3*x_3__+1;
PUSH;
PUSH;
_MASK0__:BOOLEAN;
QUERY _MASK0__;
POP;
POP;
PUSH;
.
.
.

„PUSH“ ist das Signalwort um einen Sicherungspunkt zu setzen.
„ASSERT FORALL(x_1__:INT): x_1__ >= 0 => x_1__^ 3 = x_1__*x_1__*x_1__“fügt das erste Axiom in den logischen Kontext hinzu.
„_MASK0__:BOOLEAN“ maskiert die Quantoren Formel.

Nach der Eingabe der Deklaration (siehe Abbildung 5.6) wandelt der ProofNavigator die Daten in eine schönere Ausgabe um und schreibt sie zurück in das Deklarationsfenster.

5.2.2 Induktion über x

Das System wird in den Beweismodus geschalten. Abschließend wird die markierte Formel mit der Maus ausgewählt, wodurch das Fenster (siehe Abbildung 5.7) sichtbar wird. Wenn „expand [] in 1d5“ benutzt wird, wird die Definition der Eingabe [] in die Formel kopiert.
Durch einen klick auf „induction [] in 1d5“ wird der Beweisschritt mittels Induktion gestartet. Wenn „flip 1d5“ auf die Formel angewendet wird, wird die Formel durch die Negation der Formel ersetzt.

Wir wenden bei diesem Beweis „induction n in 1d5“ auf die Formel an.

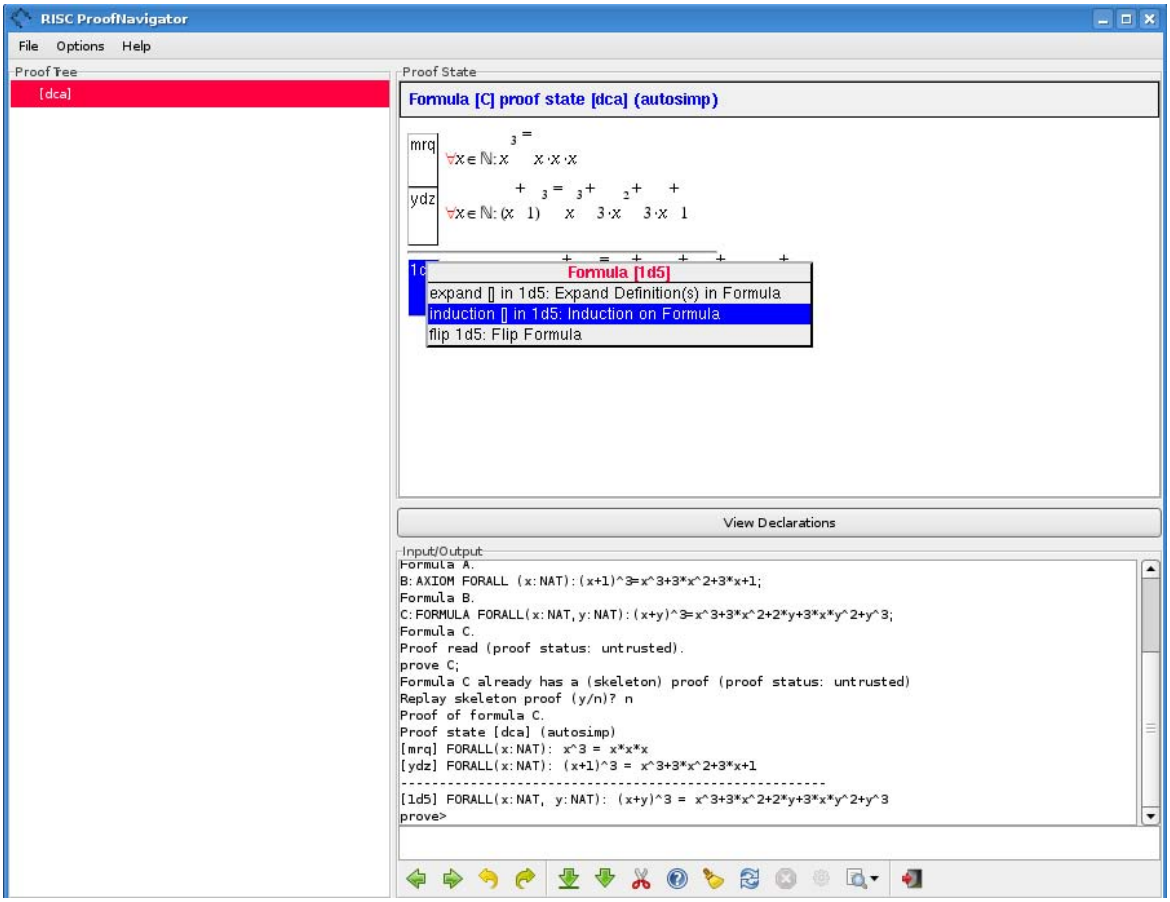

Abbildung 5.7: Induktion nach x

Hier sind Auszüge der Kommunikation zwischen CVC3 und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Induktion über x

```
x_0__:INT;
ASSERT x_0__ >= 0;
PUSH;
PUSH;
_MASK0__:BOOLEAN;
QUERY _MASK0__;
...
PUSH;
PUSH;
QUERY _MASK3__(x_0__);
POP;
POP;
PUSH;
TRANSFORM _MASK3__(x_0__);
POP;
POP;
POP;
```

5.2.3 1. Induktion über y

Die Formel (siehe Abbildung 5.8) wird mit der Maus ausgewählt und wir wenden bei diesem Beweis erneut „induction y in smc“ auf die Formel an. Der Unterschied zum vorangegangenen Beispiel ist, dass dieses Mal die Induktion über y bewiesen wird.

Hier sind nur Auszüge der Kommunikation zwischen CVC3 und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Induktion nach y

```
y_0__:INT;
ASSERT y_0__ >= 0;
PUSH;
PUSH;
```

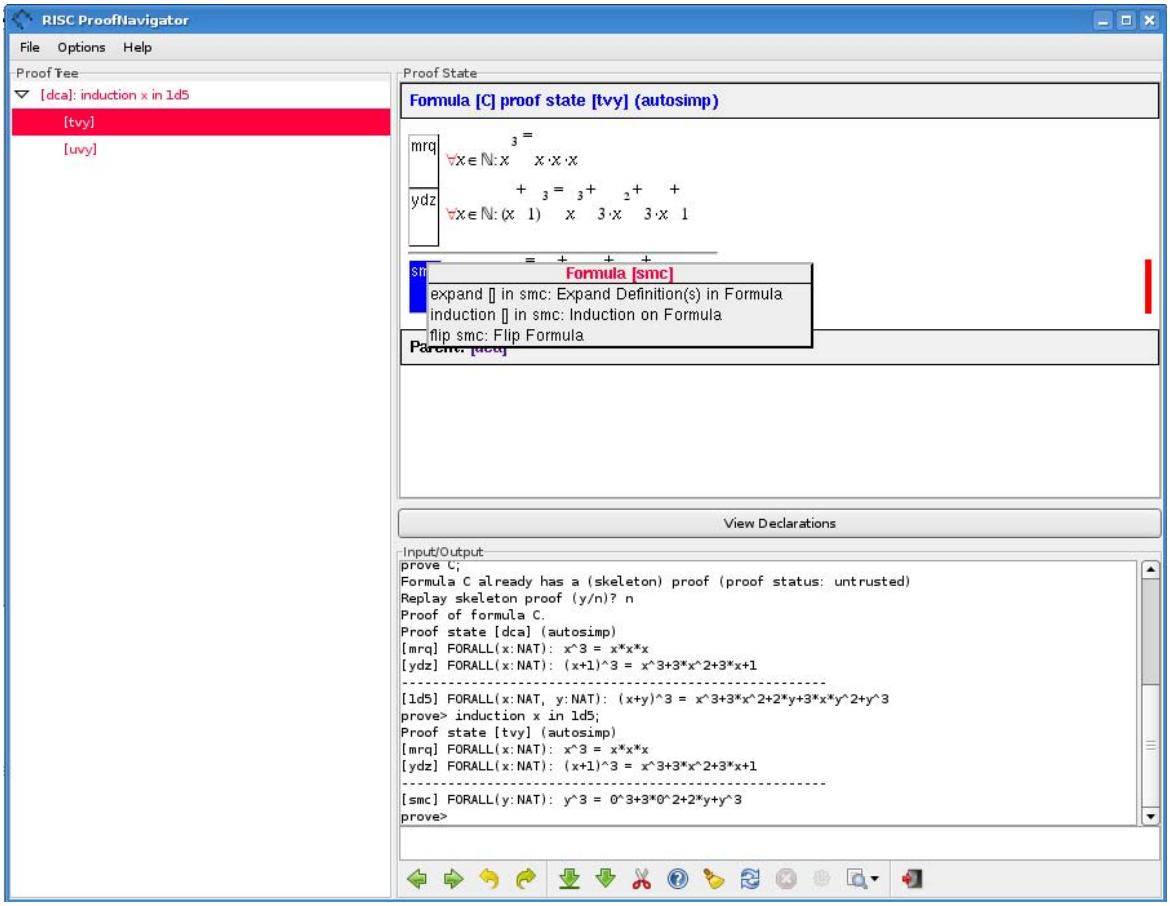

Abbildung 5.8: Induktion nach y

```
_MASK0__:BOOLEAN;
QUERY _MASK0__;
POP;
POP;
PUSH;
. . . QUERY (y_0__+1)^ 3 = 0^ 3+3*0^ 2*(y_0__+1)+(y_0__+1)^ 3;
POP;
POP;
PUSH;
ASSERT _MASK0__;
ASSERT _MASK1__;
ASSERT TRUE;
ASSERT FALSE;
PUSH;
QUERY FALSE;
POP;
POP;
```

5.2.4 2. Induktion über y

Bei der 2. Induktion wird der vorangehende Schritt wiederholt, also wird erneut eine Induktion über y auf die Formel (siehe Abbildung 5.9) angewendet.

Hier sind Auszüge der Kommunikation zwischen CVC3 und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Induktion über y

```
x_0__:INT;
ASSERT x_0__ >= 0;
y_0__:INT;
ASSERT y_0__ >= 0;
PUSH;
PUSH;
_MASK0__:BOOLEAN;
QUERY _MASK0__;
```

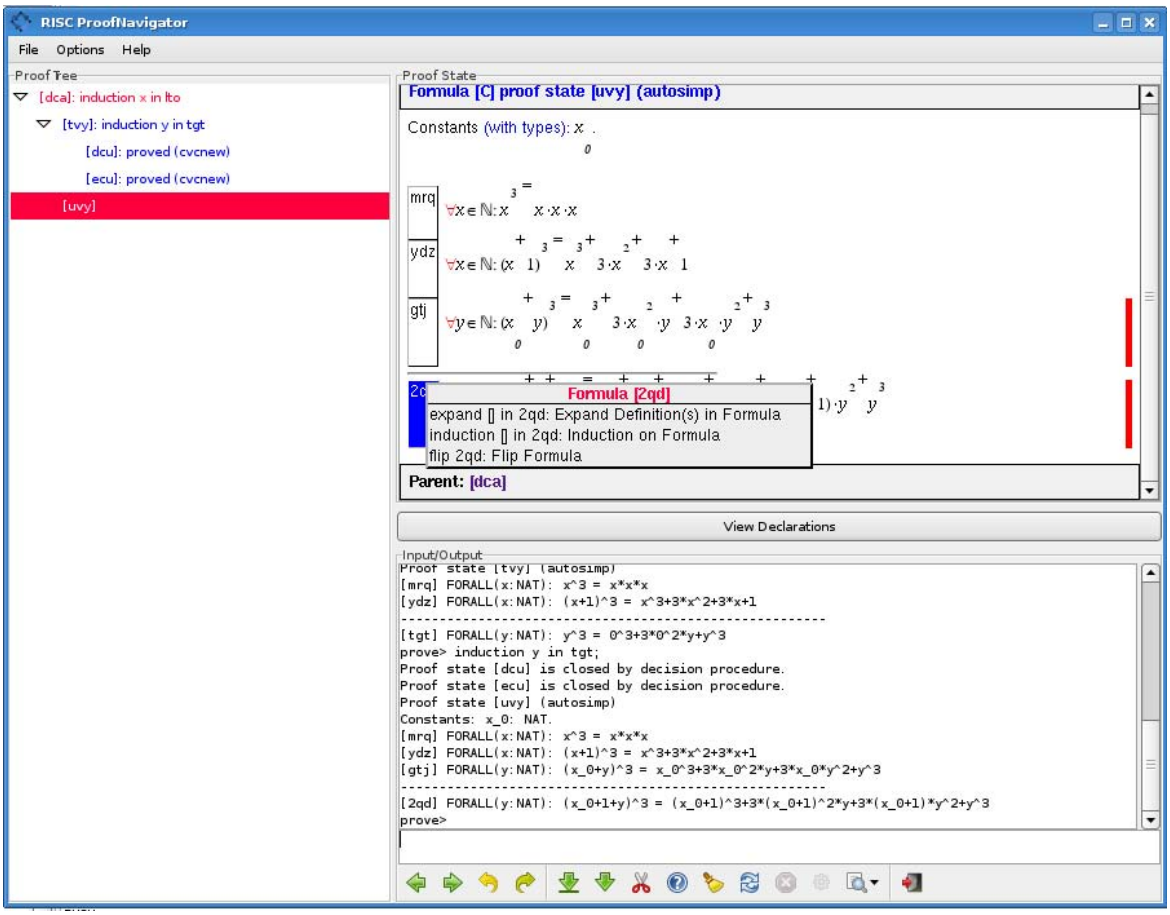

Abbildung 5.9: Induktion nach y

POP;
POP;
PUSH;
. . . TRANSFORM (1+y_0__+x_0__)^ 3 = (1+x_0__)^ 3+3*(1+x_0__)^ 2*y_0__+y_0__^ 3+3*y_0__^ 2*x_0__+3*y_0__^ 2;
POP;
POP;
PUSH;
ASSERT (1+y_0__+x_0__)^ 3 = (1+x_0__)^ 3+3*(1+x_0__)^ 2*y_0__+y_0__^ 3+3*y_0__^ 2*x_0__+3*y_0__^ 2;
PUSH;
PUSH;
QUERY (2+y_0__+x_0__)^ 3 = (1+y_0__)^ 3+3*(1+y_0__)^ 2*x_0__+3*(1+y_0__)^ 2+(1+x_0__)^ 3+3*(1+x_0__)^ 2*y_0__+3*(1+x_0__)^ 2;
POP;
POP;
PUSH;
TRANSFORM (2+y_0__+x_0__)^ 3 = (1+y_0__)^ 3+3*(1+y_0__)^ 2*x_0__+3*(1+y_0__)^ 2+(1+x_0__)^ 3+3*(1+x_0__)^ 2*y_0__+3*(1+x_0__)^ 2;
POP;
POP;
POP;

5.2.5 Autoinstanziierung

Wenn die Maus über die noch zu beweisende Formel bewegt wird, öffnet sich das Fenster (siehe Abbildung 5.10). Mit „auto mrq“ wird die Formel automatisch instaziiert.
Durch „instantiate [] in mrq“ kann mittels Instanziierung von $n_0 + 1$ der Beweis abgeschlossen werden.
Aus der Auswahl von „goal mrq“ wird die Formel durch die Negation der Formel ersetzt und zum einzigen Beweisziel.

In diesem Beweis wird „auto mrq“ verwendet um den Beweis abzuschließen.

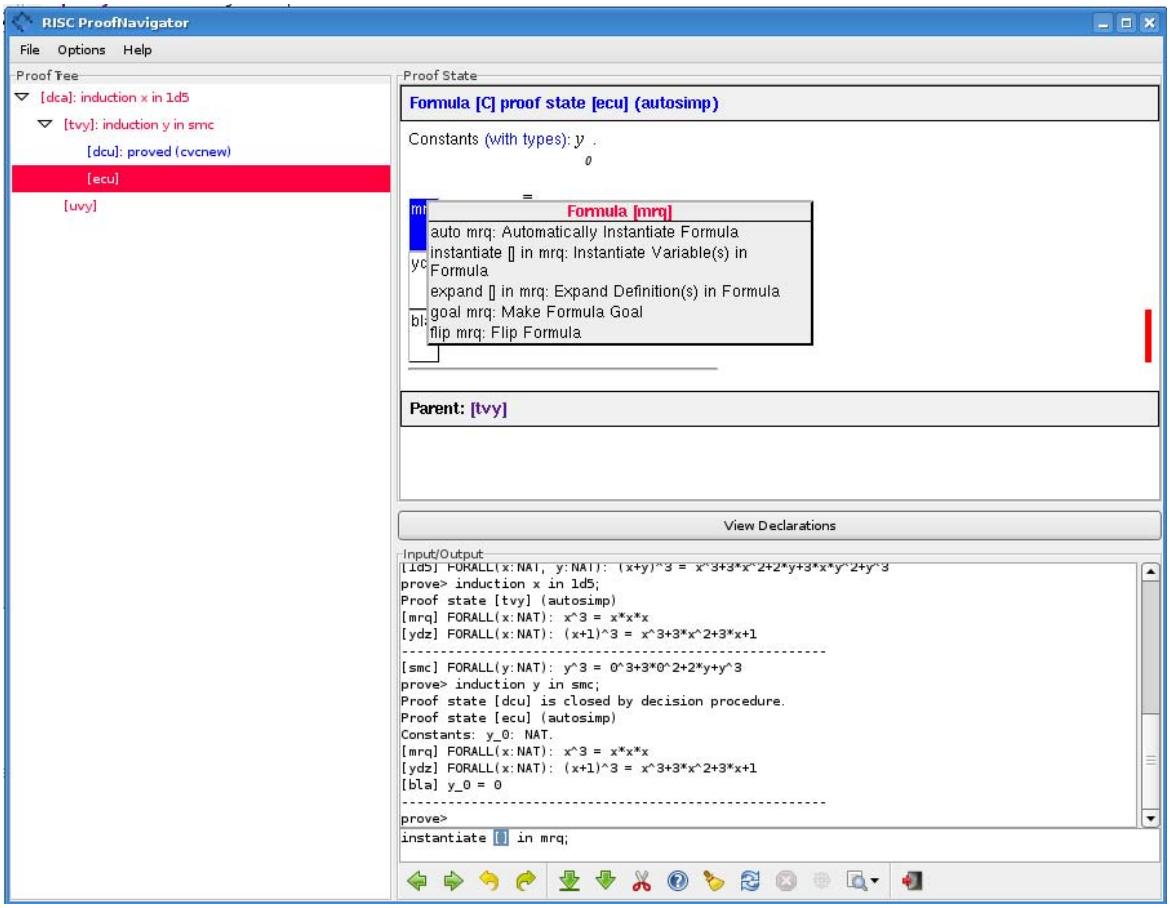

Abbildung 5.10: Autoinstanziierung

Hier sind Auszüge der Kommunikation zwischen CVC3 und dem ProofNavigator beim Aufruf des Beweisschrittes mittels Instanziieren von x

x_0__:INT;
ASSERT x_0__ >= 0;
PUSH;
PUSH;
. . . QUERY x_0__^ 3 = x_0__*x_0__*x_0__;
POP;
POP;
PUSH;

```
PUSH;
QUERY 2^ 3 = 2*2*2;
POP;
POP;
PUSH;
PUSH;
QUERY 1^ 3 = 1;
POP;
POP;
PUSH;
PUSH;
QUERY TRUE;
POP;
POP;
PUSH;
ASSERT _MASK0__;
ASSERT _MASK1__;
ASSERT _MASK2__(x_0__);
ASSERT x_0__^ 2^ 3 = x_0__^ 6;
ASSERT (3*x_0__^ 2)^ 3 = 27*x_0__^ 6;
ASSERT TRUE;
ASSERT (3*x_0__)^ 3 = 27*x_0__^ 3;
ASSERT x_0__^ 3^ 3 = x_0__^ 9;
ASSERT TRUE;
ASSERT TRUE;
ASSERT TRUE;
ASSERT FALSE;
PUSH;
QUERY FALSE;
POP;
POP;
```

Nach der Durchführung diese Beweisschrittes ist der Beweis abgeschlossen (siehe Abbildung 5.11) und kann gespeichert werden.

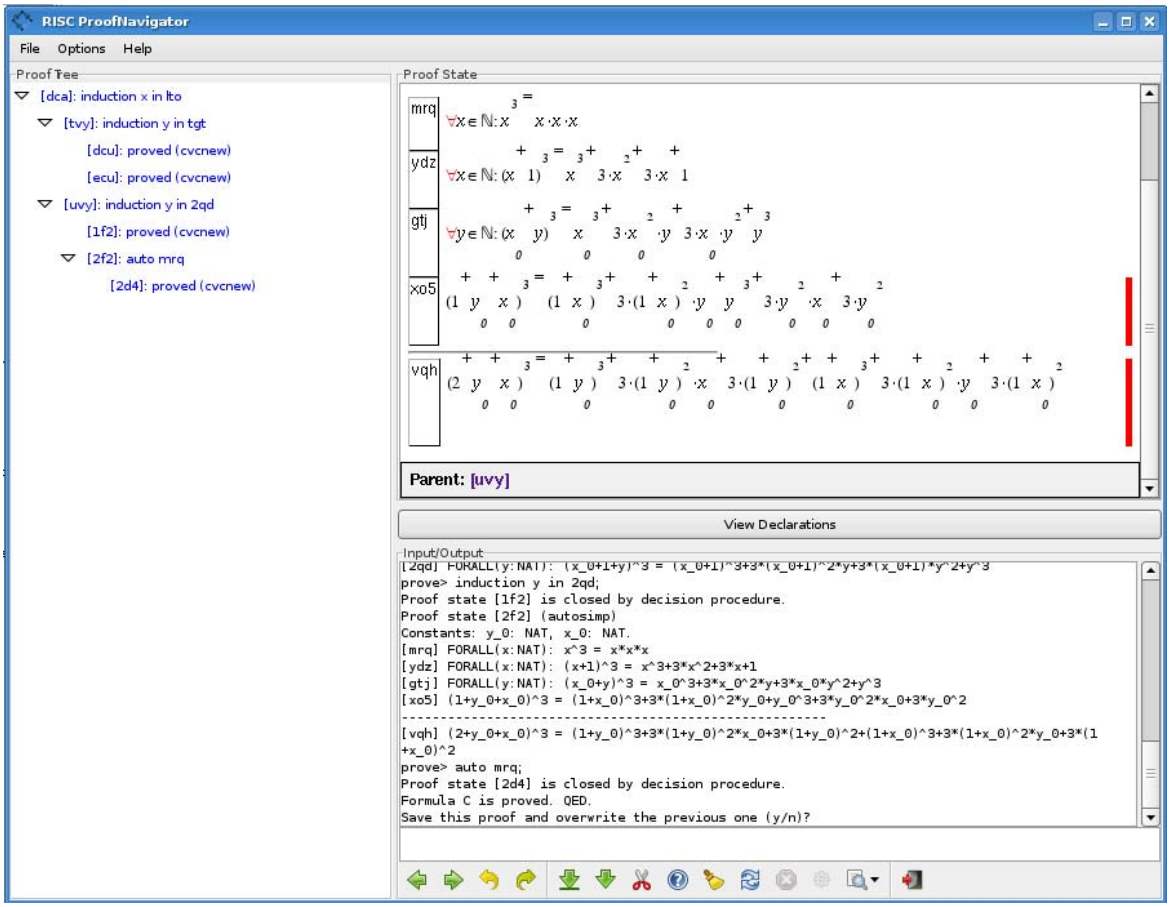

Abbildung 5.11: Abgeschlossener Beweis

6 Schlussfolgerung

Im vorhanden interaktiven Beweissystem, RISC ProofNavigator, ist die automatische Entscheidungsprozedur, CVC Lite, eingebunden. Diese Entscheidungsprozedur versteht Sprachelemente der Logik höherer Ordnung. Daher mussten einige Einschränkungen im existierenden System vorgenommen werden um die neue Entscheidungsprozedur, CVC 3, die nur die Logik der ersten Ordnung verarbeiten kann korrekt einzubinden.

Dazu wurde die Logik der ersten Ordnung, getypte Logik der ersten Ordnung, die Logik höherer Ordnung bis zur getypten Logik der höheren Ordnung definiert und anschließend eine Übersetzung zwischen der getypten Logik der ersten Ordnung und der getypten Logik höherer Ordnung beschrieben und deren Korrektheit bewiesen. Damit wurde die Basis für die Einbindung mehrerer verschiedener Entscheidungsprozeduren gelegt.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfaßt habe. Ich habe dazu keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Juni 2008 _____

Unterschrift

Literaturverzeichnis

[1] A. ARMANDO, E. Giunchiglia M. I. ; MARATEA, M.: *TSAT++: an Open Platform for Satisfiability Modulo Theories*. Electronic Notes in Theoretical Computer Science ENTCS Volume 125, Issue 3, 18 July 2005, Pages 25-36, Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)

[2] AGERHOLM, S.: *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop*. Springer-Verlag, 1994

[3] ARCHER, Myla ; VITO, Ben D. ; MUÑOZ, César: *Writing PVS Proof Strategies*. NASA Langley Research Center, 2003

[4] ARIO. *ARIO SMT Solver*. URL http://www.eecs.umich.edu/ ario/. 2006

[5] BAADER, F.: *Unification Theory*. Springer-Verlag, 1992

[6] BARCELOGICTOOLS. *BarcelogicTools for SMT*. http://www.lsi.upc.edu/ oliveras/bclt -main.html. 2006

[7] BARRETT, Clark. *CVC*. http://verify.stanford.edu/CVC/. 2006

[8] BARRETT, Clark. *CVCLite*. http://www.cs.nyu.edu/acsys/cvcl/. 2006

[9] BARRETT, Clark ; BEREZIN, Sergey: CVC Lite: A New Implementation of the Cooperating Validity Checker Category B. In: *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, ISSN 0302-9743 (Print) 1611-3349 (Online) Volume Volume 3114/2004 Computer Aided Verification, S. 515–518

[10] BERTOT, Yves. *coq*. http://coq.inria.fr/. 2006

[11] BJØRNER, Nikolaj S. ; MANNA, Zohar ; SIPMA, Henny B. ; URIBE, Tomás E.: Deductive Verification of Real-time Systems Using STeP. In: *Technical Report STAN-CS-TR-98-1616, 40 pages*, Computer Science Department, Stanford University, 1998

[12] BRUNO DUTERTRE, Leonardo de M.: *Simplics: Tool Description*. Computer Science Laboratory, SRI International, Menlo Park, CA, 2005

[13] BRYANT, R. ; LAHIRI, S. ; SESHIA, S.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: *Computer Aided Verification*, Springer Berlin / Heidelberg, 2002, S. 106–122

[14] CASTELLINI c. *TSAT++*. http://www.star.dist.unige.it/ marco/Tsat/default.htm. 2006

[15] DECOSTER, Michael. *SAMMY*. http://goedel.cs.uiowa.edu/Sammy/. 2006

[16] DOWEK, Gilles ; HARDIN, Thérèse ; KIRCHNER, Claude: HOL-$\lambda\sigma$: An Intentional First-Order Expression of Higher-Order Logic. In: *RTA*, 1999, S. 317–331

[17] GALLIER, Jean H.: *Logic from Computer Science*. Springer, 1989

[18] GRIGGIO, Roberto Bruttomesso Alessandro Cimatti Anders Franzen A. ; SEBASTIANI, Roberto: *The MathSAT 4 SMT Solver*. CAV, 2008

[19] HARALD GANZINGER, Thomas H. ; WALDMANN, Uwe: Superposition Modulo a Shostak Theory. In: *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, ISSN 0302-9743 (Print) 1611-3349 (Online) Volume 2741/2003 Automated Deduction CADE-19

[20] HOL. *HOL*. http://hol.sourceforge.net/. 2006

[21] KAUFMANN, Matt. *ACL2*. http://www.cs.utexas.edu/users/moore/acl2/. 2006

[22] KAUFMANN, Matt ; MOORE, J. S.: An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. In: *Software Engineering* 23 (1997), Nr. 4, S. 203–213

[23] LEIVANT, Daniel: *Higher Order Logic*. Clarendon Press, 1994

[24] MATHSAT@ITC.IT. *MathSAT*. http://mathsat.itc.it/. 2006

[25] MOURA, Leonardo de: *System Description: Yices 0.1*. Computer Science Laboratory SRI International 333 Ravenswood Avenue Menlo Park, CA 94025 - USA, 2006

[26] MOURA, Leonardo de ; RUESS, Harald ; SOREA, Maria: Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In: *Proceedings of the 18th International Conference on Automated Deduction* Bd. 2392, Springer-Verlag, July 2002. – ISBN 3–540–43931–5, S. 438–455

[27] N.A. *Calculemus*. http://www.eurice.de/calculemus/. 2006

[28] N.A. *VSK*. http://www.software-kompetenz.de/?25308. 2006

[29] NGUYEN, Q. ; KIRCHNER, C. ; KIRCHNER, H.: External rewriting for skeptical proof assistants. In: *Journal of Automated Reasoning, 29(3-4)*. 2002, S. 309–336

[30] NIEUWENHUIS, Robert ; OLIVERAS, Albert ; TINELLI, Cesare: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). In: *Journal of ACM* 53 (2006), Nr. 6, S. 937–977

[31] OWRE, Sam. *PVS*. http://pvs.csl.sri.com/index.shtml. 2006

[32] PAULSON, Larry. *Isabelle*. http://isabelle.in.tum.de/. 2006

[33] PROJECT, The Coq Development Team L.: *The Coq Proof Assistant Reference Manual*. 8.1. INRIA, 2006

[34] ROE, Kenneth: *The Heuristic Theorem Prover: Yet another SMT Modulo Theorem Prover (Tool Paper)*. CAV, 2006

[35] RUESS, Harald. *ICS*. http://ics.csl.sri.com/. 2006

[36] SCHMIDT, David A.: *Denotational semantics: a methodology for language development*. William C. Brown Publishers, 1986

[37] SCHREINER, Wolfgang: Program Verification with the RISC ProofNavigator. In: DUCE, David (Hrsg.) ; BOCA, Paul (Hrsg.): *Teaching Formal Methods: Practice and Experience*. London, UK, December 15 : British Computer Society, 2006 (Electronic Workshops in Computing (eWiC)), S. 1–6

[38] SCHREINER, Wolfgang. *ProofNavigator*. http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator/. 2006

[39] SCHREINER, Wolfgang: *The RISC ProofNavigator Tutorial and Manual*. Research Institute for Symbolic Computation (RISC) Johannes Kepler University, Linz, Austria, 2007

[40] SCHREINER, Wolfgang: The RISC ProofNavigator: a proving assistant for program verification in the classroom. In: *Formal Aspects of Computing*. Springer London, ISSN 0934-5043 (Print) 1433-299X (Online) 3. April 2008

[41] SHANKAR, Natarajan ; OWRE, Sam: Principles and Pragmatics of Subtyping in PVS. In: BERT, Didier (Hrsg.) ; CHOPPY, Christine (Hrsg.) ; MOSSES, Peter (Hrsg.): *Recent Trends in Algebraic Development Techniques, WADT '99* Bd. 1827. Toulouse, France : Springer-Verlag, 1999, S. 37–52

[42] SHEINI, Hossein M. ; SAKALLAH, Karem A.: Pueblo: a modern pseudo-Boolean SAT solver. In: *Design, Automation and Test in Europe* Bd. 2, IEEE, March 2005. – ISBN 0–7695–2288–2, S. 684 – 685

[43] SHUVENDU, Sanjit. *UCLID*. http://www.cs.cmu.edu/ uclid/. 2006

[44] SRI. *Simplics*. http://fm.csl.sri.com/simplics/. 2006

[45] SRI. *Yices*. http://fm.csl.sri.com/yices/. 2006

[46] STEP. *STeP*. http://www-step.stanford.edu/. 2006

[47] STUMP, A. ; BARRETT, C. ; DILL, D.: CVC: a Cooperating Validity Checker. In: *14th International Conference on Computer-Aided Verification*, 2002

[48] TINELLI, Cesare: A DPLL Calculus for Ground Satisfiability Modulo Theories / Department of Computer Science, University of Iowa. 2002. – Forschungsbericht

[49] TINELLI, Cesare. *SMTLIB*. http://goedel.cs.uiowa.edu/smtlib/. 2006

[50] WEBMASTER@FORDOCSYS.COM. *HTP*. http://www.fordocsys.com/htp.htm. 2006

[51] WENZEL, Tobias Nipkow Lawrence C. Paulson M.: *A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2007

[52] WOLFMAN, S. ; WELD, D.: Combining linear programming and satisfiability solving for resource planning. In: *Knowledge Engineering Review* (2000)

[53] WOLFMAN, Steve. *LPSAT*. http://www.cs.washington.edu/ai/lpsat.html. 2006

[54] WOLFMAN, Steven A. ; WELD, Daniel S.: The LPSAT Engine & Its Application to Resource Planning. In: *IJCAI*, 1999, S. 310–317