View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Radboud Repository

Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/75399

Please be advised that this information was generated on 2017-12-06 and may be subject to change.

Correctness and Availability

Building Computer Algebra on top of Proof Assistants
and making Proof Assistants available over the Web

Cezary Kaliszyk

Correctness and Availability

Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web

een wetenschappelijke proeve op het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus
prof. mr. S.C.J.J. Kortmann,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op donderdag 3 september 2009
om 10:30 uur precies

door

Cezary Kaliszyk

geboren op 4 augustus 1981
te Warschau, Polen

Promotor:

Prof. dr. J.H. Geuvers

Copromotor:

Dr. F. Wiedijk

Manuscriptcommissie:

Prof. dr. H.P. Barendregt	
Prof. dr. M. Beeson	(San Jose State University)
Prof. dr. A.M. Cohen	(Technische Universiteit Eindhoven)
Dr. J.R. Harrison	(Intel Corporation)
Dr. J. McKinna	

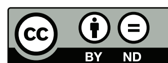
This research was partially supported by the NWO Project 600.065.130.24N19 FEAR and partially supported by the SURF Project *Web-deductie voor het onderwijs in formeel denken.*
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

© Copyright 2009 Cezary Kaliszyk

ISBN 978-90-9024444-0

IPA Dissertation Series 2009-18

Typeset with LaTeX 2_ε
Printed by PrintPartners Ipskamp, Enschede.



This work is licensed under a Creative Commons Attribution-No Derivative Works 3.0 Netherlands License. To view a copy of this license, visit: `http://creativecommons.org/licenses/by-nd/3.0/nl/`.

Acknowledgments

First, I would like to thank my co-promotor Freek Wiedijk and my promotor Herman Geuvers not only for their scientific advice and research suggestions but also for the inspiration and motivation they gave me.

I thank the reading committee, consisting of Henk Barendregt, Michael Beeson, Arjeh Cohen, John Harrison and James McKinna.

Many people showed interest in my research. The discussions with Russell O'Connor, Dan Synek, Pierre Corbineau, Bas Spitters, Venanzio Capretta, Lionel Mamane influenced me in my research. I thank everyone of the Foundations group in Nijmegen for being social and friendly and for creating a nice working environment.

For a year my research was also carried out within the SURF *Web Deduction* project. I am grateful to its members, and I would especially like to thank Maxim Hendriks and Femke van Raamsdonk. For three months I pursued an internship at the National University of Singapore. I would like to thank Chin Wei Ngan and everyone in the PLS II laboratory at NUS.

I would like to thank my friends at work: Wojtek, Łukasz, Pavol, Henriëtte, Botond, Martin, Agnieszka. The discussions with you brought many interesting ideas, and the coffee breaks together were enjoyable.

Finally, my greatest gratitude goes to my family, for being there when necessary. Your support and understanding were indispensable throughout my education.

Nijmegen, 2009

Contents

Acknowledgments v

Introduction 1

I Basing Computer Algebra on Proof Assistants 9

1 Computer Algebra in HOL Light 11
- 1.1 Introduction 11
  - 1.1.1 Motivation 11
  - 1.1.2 Approach 12
  - 1.1.3 Related work 13
  - 1.1.4 Contents 15
- 1.2 Architecture 15
  - 1.2.1 Input-response loop 16
  - 1.2.2 Abstract CAS conversion 17
- 1.3 CAS-like knowledge 19
  - 1.3.1 Knowledge base 19
  - 1.3.2 Knowledge representation 20
  - 1.3.3 Numerical approximations 21
  - 1.3.4 Assumptions 22
  - 1.3.5 Manipulating assumptions 24
- 1.4 Concluding remarks 25

2 Automating partiality side conditions 27
- 2.1 Introduction 27
  - 2.1.1 Motivation 27
  - 2.1.2 Approach 29
  - 2.1.3 Related work 29
  - 2.1.4 Contents 30
- 2.2 Approach 30
  - 2.2.1 Basic definitions 30
  - 2.2.2 Example in mathematical notation 31
- 2.3 The formalization 33

2.3.1 Design decisions 33
2.3.2 HOL Light implementation details 33
2.3.3 How to extend the system 38
2.4 Concluding Remarks 39

3 Computing with classical real numbers 41
3.1 Introduction 41
3.1.1 The two universes of Coq 42
3.2 Logical Consequences of Coq real numbers 43
3.2.1 The axiomatic definition of the real numbers 43
3.2.2 Decidability of Π^0_1 sentences 44
3.3 The construction of the isomorphism 46
3.3.1 Building a constructive reals structure based on Coq reals 46
3.3.2 The isomorphism 48
3.4 Computation with classical reals 50
3.4.1 Solving ground inequalities 50
3.4.2 Using facts about Coq reals in CoRN 51
3.5 Related Work 52
3.6 Concluding remarks 53

II Interactive formalized math on the web 55

4 Web Interfaces for Proof Assistants 57
4.1 Introduction 57
4.1.1 Motivation 57
4.1.2 Our Approach 58
4.1.3 Related work 58
4.1.4 Contents 59
4.2 Asynchronous DOM Modification 60
4.3 Generic Interface for Proof Assistants 61
4.4 General Architecture 62
4.4.1 The Client Part 63
4.4.2 The Server Part 63
4.5 Security and Efficiency 64
4.5.1 User side 64
4.5.2 Server side 64
4.6 Prototype 65
4.6.1 Possible Uses 67
4.7 Implementation 67
4.7.1 The server process 68
4.7.2 The server environment 68
4.7.3 Client side 69
4.8 Concluding Remarks 70

5 Teaching logic using a proof assistant **73**
5.1 Introduction . . . 73
5.1.1 Motivation . . . 73
5.1.2 Our contribution . . . 74
5.1.3 Related work . . . 75
5.1.4 Contents . . . 75
5.2 Experiences in the project . . . 75
5.3 Architecture of the interface . . . 77
5.4 Natural deduction for first-order logic . . . 77
5.4.1 Visualization . . . 79
5.5 The exercise set . . . 80
5.6 Outlook . . . 83
5.6.1 Beyond the project . . . 84

6 Merging proof styles **85**
6.1 Introduction . . . 85
6.1.1 Procedural versus declarative proof assistants . . . 85
6.1.2 Approach . . . 90
6.1.3 Related Work . . . 91
6.1.4 Contents . . . 91
6.2 Translating minimal logic tree proofs . . . 92
6.3 Translating proofs in more complicated logical systems . . . 94
6.4 Simplification of obtained proofs . . . 95
6.5 Simplification of forward proofs . . . 98
6.6 Implementation for Coq proofs . . . 99
6.6.1 Transforming Coq proof state in a flag style proof . . . 100
6.7 Concluding Remarks . . . 101

7 Repositories for formal proofs **103**
7.1 Introduction . . . 103
7.1.1 Motivation . . . 103
7.1.2 Related Work . . . 104
7.1.3 The future of proof interfaces . . . 105
7.1.4 Chapter contents . . . 105
7.2 Web Technologies . . . 106
7.2.1 Wikis . . . 106
7.3 Architecture . . . 106
7.3.1 Main Components . . . 106
7.3.2 Global Design . . . 107
7.3.3 Consistency Issues . . . 108
7.3.4 Towards a hybrid approach . . . 109
7.4 Prototype . . . 110
7.4.1 Implementation . . . 110
7.4.2 Security and Efficiency . . . 113
7.4.3 How to integrate other provers . . . 114
7.5 Concluding Remarks . . . 114

8 Environment for Computer Mathematics **117**
8.1 Vision 117
8.2 Progress of this thesis towards the vision 118
8.2.1 Multivaluedness 118
8.3 Future Work towards obtaining the Vision 119
8.3.1 Computer Algebra with Strong Semantics 119
8.3.2 MathWiki 119
8.3.3 Using a web interface for software 119
8.4 Conclusion 120

Bibliography **121**

Summary **133**

Samenvatting (Dutch summary) **135**

Streszczenie (Polish summary) **139**

Curriculum Vitae **141**

Introduction

In this thesis we present an approach to extending the usability of proof assistants in mathematics and computer science. We do it in two ways: by combining proof assistants with computer algebra systems and by providing interactive access to such systems on the web.

Proof Assistants

Proof assistants are computer programs that help the user build a proof that is mechanically checked. In a typical proof assistant one can set up a theory that describes mathematical concepts or models a computer system. Proof assistants support expressing logical properties of the objects in this theory, proving those properties and checking such proofs.

A formalization of a property in a proof assistant ensures that there are no mistakes in a mathematical proof or that there are no errors in the model of a piece of software. Mathematical proofs are sometimes not accepted by the community of mathematicians, since they are too large to be comprehensible, or because they require results from computer programs [Hal05]. In computer science mistakes may have even more severe consequences. Examples include the error introduced by Intel in the Pentium processor chip [Pra95] which cost the company millions of dollars and the conversion error in the Ariane 5 missile software which caused it to explode 36 seconds after launch.

Proof assistants are typically interactive tools. The specified theory and proofs can be edited. The tool provides information about required proof obligations and allows further refinement of the proof. This is often done by manually providing a direction in which to proceed. Some proof assistants allow for automation. This is provided by various decision procedures that solve or simplify goals in a particular theory. Finally proof assistants may check a complete proof, sometimes providing a certificate of correctness.

There are many proof assistants. There are multiple reasons for this. First, there are many possible mathematical foundations. There exist proof assistants based on type theory, on higher order logic and on set theory. Some are based on classical logic and some on intuitionistic logic. Second, there are many interaction models. In some proof assistants proofs are processed by stepping over a proof script, in some proof assistants proofs are sequences of

toplevel commands (LCF-style), in others proofs are processed in batch mode as a whole. Some proof assistants are tactic based and some implement declarative languages. Furthermore there are many automation strategies or decision procedures. The level of development of the libraries of various proof assistants differs per domain.

The most important proof assistants are Coq, HOL Light, PVS [ORS92, Pvs], Isabelle [NPW02], NuPRL [NuP], Agda [Agd], HOL [Hol], Mizar [Muz93, Miz], and ACL2 [KM96].

Coq [CDT08, BC04] has been developed since 1984 at INRIA in France. It is based on the *Calculus of Inductive Constructions* which is a version of type theory. It has been implemented in Objective Caml, and has been used for the formal verification of many proofs, both from mathematics and from computer science. The most impressive verifications done using Coq are the proof of the Four Colour Theorem by Georges Gonthier [Gon06] and the development of a verified C compiler by Xavier Leroy and others [Ler06].

HOL Light [Har96a] is a proof assistant for classical higher order logic. It has been developed by John Harrison. It is implemented in Objective Caml and is an extension of its toplevel. Compared to other proof assistants it has a relatively simple foundations and its kernel (the core part that implements the logic) is very small. The most impressive results created in HOL Light are the proof of the Jordan Curve Theorem by Thomas Hales and the proof of the correctness of the kernel [Har06]. It is also the theorem prover with the highest number of theorems proved from a list of top 100 mathematical theorems [Wie09].

The use of proof assistants is currently limited to specialists in the domain. Formalization is a difficult process for many reasons. First, to make a proof as done on paper accepted by a proof assistant, one needs to add many details. Furthermore analyzing software in a proof assistant requires not only modeling the software itself, which may be a big piece of code, but also the semantics of interpretation or compilation of the software as well as the whole environment in which the software is supposed to run.

This is why proof assistants are mostly used by specialists, and are not widely known to mathematicians and computer scientists. In contrast mathematicians often use computer algebra software.

Computer Algebra Systems

Computer algebra systems (*CASs*) are computer programs that process mathematical expressions. Most CASs allow manipulation of symbolic expressions including automatic simplifications, performing substitutions, solving equations over various domains, calculating arbitrary precision numerical approximations and plotting graphs of functions.

The main computer algebra systems are: Maple [CGGG83, CGG$^+$91], Mathematica [Wol03] and Matlab. They are all commercial programs with no source available. They include general purpose mathematical utilities, not only computer algebra. Each of these three is based on a small efficient kernel written in

a low-level language and each provides different versions of high level languages designed with writing mathematical algorithms in mind.

Many CASs are designed to be easy to use for a beginner. There are many ways in which this is achieved. The interface of most CASs resembles an advanced calculator. Computer algebra systems allow the users to enter mathematical expressions in traditional mathematical notation and output results in user friendly format. There are built-in mechanisms to automatically handle partiality as well as care about side conditions in the expressions. Defining structures or functions and computing with them is easy, so performing experiments inside a CAS is simple.

This simplicity comes with a drawback. The algorithms implemented in those systems are not formally checked and are therefore known to make mistakes [Asl96, Wes99]. Even further: the formulas and the algorithms of computer algebra systems do not have a precise semantics. They are implemented in such a way to resemble the style mathematicians simplify expressions on paper, so it is not always obvious what are the correctness criteria for such simplifications. The errors may come from missing assumptions, incorrect types, incorrect handling of multi-valuedness or from errors in the algorithm implementation.

Web technology

Most proof assistants are interactive. When a theorem is expressed in a proof assistant the proof follows by the user issuing a sequence of commands, that transform the state. The proof assistant responds with the remaining goals to be proven. The resulting sequence of commands, called the *proof script* is often unreadable without the intermediate states.

This is often the case when proofs are rendered on the Web. Most proof assistants already provide tools for rendering proofs as HTML pages, sometimes using advanced math rendering schemes like MathML [CIMP03]. The communities of users of various provers have a variety of tools that allow the display of created information about developments in them on the web in static representations. Examples include the HELM tool [APCS01] developed in the MoWGLI project and the Mizar Mathematical Library [BR03].

With the development of web technology people are more and more accustomed to having a connection to the internet at all times. There are many services available just by accessing certain web pages. This allows one to use a particular tool without installing it on ones machine. Examples include web interfaces to e-mail, calendars, chat clients, word processors and maps.

One of the most knows services available via the web is Wikipedia. It is a free, multilingual encyclopedia based around the wiki concept. A wiki is a collection of Web pages that allow visitors to modify content or add pages, with the changes published immediately. Wikis usually offer simplified markup languages to allow users to quickly create elegant pages. There are many implementations of the wiki concept, but all allow *collaborative authoring* of knowledge repositories.

There are many services for doing informal mathematics online; examples include computation, computer algebra systems available on the web [Map95], as well as systems for gathering and exploring mathematical knowledge [Gru06].

Advances in web technology allow the creation of an interactive interface to proof assistant that resembles, and has the functionality of a prover interface running locally, but which is available completely within a web browser. In the second part of this thesis, we present such an interface, as well as possible extensions that which allow it to be used for teaching and in a wiki environment.

Contents of this Thesis

The thesis consists of two parts. Part I provides approaches that combine computer algebra systems and proof assistants. Part II presents the use of proof assistants on the web.

The chapters of this work are based on peer reviewed published articles. Slight modifications were made to the chapters to avoid redundancy. Furthermore the layout and visual style of the papers was changed to improve consistency.

Many of the code fragments presented in the Thesis are written in Coq and HOL Light. We assume the knowledge of the syntax of these proof assistants and basic knowledge of functional programming to understand the presented proof assistant definitions, tactics, and proof scripts.

Chapter 1:

Various algorithms and simplifications present in computer algebra systems are already available in state-of-the-art proof assistants. In this chapter we present a prototype computer algebra system built on top of a proof assistant, HOL Light. This architecture guarantees that one can be certain that the system will make no mistakes. All expressions in the system will have precise semantics and the proof assistant will check the correctness of all simplifications according to this semantics. The system actually *proves* each simplification performed by the computer algebra system.

Although our system is built on top of a proof assistant, we designed the user interface to be very close in spirit to the interface of systems like Maple and Mathematica. The system, therefore, allows the user to easily probe the underlying automation of the proof assistant for strengths and weaknesses with respect to the automation of mainstream computer algebra systems. The system that we present is a prototype, we describe how it can be scaled up to a practical computer algebra system.

This chapter is an adaptation of a peer reviewed article [KW07]. The article was written in collaboration with Freek Wiedijk, who contributed to designing the behavior of the system and the semantics of the simplification operations.

Chapter 2:

The computer algebra system designed in Chapter 1 had trouble dealing with even the simplest partial functions. This was due to the fact that assumptions about the domains of partial functions are necessary when we guarantee correctness. On the other hand when mathematicians write about partial functions they tend not to explicitly write these side conditions. In this Chapter we present an approach to formalizing partiality in real and complex analysis in total frameworks that allows the side conditions to be kept hidden from the user as long as they can be computed and simplified automatically. This framework simplifies defining and operating on partial functions in formalized real analysis in HOL Light. Our framework allows simplifying expressions under partiality conditions in a proof assistant in a manner that resembles computer algebra systems.

This chapter is an adaptation of the peer reviewed article [Kal08].

Chapter 3:

In Computer Algebra we use floating point numbers to approximate real number expressions. Numeric methods (e.g. interval arithmetic) are used to provide correctness of these approximations. In Coq, Russell O'Connor's work [O'C07] allows working with infinite precision real numbers effectively. In this chapter we investigate how to use the classical theory of real numbers together with approximations computed constructively.

There are two main Coq libraries that have a theory of the real numbers. The Coq standard library gives an axiomatic treatment of the classical theory of real numbers, while the CoRN library from Nijmegen defines a constructively valid theory of real numbers. We present a way of making these two libraries compatible by showing that their real number structures are isomorphic assuming the classical axioms already present in the standard library reals.

To do this we show that the axioms of classical Coq reals imply decidability of Π^0_1 sentences. We show that if $\varphi(n)$ is decidable for any number n, then we can decide if it is true for all numbers, and this decision can be made using the constructive disjunction $\oplus$. Namely: $(\forall n : \mathbb{N}.\ \varphi(n) \oplus \neg\varphi(n)) \rightarrow (\forall n : \mathbb{N}.\ \varphi(n)) \oplus \neg(\forall n : \mathbb{N}.\ \varphi(n))$. This shows that classical logic leaks into the computational part.

We then show that this isomorphism preserves some basic constants (0, 1, e, π), arithmetic operations and the elementary transcendental functions (exponential, logarithm, sine, cosine). This allows us to use O'Connor's decision procedure for solving ground inequalities present in CoRN using the theory of real numbers from the Coq standard library. It also makes it possible to use some of the theorems from the Coq standard library with the CoRN reals, if one does not mind having classical logic.

This chapter is a slight adaptation of a submitted article [KO08]. It was written in collaboration with Russell O'Connor who developed the fast implementation of the constructive reals in CoRN and who helped with the formalizations

presented in the chapter.

Part II

In the second part of this thesis we look at interactive formalized mathematics on the web. We describe a web interface for proof assistants and we investigate the use of this interface in teaching logic and in collaborative proof development.

Chapter 4:

This chapter describes an architecture for creating responsive web interfaces for proof assistants. The architecture combines current web development technologies with the functionality of local prover interfaces. We create an interface that is available completely within a web browser, but resembles and behaves like a local one. Security, availability and efficiency issues of the proposed solution are described. We present a prototype implementation of a web interface for Coq and describe our experiments with other proof assistants.

This chapter is an adaptation of the peer reviewed article [Kal07].

Chapter 5:

This chapter describes the system ProofWeb developed for teaching logic to undergraduate computer science students. The system is based on the proof assistant Coq. It is made available to the students through the interactive web interface. Part of this system is a large database of logic problems. This database also holds the solutions of the students. The students do not need to install anything beyond a web browser to be able to use the system (not even a browser plug-in) and the teachers are able to centrally track progress of their students.

The system makes the full power of Coq available to the students, but simultaneously presents the logic problems in a way that is customary in undergraduate logic courses. Part of the system is a parser that indicates whether the students used the automation of Coq to solve their problems or that they solved it themselves using only the inference rules of the logic. For these inference rules dedicated tactics for Coq have been developed.

The system has already been used in a number of type theory courses and logic undergraduate courses at the universities in Nijmegen, Amsterdam and Eindhoven.

This chapter is a slight adaptation of the peer reviewed article [MHW08]. It was written in collaboration with Maxim Hendriks, Femke van Raamsdonk and Freek Wiedijk. The database of the exercises was created by Maxim Hendriks, the special Coq tactics were written by Maxim Hendriks and Freek Wiedijk.

Chapter 6:

There are two different styles of writing natural deduction proofs:

- the 'Gentzen' style in which a proof is a tree with the conclusion at the root and the assumptions at the leaves,
- the 'Fitch' style (also called 'flag' style) in which a proof consists of lines that are grouped together in nested boxes.

In proof assistants these two kinds of natural deduction correspond to *procedural proofs* and *declarative proofs. Procedural proofs are tactic scripts that work on one or more subgoals*, like those of the Coq, HOL and PVS systems. Declarative proofs are sequences of named statements, like those of the Mizar and Isabelle/Isar languages.

In this chapter we give an algorithm for converting tree style proofs to flag style proofs. We then present a rewrite system that simplifies the results.

This algorithm can be used to convert arbitrary procedural proofs to declarative proofs. It does not work on the level of the proof terms (the basic inferences of the system), but on the level of the statements that the user sees in the goals when constructing the proof.

The algorithm from this chapter has been implemented in the ProofWeb interface to Coq. In ProofWeb a proof that is given as a Coq proof script (even with arbitrary Coq tactics) can be displayed both as a tree style and as a flag style proof.

This chapter is an adaptation of an article submitted to [KW08] written in collaboration with Freek Wiedijk.

Chapter 7:

In this chapter we combine the web interface from Chapter 4 with a wiki to create a complete environment for the online development of formalized mathematics. In this framework users can collaborate and see a rendered and browsable version of their work. We describe a prototype based on Coq, its web interface, and a modified version of the MediaWiki code-base. We discuss computing dependencies and preserving repository consistency. We explain limitations of the current prototype and we give a perspective towards a more robust solution.

This chapter is an adaptation of the peer reviewed article [CK07]. It was written in collaboration with Pierre Corbineau. We collaborated in modifying MediaWiki and he adapted the coqdoc utility to produce MediaWiki compatible output.

Part I

Basing Computer Algebra on Proof Assistants

Chapter 1

Certified Computer Algebra on top of an Interactive Theorem Prover

1.1 Introduction

1.1.1 Motivation

In this chapter we study the relationship between proof assistants and computer algebra systems trying to find a semantically valid and user friendly approach to manipulating mathematical formulae.

A computer algebra system has at its core a set of algorithms for processing mathematical expressions, which is provided to the user through an interface. The formulas, that the algorithms of a computer algebra system manipulate generally do not have precise semantics. They are simplified in a way that resembles the style a mathematician simplifies expressions on paper. To perform such simplifications the symbols that occur in those formulas do not need a mathematical definition inside the system.

On the other hand, the mathematical symbols that occur in proof assistants always have precisely defined semantics. Proof assistants not only manipulate symbolic expressions, but also theorems that are stated using those symbols. The basic activity in a proof assistant is not simplification of expressions but construction and checking of proofs of theorems. A proof assistant generally does not have many algorithms available for automating the application of those theorems, and to make use of those that are there one needs to have good knowledge of the prover environment.

Computer algebra systems do not always give correct answers. This happens because those systems do not certify the operations performed. There can be various reasons for errors in a CAS: assumptions can be lost, types of expressions can be forgotten [Asl96], the system might get confused between branches of

'multi-valued' functions, and of course the algorithms of the system themselves may contain implementation errors [Wes99].

As an example of the kind of error that we are talking about here, consider the following Maple session that tries to compute $\int_0^\infty \frac{e^{-(x-1)^2}}{\sqrt{x}}dx$ numerically in two different ways:

```
> int(exp(-(x-t)^2)/sqrt(x), x=0..infinity);
```

$$\frac{1}{2}\frac{e^{-t^2}\left(-\frac{3(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{3}{4}}(\frac{t^2}{2})}{t^2}+(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{7}{4}}(\frac{t^2}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> subs(t=1,%);
```

$$\frac{1}{2}\frac{e^{-1}\left(-3\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{3}{4}}(\frac{1}{2})+\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{7}{4}}(\frac{1}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> evalf(%);
```

$$0.4118623312$$

```
> evalf(int(exp(-(x-1)^2)/sqrt(x), x=0..infinity));
```

$$1.973732150$$

The two numerical values obtained are significantly different. The first value was obtained by integrating the function symbolically, substituting the value of t to 1 and computing an approximation. The second value was obtained by integrating with the value of t substituted. (We are showing Maple here, but all major computer algebra systems make errors like this.)

To be sure that results are correct, one may use a proof assistant instead of a CAS. But in that case even calculating simple things, like adding fractions or calculating a derivative of a polynomial becomes a non-trivial activity, which requires significant experience with the system.

1.1.2 Approach

Our approach is to implement a computer algebra system on top of a proof assistant. For our prototype we chose the LCF-style theorem prover HOL Light [Har96a]. We obtain a CAS system where the user can be sure of the correctness of the results. Such a system has strong semantics, that is all variables have types, all functions have precise definitions in the logic of the prover and for every simplification there is a theorem that ensures the correctness of this simplification.[1] The interface of our computer algebra system resembles

[1]In HOL Light *simplification* is implemented through what in the LCF world is called *conversions* [Pau83]. A conversion is a function that takes a term and returns an equational theorem. The theorem has the given term on its left side and a simplified version of the term on the right side.

In this chapter 'simplification' should *not* be taken to be a fixed reduction hard-wired into the logic of the proof assistant, the way it is in type theoretical systems like Coq [CDT08].

most CAS systems. It has a simple read-eval-print loop. The language of the formulas typed into the system is as close as possible to the language in which formulas are generally entered in CAS and to the language in which mathematics is done on paper. Interaction with the system currently looks like this:

```
 In1 := (3 + 4 DIV 2) EXP 3 * 5 MOD 3
Out1 := 250
 In2 := vector [&2; &2] - vector [&1; &0] + vec 1
Out2 := vector [&2; &3]
 In3 := diff (diff (λx. &3 * sin (&2 * x) + &7 + exp (exp x)))
Out3 := λx. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
        - &12 * sin (&2 * x)
 In4 := N (exp (&1)) 10
Out4 := #2.7182818284 + ... (exp (&1)) 10 F
 In5 := 3 divides 6 ∧ EVEN 12
Out5 := T
 In6 := Re ((Cx (&3) + Cx (&2) * ii) / (Cx (- &2) + Cx (&7) * ii))
Out6 := &8 / &53
 In7 := x + &1 - x / &1 + &7 * (y + x) pow 2
Out7 := &7 * x pow 2 + &14 * x * y + &7 * y pow 2 + &1
 In8 := sum (0,5) (λx. &x * &x)
Out8 := &30
```

In the above:

- `&` is the coercion from natural numbers to real numbers
- `Cx` is the coercion from real numbers to complex numbers
- `#` is the notation for decimal floating point numbers
- `vector` is a notation for vectors that give all their coordinates
- `vec` is a notation for a vector that is same in all dimensions
- `N r p` is a notation for a decimal approximation of the number r to decimal precision p and `...` is the approximation rest, explained further in Section 1.3.3

1.1.3 Related work

One can distinguish three categories of systems that try to fill the gap between computer algebra and proof assistants:

- Theorem provers inside computer algebra systems:
  - Analytica [BCZ98],
  - Theorema [BDJ+00],
  - RedLog [DS97],
  - logical extension of Axiom [PT98].
- Frameworks for mathematical information exchange between systems:

  - MathML [CIMP03],
  - OpenMath [BCC+02],
  - OMSCS [BCGH99],
  - MathScheme [CFW03] (information exchange within the system),
  - Logic Broker [AZ00].

- Bridges between theorem provers and computer algebra systems, also referred to as ad-hoc information exchange solutions:
  - PVS and Maple [ADG+01],
  - HOL and Maple [HT98],
  - Isabelle and Maple [BHC95],
  - NuPrl and Weyl [Jac95],
  - Omega and Maple/GAP [Sor00],
  - Isabelle and Summit [BP99].

In the bridges category, it is important to distinguish different levels of *degree of trust* between the prover and the CAS. In certain approaches the prover uses the algorithms present in the CAS without checking their correctness, i.e. as an oracle. In other approaches the prover takes the CAS output and then builds a verified theorem out of it. In this case there are again two possibilities: either the result is verified independently of how the CAS obtained it, or the system takes a trace of the rules that the CAS applied, and then uses that as a suggestion for what theorems should be used to construct a proof of the result.

In the approaches mentioned above either the proof assistant is built inside the CAS, or the proof assistant and the CAS are next to each other. In our work however, we have the CAS inside the proof assistant.

Of course in many proof assistants there already is CAS-like functionality, in particular many proof assistants have arithmetic procedures and powerful decision procedures. However, we do not just provide the functionality, but also build a *system* that can be used in a similar way as most other computer algebra systems are used.

Our system is the first combination of a CAS *inside* a proof assistant (in which all simplifications are validated), with an interface that has the customary CAS look and feel.

The advantage of this approach is that all calculations done by our system are certified by the architecture of our system. No translation of formulas or semantics is necessary, as the CAS shares the internal data structures of the proof assistant. There is no need to worry about mistakes in the implementation of the CAS, since all conversions are certified using the logic of the underlying prover. There is no verification required after the result is obtained, thanks to the creation of theorems alongside with the results. All simplifications performed by our architecture are completely certified, that is if a certificate for a particular simplification does not exist [BC01] it cannot be performed.

1.1.4 Contents

The chapter is organized as follows: in Section 1.2 we present the architecture of the system. In Section 1.3 we talk about the knowledge base. Finally in Section 1.4 we present a conclusion.

1.2 Architecture

We present a general architecture for a certified computer algebra system, and we will describe an implementation prototype [Kal]. For the implementation we chose the proof assistant HOL Light [Har96a]. The factors that influenced our choice were: the possibility to manipulate terms to create the conversions, prove theorems and implement the system in the same language[2], as well as a good library of analysis and algebra. The system created is rather a proof of concept than a real product, which is why the efficiency of the underlying prover was not a decisive factor. In particular we perform all computations inside the proof assistant's logic.

Our system is divided in three independent parts (Fig. 1.1): the user interface (input-response loop), the abstract algorithm of dealing with a formula (we will call this *the CAS conversion*), and the knowledge that is specific to the CAS system. That architecture allows the user both to use it as a computer algebra system, as well as making it usable in the context of theorem proving[3].

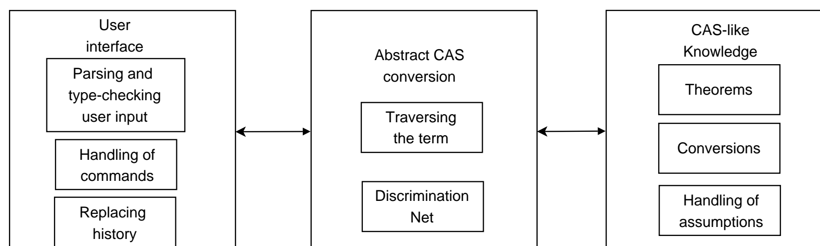


Figure 1.1: Architecture of a CAS inside a TP system with responsibilities of the parts of our implementation marked. The prover is not marked on the figure, since all parts make use of it, by using its type of terms and theorems, as well as tactics and conversions to build them.

HOL Light does not have its own toplevel, instead it uses the OCaml toplevel and syntax with additional parsing and printing functionality. It is loaded inside the OCaml toplevel by reading the prover files one by one and executing the definitions provided there inside the interpreter. All definitions and proofs are valid OCaml statements given in the programming language syntax and the

[2]HOL Light is written in OCaml and is provided as an extension of it. Terms and theorems are defined as ML types in the toplevel and tactics are provided as ML functions. The only modification to the toplevel itself are special parsing and printing mechanisms provided.

[3]*The CAS conversion* can be applied to a goal to be proved using `CONV_TAC`.

theorems proved are objects of the theorem type. The prover libraries are also provided as OCaml source files. Additionally they often provide a `make.ml` file that loads all the files needed for a particular theory. We implemented the CAS in a similar way; that is providing a make file that loads all the needed theories and our files. HOL Light is often checkpointed for reuse and this can be also done with the CAS built inside HOL.

HOL Light does not have a unified library. The user chooses a number of theories that will be requires in a particular development. In our prototype implementation we load the main theory of HOL Light and the following independent theories:

- Binomials
- Permutations
- Vectors
- Matrix operations (Determinants)
- Real analysis and Transcendental functions
- Approximations and decimal approximations
- Rounding
- Complex Analysis

1.2.1 Input-response loop

The system displays a prompt, where one can write expressions to be simplified, as well as commands. We distinguish expressions to be computed or simplified from commands that represent actions that do not evaluate anything, like listing theorems or modifying and printing assumptions.

Every expression that is not recognized as a command is passed to *the CAS conversion*, which will try to compute or simplify the expression. The theorem given back by *the CAS conversion* is the certificate that the output is correct. If *the CAS conversion* is not able to simplify the term, the system returns an instance of reflexivity, and the output is then the same as the input.

In most CASs variables can be used without declaring them, but for certain algebraic operations one can define a variable to be of a particular type (necessary for example in Magma). Our system can handle expressions in both ways. The free variables are typed using HOL Light type inference, but one can also require a specific type with the `assumetype` command (described in Section 1.3.4).

Most computer algebra systems allow one to reuse previously typed in expressions and calculated outputs. One may calculate `In1 + Out2`. The loop has to have access to all expressions entered, theorems proved and outputs. In our framework every expression entered is stored with its type, so when it is reused, parsing the same expression, even in a different context, gives the same type.

The input-response loop is provided in two files. The first, `cas_commands.ml` has the following functions:

- to store a list of already known commands, and implements a handle for parsing and executing commands
- to create a hash table that will be used as a cache for the intermediate computation results. Since the cache will be used in the whole processing it has to be done first. Commands for printing the cache and clearing it are also provided here.
- to create a list of assumptions and the conversions that perform simplification under assumptions and assume variables. A command for printing all assumptions as well as assumptions about a particular variable (`about var`) are added; those will be described in Subsection 1.3.5.

`cas_loop.ml` contains the rest of the functionality:

- A modified version of the term parsing mechanism. The default term parsing mechanism prints a warning when a term contains variables that are not defined or quantified. Since we use the parsing mechanism in a different way, this would mean that the user would be given a huge amount of warnings about every input. We allow inferring the types of variables according to prioritized types, and ignore the warnings.
- A history of inputs and outputs and a mechanism that is able to distinguish `In[]` and `Out[]` terms in the input. They are replaced with the stored preterms and the whole expression is reparsed. This allows history replacement that is compatible with the HOL Light type-checking kernel.
- A function (`cas_mainloop`) which presents a user with an input-response loop that performs a conversion on every input. The input after parsing and history replacement is processed by the conversion and the output is printed. The mainloop also calls the commands defined in the previous file. This is intended to be used with the CAS conversion described in the next subsection, but is independent from it.
- The history of all the theorems proved by the conversion and the `theorems` command that prints the simplification history.

1.2.2 Abstract CAS conversion

To be able to benefit from CAS simplification in theorem proving, it is useful to have the CAS functionality available as a single conversion (that we call here *the CAS conversion*). Since the underlying prover can be further developed with theorems proved later, it is useful to separate *the CAS conversion* from the knowledge that it uses. For this reason *the CAS conversion* is parametrized. The general idea behind *the CAS conversion* is to try to apply all sub-conversions from the knowledge base at all positions in the term until no more simplifications

can be performed (Fig. 1.2). Applying the same conversions to a modified term is necessary, since some conversions return terms, parts of which can be again simplified.

We are not aiming at completness of the conversions, since completeness can be only assured for simple theories. However any algorithm that exists for computer algebra systems can be implemented in a HOL Light conversion that does the same calculation while building the proof of correctness of this conversion. Examples may include conversions that perform algorithms for integration, conversions that perform a split and join for calculating results that have more branches or conversions that simplify term operations (for example a higher order summation operator).

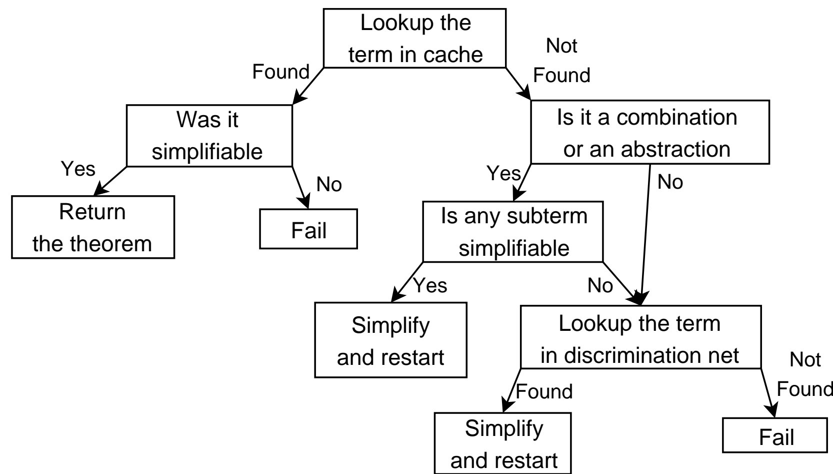


Figure 1.2: Our implementation of *the CAS conversion* first tries to look up the term in a built-in cache (for efficiency). If the term is an application or an abstraction, then it tries to simplify subterms recursively (not performed if the term is known not to be expandable or is a suggestion that should not be expanded, for example `NUMERAL` or `assuming`). Finally it tries to apply all conversions from the knowledge base to the term.

The file that defines the CAS conversion includes:

- The `PROGRESS_DEPTH_CONV` conversion which is a version of a depth-first conversion, that traverses a term like `DEPTH_CONV`, by unfolding applications and abstractions and trying the given conversion at all levels in a term. The difference is that it additionally checks for progress on the level of the whole term, and does not fail if the conversion fails only at one level. The conversion is implemented both in the top-down and bottom up priority versions.
- A discrimination net to be used for the stored conversions. Since the knowledge of the CAS system can be independent from the CAS con-

version, we will describe the usage of the discrimination net and adding knowledge to it in the next section. We also provide functions that allow storing particular conversions or conversions with associated patterns in the discrimination net, as well as adding particular theorems for rewriting. A conversion for handling conditional rewrite rules is provided with the discrimination net. We will describe how theorems given as implications are given to this conversion in the next section.

- Finally the `cas_cache_conv` conversion which behaves as shown in Fig. 1.2. This is the main recursive function called by the CAS conversion. The main CAS conversion `cas_conv` calls this conversion and if it fails returns an instance of reflexivity.

1.3 CAS-like knowledge

The knowledge of a computer algebra system is substantially different from that of a proof assistant. Usually the main part of a CAS are algorithms that manipulate formulas, and the theory on which they are based is not represented inside the system. On the other hand, the primary part of a proof assistant is a large collection of definitions and theorems.

There are different possible approaches we implement to the creation of theorems that certify correctness of algorithms implementing CAS-style simplifications. In certain cases a theorem is built alongside with the term analyzed. Sometimes it is sufficient to do a calculation in OCaml code and only verify the result in the logic. For example a factorization of big integers or polynomials can be simply verified by multiplying them out. This is where our approach is similar to the "bridge" approach. The important difference is that an algorithm for a calculation *together* with a certification form a single conversion that can be introduced in the knowledge base.

1.3.1 Knowledge base

The knowledge base is a separate part of the system. The conversions are kept in a discrimination net (a structure that allows matching a term to a number of patterns efficiently). There is an interface on the theorem prover level that allows introducing knowledge to the knowledge base in the following three forms:

- Rewrite rules, for example:
  `|- ∀z.abs (norm z) = norm z`
- Conditional rewrite rules, for example:
  `|- ∀x.&0 <= x ==> abs x = x`
- Conversions that are able to simplify terms of a particular form. Those are used only on arguments that match a particular pattern and return an ad-hoc rewrite rule. An ad-hoc rewrite rule is a theorem that is generated to be used for rewriting the formula, but is not intended to be kept

in the knowledge base as such (although our implementation keeps all theorems used for rewriting in a cache, implemented as a hash-table, for efficiency reasons). For example the HOL Light conversion `DIVIDES_CONV` takes terms that match the pattern `n divides m` and then returns ad-hoc rewrite rules for the given data like `|- 33 divides 123453 <=> T`.

The CAS conversion has to check whether the given term matches one of the rewrite rules and ad-hoc rewrite rules in the knowledge base. For efficiency it keeps all theorems and conversions included in the knowledge base in a discrimination net. To allow matching conversions with even less overhead, optional patterns for matching associated with conversions can be provided. The discrimination net is not changed, the particular used instances are only added to the cache.

1.3.2 Knowledge representation

To resemble a CAS system, we try to provide functions that perform the operations that are given only as predicates in the theorem prover. This way the formulas processed by the system can be in the "evaluation" form and not in "verification" form.

Let us compare the ways in which one writes differentiation in the HOL Light library and the way it is written in our CAS:

```
∀x. (f diffl (g x)) x      →      diff f = g
(f diffl (g x)) x          →      diff f x = g x
```

In HOL Light the `diffl` predicate takes three arguments: the function (on the left of the predicate), the value of its derivative and the point. To write a general derivative we need to generalize the point and replace the value with the derivative function in this point. Even then it is still a binary predicate.

In most computer algebra systems there exists a simple `diff` operator, that takes a function and returns its derivative. Using the Hilbert's choice operator, we created a such function, defined: `diff f = λx. εv. (f diffl v) x`. This means that `diff f` is defined and can be requested by the user, even if `f` is not differentiable. We also created a conversion that is able to calculate the derivative of a function, if HOL Light's `DIFF_CONV` can.

Just like we defined a functional form of differentiation, we also defined a functional integration operator. Using these we can then compute the following expressions in the system. In the following `dint (a,b) (λx.f(x))` denotes the definite integral $\int_a^b f(x)dx$:

```
  In9 := dint (&1,&2) sin
 Out9 := - &1 * cos (&2) + cos (&1)
  In10 := dint (&1,&2) (λx. x pow n)
 Out10 := &1 / &(n + 1) * &2 pow (n + 1) +
          - &1 * &1 / &(n + 1) * &1 pow (n + 1)
  In11 := diff (diff (λx. &3 * sin (&2 * x) + &7 + exp (exp x))) (&2)
 Out11 := exp (exp (&2)) * exp (&2) pow 2 + exp (exp (&2)) * exp (&2) +
          - &12 * sin (&4)
  In12 := diff (λx. dint (x,x + &2) (λx. x pow 3))
 Out12 := λx. &6 * x pow 2 + &12 * x + &8
```

If a function is not differentiable, or the system does not know how to differentiate it, then the conversion is not able to simplify it, and the input is returned unchanged. These differentiation and integration definitions do not work well with partial functions. An approach to defining them, so that partial functions are handled in a way that resembles computer algebra, will be described in the following chapter.

The file `cas_basic_convs.ml` includes a number of conversions that simplify commonly found computer algebra expressions as well as the differentiation mechanism. More precisely:

- Simplification of integer expressions and integer polynomials
- Computation of binomial coefficients
- Expansion of finite sums
- Computation of some common complex expressions
- Verification of primality
- Expansion of decimal notation
- A conversion for simplifying differentiation.

1.3.3 Numerical approximations

In complex calculations computer algebra systems provide users with numerical approximations. They are usually implemented with an approximation algorithm, which keeps an error bound with every calculation. In a proof assistant a numerical approximation must have its semantics completely defined, and the algorithm has to respect the approximation definition and theorems that specify its properties.

The two main ways of rounding a real number are down to an integer and towards the nearest integer. Both these operations are not computable functions (see for example [Les01]). This is a problem since in computer algebra, calculating an approximation is an operation that always terminates. Vuillemin [Vui88] shows that a function that computes non-deterministically either one of those values is a computable function. We will use a conversion that calculates the value rounded both down and to nearest value, that terminates when one of those calculations terminate.

We define the numerical approximation of a given number `x` to a precision `p` decimal points behind the dot as identical to the number itself: `N x p = x`. It is only a hint for the system that the number has to be simplified to a decimal fraction plus a rest. It is the rest, that determines in which form is the number given: rounded down or rounded to the nearest. For rest defined in this way we provide a theorem, that states that the approximation can be different from the exact value only on the last digit, and the difference is less than one.

In the following HOL Light definitions, `N` is the numerical approximation of a number to a precision (following the convention of Mathematica) and `...` is the rest of a number to the given precision with an additional argument that specifies the form of the rest. `T` stands for rounding to nearest and `F` stands for rounding down.

```
... x p F = x - floor (&10 pow p * x) / &10 pow p
... x p T = x - floor (&10 pow p * x + &1 / &2) / &10 pow p
```

We then prove a theorem that states the correctness of the definition of `...`:

```
|- abs(... x p T - x) < &1 / &10 pow p
```

The system is able to compute some numerical approximations with this scheme (it currently returns the approximation rounded to nearest only if the original number is already decimal):

```
  In13 := N (&1 / &3) 8
 Out13 := #0.33333333 + ... (&1 / &3) 8 F
  In14 := N (&1 / &2) 3
 Out14 := #0.500 + ... (&1 / &2) 3 T
  In15 := N (sqrt #5.123456789) 8
 Out15 := #2.26350542 + ... (sqrt #5.123456789) 8 F
  In16 := N (dint (#0.1,#0.4) exp) 7
 Out16 := #0.3866537 + ... (- &1 * exp #0.1 + exp (&2 / &5)) 7 F
```

The implementation loads the file `cas_evalf.ml` with the following functionality for approximations:

- The definitions of `N`, `...` and the theorem that proves, that approximations are close enough to the approximated number.
- A conversion that is able to round numbers in a used-defined fashion. This is used for simplification of `floor(...)`.
- A conversion that tries to evaluate the `N` terms that returns the approximation and the rest.

1.3.4 Assumptions

In most CASs it is possible to make type assumptions or logical assumptions about variables. Examples include assuming a variable to be greater than zero, greater than another variable, natural or real. There are various methods of introducing assumptions in computer algebra systems:

- Assumptions associated with a simplification in Mathematica: `Simplify[Sin[n Pi], Element[n,Integers]]`
- Global list of assumptions in Maple: `assume(x>0); sqrt(x*x);`
- Asking the user for conditions on variables (e.g. Maxima)
- Adding assumptions automatically and silently to the prover environment (e.g. MathXpert)

In our system we keep a global list of assumptions, which are Boolean properties that may be used later to instantiate assumptions of rewrite rules and ad-hoc rewrite rules. In a big CAS the number of rules that can be used is so big that asking the user seems not to be a good choice. Also automated assuming would not behave well with many possible assumptions.

An assumption can be introduced by the user either using `assume`, which takes a Boolean, or `assumetype` which takes a typed variable. An assumption associated with a single simplification of a sub-term may be also introduced using `assuming`. The latter method temporarily changes the assumptions list to simplifying the sub-expression. The assumptions will be added to the assumptions of the theorem generated by *the CAS conversion*, which is why changing the assumptions list is only useful at the top-level of the expression to simplify.

The global list of assumptions is used by the conversions from the knowledge base, therefore we consider is a part of the latter. To ensure the usage of variables with correct types, type checking has to have access to this list. When an expression is typed in the system it is type-checked in a particular context. This context includes types already assigned to all free variables from the assumptions list, as well as all variables for which types have been assumed with `assumetype`. To do this, the latter are kept in another global list.

For example, $\sqrt{x^2}$ cannot be simplified to x, since we don't know whether x is positive or not. Also $\frac{x}{x}$ cannot be simplified to 1, since it is possible that $x = 0$.

```
 In17 := sqrt (x * x)
Out17 := abs x
 In18 := x / x
Out18 := x * inv x
```

An assumption about x, which states that it is greater than 1, allows the simplification to find and proving certain numeric properties of x; which in turn allows both the above formulas to be simplified:

```
 In19 := assume (x > &1)
Out19 := T
 In20 := x > #0.5
Out20 := T
 In21 := sqrt (x * x)
Out21 := x
 In22 := x / x
Out22 := &1
```

There are two ways in which assumptions are used: directly and indirectly. The first way is to use an assumption directly in the derivation in unchanged form. It can be used to a prove a reflexive theorem or to fill the requirement of a certain conditional rewrite rule (or a conditional ad-hoc rewrite rule). An assumption may be used as an indirect step in the derivation, for example simplifying $abs(x)$ to x requires $x \geq 0$, and the assumption $x > 1$ can be used for this.

1.3.5 Manipulating assumptions

A CAS has to provide a mechanism for adding assumptions and listing defined assumptions. In our prototype we added the `assumptions` and `about` commands, which resemble their Maple equivalents.

```
Command: about Argument: x
  `x > &1`
```

In any approach it is hard to handle errors that may be caused by incorrect parsing and printing. We try to be as close as possible to the original HOL Light's parsing and printing mechanism. In fact, the system currently uses HOL's term printing (with special output for errors) but, when parsing, the system has to add typing information and distinguish commands from terms. Special output is added, so that the user always knows when a given string has been interpreted as a command.

To further lower the risk of parsing and printing problems, we add the `theorems` command. It allows printing all theorems defined in a session. The standard HOL Light theorem printing function is used for this. It is especially useful for conversions that use assumptions, since the assumptions that have been actually used to prove the theorem will be included in its assumptions and will be printed on the left side of `|-`. Below are the first five and last two theorems proved by the examples from this chapter:

```
Command: theorems
  |- (3 + 4 DIV 2) EXP 3 * 5 MOD 3 = 250
  |- vector [&2; &2] - vector [&1; &0] + vec 1 = vector [&2; &3]
  |- diff (diff (λx. &3 * sin (&2 * x) + &7 + exp (exp x))) =
       (λx. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
       - &12 * sin (&2 * x))
  |- N (exp (&1)) 10 = #2.7182818284 + ... (exp (&1)) 10 F
  |- 3 divides 6 ∧ EVEN 12 <=> T
 ...
  x > &1 |- sqrt (x * x) = x
  x > &1 |- x / x = &1
```

The last loaded file in our prototype implementation, `cas_add_convs.ml`, adds all the conversions defined in the previous sections to the knowledge of the CAS conversion. We give below parts of the code provided in this file:

```
cas_pattern `SUC(NUMERAL n)` NUM_SUC_CONV;
```

```
cas_pattern ‘PRE(NUMERAL n)‘ NUM_PRE_CONV;
cas_pattern ‘FACT(NUMERAL n)‘ NUM_FACT_CONV;
cas_pattern ‘NUMERAL m < NUMERAL n‘ NUM_REL_CONV;
...
cas_pattern ‘int_add x y‘ cas_int_conv;
cas_pattern ‘int_sub x y‘ cas_int_conv;
...
cas_pattern ‘diff f‘  diff_conv;
cas_pattern ‘sum range f‘  sum_conv;
cas_pattern ‘N x p‘  evalf_conv;
cas_pattern ‘binom (n,m)‘  binom_conv;
cas_pattern ‘n divides m‘   DIVIDES_CONV;
cas_pattern ‘prime n‘   prime_conv;;
...
cas_eq_thm POW_2_SQRT_ABS;
cas_eq_thm SQRT_0;
cas_eq_thm ETA_AX;
cas_eq_thm CX_DEF;
cas_eq_thm SELECT_LEMMA;
...
cas_impl_thm abs_thm;
cas_impl_thm abs_neg_thm;
cas_impl_thm REAL_MUL_LINV;
```

1.4 Concluding remarks

Our work integrates computer algebra and proof assistant technology. We will now look at how our architecture compares with what one gets by just having a CAS or a proof assistant.

Developing a system according to our architecture (i.e., where the algorithms not only generate the results, but also generate certificates of the correctness of those results) will be slower than the development of traditional CAS systems (because that only has to generate the results). As far as the performance of the system is concerned, our architecture will be slower than a traditional CAS as well. This is mostly because generating the certificates for all simplifications is time consuming. However, we expect this slow-down over traditional CAS to only multiply the running time by a constant factor. Our expectation is not experiment based, but based on the architecture, we trace what a traditional CAS does and provide proofs for every step.

When we compare our architecture to the way that one normally does CAS-like manipulations in an interactive theorem prover, the main difference is the interaction model. Our CAS system does not interactively work on propositions that are to be proved, but instead takes an expression and automatically simplifies it.

A feature that we plan to investigate are the coercions that many proof

assistants use, like the embedding of the integers in the real numbers or the real numbers in the complex numbers. Currently a user of our prototype needs to use the '`&`' and '`Cx`' symbols for this (as is customary in the HOL Light library). A small improvement to the current situation might be to overload the '`&`' operator, but we would rather not make the user write these functions at all.

An issue that our approach does not cover is completeness of the conversions. In the case of rewrite rules the completeness is clear. But in the case of arbitrary algorithms, it is not guaranteed by our architecture that a given conversion will always terminate and never fail.

We presented an architecture for a certified computer algebra system, and addressed some issues that were encountered when using this architecture, like numerical approximations and variable assumptions. We also presented the details of a prototype of our architecture that was implemented on top of a state-of-the-art proof assistant. Although the prototype is not a powerful CAS at the moment, we believe it can be extended into one, by extending the knowledge base that the system uses and by providing more automatic simplification algorithms.

We believe that both computer algebra systems and proof assistants currently have a problem. In computer algebra the lack of explicit semantics and the lack of verification of the results inside the system makes the systems less reliable than one would like them to be. In proof assistants the powerful symbolic manipulations that are taken for granted in computer algebra often are missing and, even when present, it takes work and expertise to make use of them.

We claim that the architecture that we present here solves both problems simultaneously. The computer algebra systems get explicit semantics and certification and the proof assistants get CAS-like functionality that makes them more powerful and easier to use than they are today.

Chapter 2

Automating the generation and verification of side conditions in formalized partial functions

2.1 Introduction

2.1.1 Motivation

Partiality is an important and difficult issue in formalizing mathematics. Mathematicians tend not to write explicit proofs about partial functions having their arguments in their domains. It is common to see expressions that include terms like:

$$\dots \frac{1}{x} \dots$$

without defining what the variable x is or giving any assumptions about it. On the other hand these assumptions are necessary in proof assistants. Since most proof assistants are total frameworks, a similar formula expressed there usually looks like:

$$\forall x \in \mathbb{R}. x \neq 0 \Rightarrow \dots \frac{1}{x} \dots$$

These assumptions are obvious for any mathematician, in fact they can be generated by an algorithm. All names that have not been defined previously are considered to be universally quantified variables and all applications of partial functions give rise to precondition assumptions about their arguments. Inferring the types of variables is something that proof assistants are already good at, usually giving the type of just one of the terms in an expression is enough to infer the types of the others[1].

[1] Some proof assistants allow prioritizing a type, that gets inferred automatically.

There are many examples of statements in libraries of theorems for proof assistants that include assumptions which are often omitted in mathematical practice. In particular the HOL Light library part concerning real analysis includes statements like `EXP_LN`:

$$\forall x.0 < x \Rightarrow exp(ln(x)) = x$$

Here the type of x is inferred automatically as `real` from the type of the functions (the complex versions of the exponent and logarithm functions have different names in the library), but the domain conditions are not taken care of. The real logarithm is defined only for positive numbers, so the positivity assumption is required in statements of the theorems that talk about the logarithm, as well as in proofs every time the logarithm is used.

Computer algebra systems allow applying partial functions to terms and some of them have assumptions about variables computed automatically. This might be one of the reasons why computer algebra systems are usually more appealing than proof assistants for mathematicians. Unfortunately the way assumptions are handled in those systems is often approximate. This can be for example since assumptions were added in the later stage of the development of a particular CAS and the checking that assumptions hold is added on a per-algorithm basis. This is one of the reasons computer algebra systems sometimes give erroneous answers. Therefore handling assumptions cannot be done in the same way in theorem proving.

Our work on implementing a prototype computer algebra system in HOL Light presented in the Chapter 1 has shown that proof assistants are already able to perform many simplifications that one would expect from computer algebra. The prototype is able to perform many computations that involve total functions, but even the simplest operations that require understanding partiality fail, since HOL Light is a total framework:

```
 In1 := diff (diff (\x. &3 * sin (&2 * x) + &7 + exp (exp x)))
Out1 := \x. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
        -- &12 * sin (&2 * x)
 In2 := diff (\x. &1 / x)
Out2 := diff (\x. &1 / x)
```

The problem with the above example is that the function $\frac{1}{x}$ is partial and not defined at zero in HOL Light. Still, computer algebra systems asked for the derivative of it reply with $\frac{-1}{x^2}$, since the original function is differentiable on the whole domain where it is defined, and its derivative has the same domain. Our approach will let the framework compute correctly this kind of expressions.

Finding an approach to handling partiality in an automated way might be useful not only in formalizing partiality but also in formalizing functions that operate on more complicated data structures, such as formalizing multivaluedness.

2.1.2 Approach

Our approach is to let the user type the partial functions as functions on the option type and show them to the user as such, but to perform all operations on the total functions of the underlying proof assistant while keeping the domain predicate alongside with the function. To do this we will have two representations for functions and convert between them. The first representation is as functions of option types and the second is as pairs of total functions and domain predicates. We will show how higher order functions (differentiation) can be defined in this framework and how terms involving it can be treated automatically.

2.1.3 Related work

There are multiple approaches and frameworks for formalizing partial recursive functions. Ana Bove and Venanzio Capretta [BC05] introduce an approach to formalizing partial recursive functions and show how to apply it in the Coq proof assistant. Normally recursive functions are defined directly using `Fixpoint`, but that allows only primitive recursion. A general recursive definition gives rise to an inductively defined predicate whose constructors characterize the allowable recursive calls of the function. A total recursive function may then be defined by `Fixpoint` by recursion on proofs of this predicate.

Alexander Krauss [Kra06] has developed a framework for defining partial recursive functions in Isabelle/Hol, that formally proves termination by searching for lexicographic combinations of size measures. William Farmer [Far99] proposes a scheme for defining partial recursive functions and implements it in IMPS. The main difference is that those approaches and frameworks compute the domains of partial recursive functions whereas we concentrate on functions in analysis which cannot be obtained by recursion and where the domain is limited because there are no values of the functions that would match their intuitive definition or that would allow properties like continuity. Therefore we need to determine the domains of composed functions.

The existing libraries for proof assistants contain formalized properties of functions in real and complex analysis. There are common approaches to partiality in existing libraries. It is common to define every function total. This is the case for the HOL Light [Har96a] library. Division is defined to return zero when dividing by zero. The resulting theory is consistent, only some theorems have to include additional assumptions. For example REAL_DIV_REFL:

$$\forall x. x \neq 0 \Rightarrow \frac{x}{x} = 1.$$

Another common approach is to require proofs that arguments applied to partial functions are in their domains. This is the case for the CoRN library [CFGW04] of formalized real and complex analysis for Coq. There division takes three arguments, the third one is a proof that the second argument is different from zero.

There are approaches to include partiality in the logic of the proof assistant. Those unfortunately complicate the logic very much and are already complicated for first order logics [WZ03]. Some proof assistants are based on logics that support partial functions. An example is PVS [ORS92] where partial functions are obtained by subtyping.

Olaf Müller and Konrad Slind [MS97] present an approach for lifting functions with the option monad that is closest to the one presented here. Their approach is aimed at partial recursive functions where computation of the domains of functions is not possible. Our approach is similar to applying the option monad to the real and complex values, but since particular functions need to have their domains reduced, we will explicitly compute and keep the domains of functions and be able to transform these values back to original ones.

2.1.4 Contents

This chapter is organized as follows: in Section 2.2 we give the basic definitions of the two representations of partial functions and we define the operations used to convert between those representations. We also show a simplified example of a computation with partial functions. In Section 2.3 we present the design decisions and the details of our formalization. We show the automation works and show its limitations. Finally in Section 2.4 we present concluding remarks about the results of the chapter.

2.2 Approach

2.2.1 Basic definitions

Our approach will involve two representations of partial functions. We will represent partial functions either as pairs of a total extension of the original function and a domain predicate or a function from an option type to an option type. The first representation will be used in all automated calculations and the latter will be used in the input and if possible in the output since it best resembles mathematical notation.

An option type is a type built on another type that has two constructors: one denoting that the variable has a value and one used for no value. In proof assistants they are usually written as `SOME` α and `NONE`. We adopt the convention to denote those with $\overline{\alpha}$ and $-$. To simplify reading of the types, variables of the option type will be denoted as z and real variables as x.

We will define two operations to convert between the two representations. Creating operations that work on the option type from the operations on the underlying proof assistant type is similar to applying the option monad (*bind* composed with *return*) to the underlying type. In fact this is equivalent to our approach for total functions. For partial functions we additionally require the desired domain so we create our own operation that will additionally require the domain predicate and check it in the definition. We define @ that converts

functions from the pair representation to the option representation (written as `papp` in the HOL Light formalization) and $@^{-1}$ that converts a function on the option type to a pair (`punapp` in HOL Light). The definition of @ is straightforward:

$$(f, D)@z = \begin{cases} \overline{fx} & \text{if } z = \overline{x} \wedge D(x) \\ - & \text{otherwise} \end{cases}$$

The inverse operation can be defined using the Hilbert operator (which we will denote as ϵ). This operator takes a property and returns an element that satisfies this property. The inverse operation is defined as:

$$@^{-1}f = (\lambda x.\epsilon v.f(\overline{x}) = \overline{v}, \lambda x.\exists v.f(\overline{x}) = \overline{v})$$

The $@^{-1}$ function is the left inverse of @, (in fact we prove this in our formalization)[2]:

$$@^{-1}(\lambda z.(f, D)@z) = (f, D)$$

With the two operations definitions of the translations of the standard arithmetic operations are simple:

$$a + b =_{\text{def}} @(\lambda xy.(x + y), \lambda xy.\top)$$

We can also define higher order functions that operate on partial functions by embedding the existing higher order operators from the proof assistant, first in the pair representation:

$$\begin{aligned} (f, D)' &=_{\text{def}} (f', \lambda x.D(x) \wedge f \text{ is differentiable in } x) \\ f'(z) &=_{\text{def}} (@^{-1}(f)')@z \end{aligned}$$

2.2.2 Example in mathematical notation

With the definitions from the previous section it is possible to automatically simplify the side conditions in partial functions, we will first show it in the example and then show the full HOL Light definitions and the algorithm for simplification in Section 2.3.2.

We will show a simplified example of automatically computing a derivative of a partial function in our framework. We will denote the derivative of a function $f(x)$ as $f(x)'$. The user types an expression:

$$(\lambda z.\pi z^2 + cz + \frac{2}{z})'$$

The expression that the user sees is written with standard mathematical operators. All the operator symbols are overloaded, and they are understood as the operations on partial functions. In the above expression z is a variable of the option type. Most constants and expressions are their translations from the

[2] The $@^{-1}$ function is not the right inverse of @, since $(@^{-1}f)@\text{NONE} = \text{NONE}$ but it is not true that $f(\text{NONE}) = \text{NONE}$ for an arbitrary function f.

underlying total functions or constants. The only partial functions are division and differentiation that are defined in a special way. The translation of all operators and constants is unfolded, and a total function and its domain are computed[3]:

$$(\lambda z.\langle \lambda x.\pi x^2 + cx + \frac{2}{x}, \lambda x.x \neq 0\rangle @ z)'$$

We finally translate the derivative. For the obtained function we add the requirement that the derivative of the original function exists in the given point, otherwise a function defined in one point would always be differentiable there. This domain condition will be often quickly combined with the assumptions about the domain of the original function:

$$\lambda z.(\langle(\lambda x.\pi x^2 + cx + \frac{2}{x})', \lambda x.x \neq 0 \wedge (\lambda x.\pi x^2 + cx + \frac{2}{x})\text{is differentiable in } x)\rangle @ z)$$

We can then apply the decision procedure for computing derivatives of total functions in the underlying proof assistant. This is possible because the definition of @ ensures that the result does not depend on the function outside its domain. Since we also know the set on which the reciprocal is differentiable the domain can be simplified:

$$\lambda z.(\langle(\lambda x.2\pi x + c - \frac{2}{x^2}), \lambda x.x \neq 0\rangle @ z)$$

Finally we try to return to the partial representation. This is done by reconstructing a partial function with the same symbols and recomputing its domain.

$$\lambda z.2\pi z + c - \frac{2}{z^2} = \lambda z.(\langle(\lambda x.2\pi x + c - \frac{2}{x^2}), \lambda x.x \neq 0\rangle @ z)$$

Since the domains agree we can convert back and display the left hand side of the above equation as the final result to the user.

Returning from the representation of the function as a total function and its domain to the option type representation is not always possible, since a partial expression does not need to have an original form. On the other hand the simplification is often possible and when it is possible it is desired since it allows for greater readability. An example where it is not possible is:

$$\lambda z.\frac{1}{z} - \frac{1}{z} = \lambda z.(\langle \lambda x.0, \lambda x.x \neq 0\rangle @ z)$$

but it is not equal to the constant function zero, since it does not have a value when x is zero. Furthermore for values of the option type even $y - y$ is not equal to zero if y does not have a value, therefore even after simplification to zero its value will depend on the variable y.

There are two approaches of treating this kind of terms. One can either simplify it to zero leaving the domain condition or not simplify the expression

[3] In some proof assistants all computation is really simplification done by rewrite rules. This is the case in HOL Light in which we will be formalizing this example, but we will refer to those simplifications as computation in the text.

at all. We currently do not simplify expressions for which we cannot find a valid partial representation to return to. This is to avoid showing the user the complicated representation with the domain conditions. A possible approach that allows those simplifications and displays results in the option representation will be mentioned in Section 2.4.

2.3 The formalization

2.3.1 Design decisions

For our formalization we chose HOL Light. The factors that influenced our choice were: a good library of real and complex analysis, as well as the possibility to write conversions in the same language as the language of the prover itself. HOL Light is written in OCaml and is provided as an extension of it. This is very convenient for developing since it allows generating definitions and simplification rules by a programs an immediately using them in the prover.

In the representation with option types we use the vector type $\mathbb{R}^n \to \mathbb{R}$ instead of the curried types $\mathbb{R} \to \mathbb{R} \to \ldots \to \mathbb{R}$ to represent functions. One can convert between these two representations in most proof assistants and the latter representation is often preferred since it allows partial application. The reason why we chose to work with the vector representation is that HOL Light does not have general dependent types. Instead it has a bit less powerful mechanism that only allows proving theorems that reason about $\mathbb{A}^n$ for any n. We will use this to prove theorems about n-ary functions. With this approach some definitions (`papp` mentioned below and its properties) will have to be defined for multiple arities. On the other hand the theorems that are hard to prove only have to be proved once; if we used curried functions, they would have to be proved for all versions of currified functions that occur.

2.3.2 HOL Light implementation details

In this section we will give the formalization details, and to understand them the knowledge of basic HOL Light [Har96a] definitions is required. We will show an example of automatically computing the derivative of the partial function

$$f(x) = \pi x^2 + cx + \frac{2}{x}.$$

When the user inputs this function in the correct syntax in the main loop of the CAS, the system responds with the correct answer:

```
 In1 := pdiff (\x. SOME pi * x * x + SOME c * x + & 2 / x)
Out1 := \x. & 2 * SOME pi * x + SOME c + --& 2 / (x * x)
```

The system computed this derivative automatically, but we will look at the conversions performed step by step. First let us examine the types in the entered expression. The variable x used in the function definitions is of the type

`(real)option`. We overload all the standard arithmetic operators to their versions that take arguments of the `(real)option` type and produce results of this type. The coercion from naturals operator `&` creates values of this type. The coercion from the `real type` still needs to be written as `SOME`, since further overloading of `&` would lead to ambiguity.

The semantics of the standard arithmetic operations is to return a value if all arguments have a value and `NONE` if any of the arguments is `NONE`. For real partial functions we define an operation (called `papp`) that will create a partial function of type `(real)option` → `(real)option` → ... → `(real)option` from a pair of a HOL Light total function `real` → `real` → ... → `real` and a predicate expressing its domain `real` → `real` → ... → `bool`. We show below the definitions of `papp` for one and two variables. In the formalization we see them as `papp1`, `papp2`, ..., but in the text we will refer to all those definitions together as `papp`:

```
new_definition `(papp1 (f, d) (SOME x) = if (d (lambda i.x)) then
 (SOME (f (lambda i.x))) else NONE) /\
 (papp1 ((f:A^1->A), (d:A^1->bool)) NONE = NONE)`

new_definition `(papp2 ((f:A^2->A), (d:A^2->bool)) (SOME x) (SOME y) =
 if (d (lambda i.if i = 1 then x else y)) then
   (SOME (f (lambda i.if i = 1 then x else y))) else NONE) /\
 (papp2 (f, d) NONE v = NONE) /\ (papp2 (f, d) v NONE = NONE)`;;
```

In the above definitions we see the usage of `lambda` and `$`. Those are used to create vectors and refer to vector elements. The reasons for using the vector types instead of curried type for functions was discussed in Section 2.3.1.

The total binary operations can be defined by applying a common operator, that defines binary operators in terms of `papp` for two variables. The types of all defined binary operations is `(real)option` → `(real)option` → `(real)option`. We show only the definition of addition on partial values:

```
new_definition `pbinop (f:A->A->A) x y =
  papp2 ((\x:A^2. (f:A->A->A) (x$1) (x$2)),(\x:A^2.T)) x y`;;

new_definition `padd = pbinop real_add`;;
```

The first partial function is division defined in terms of the reciprocal.

```
new_definition `pinv = papp1 (partial ((\x:real^1. inv (x$1)),
  \x:real^1. ~((x$1) = &0)))`;;

new_definition `pdiv x y = pmul x (pinv y)`;;
```

`pdiff` is the unary differentiation operator. It takes partial functions of the type `(real)option`→`(real)option` and returns functions of the same type. Since the derivative may not always exist it is defined using the Hilbert operator. Given a (partial) function it returns a partial function being a derivative of the given one on the intersection of its domain and the set on which it is differentiable. We will again define it in terms of `papp` applied to a total function

and its domain. Since we are given a function and need to find its underlying total function and domain to apply the original differentiation predicate we will define `punapp` that returns this pair. For our definition it returns a pair of `real`→`real` and `real`→`bool`:

```
new_definition `punapp1 f = ((\x:real^1. @v:real. (f(SOME (x$1))) =
  (SOME v)), (\x:real^1. ?v. (f (SOME (x$1))) = (SOME v)))`;;

new_definition `pdiff_proto (f:real^1->real, d:real^1->bool) =
  ( (\x:real^1. if d x /\ ?v. ((\x. f (lambda i. x)) diffl v) (x$1)
  then @v. ((\x. f (lambda i. x)) diffl v) (x$1) else &0) ,
  (\x:real^1. d x /\ ?v. ((\x. f (lambda i. x)) diffl v) (x$1)) )`;;

new_definition `pdiff f = papp1 (pdiff_proto (punapp1 f))`;;
```

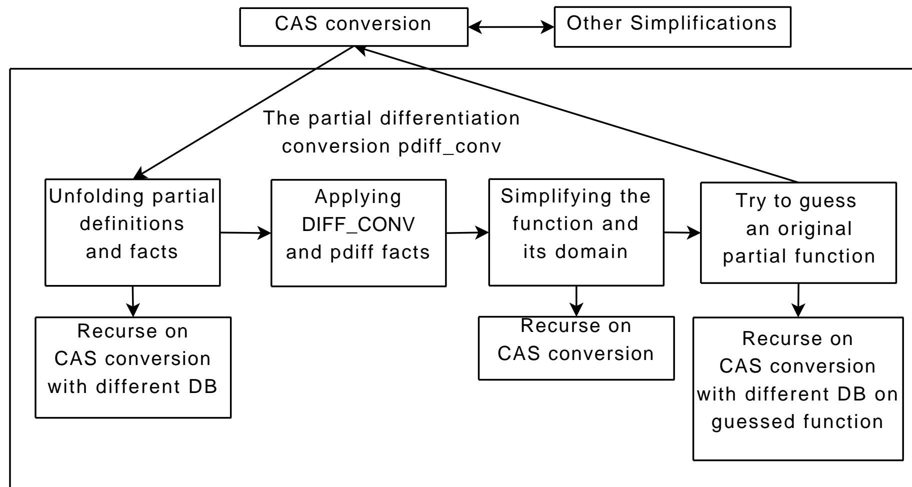


Figure 2.1: A schematic view of the simplification performed by the partial differentiation conversion.

The simplification of the term will be performed by a partial differentiation conversion `pdiff_conv` (Fig. 2.1). This conversion is a part of the knowledge base of the CAS and will be called by the CAS framework when the term has a `pdiff` term in it. To simplify the implementation of the partial differentiation conversion it will recursively call the CAS conversion to simplify terms. The first step is a simplification performed by the main CAS conversion with the database of theorems extended with above definitions of the partial operators and some basic facts, that will be described below. The conversion proves:

```
|- pdiff (\x. SOME pi * x * x + SOME c * x + & 2 / x) =
papp1 ((\x. @v. ((\x. x pow 2 * pi + c * x + &2 * inv x) diffl v) (x$1)),
(\x. ~(x$1 = &0) /\ (?v. ((\x. if ~(x = &0)
```

```
then x pow 2 * pi + c * x + &2 * inv x else @v. F) diffl v) (x$1))))
```

All the partial operators and the `pdiff` operator were unfolded to their definitions. We notice that the partiality included in division (reciprocal) and differentiation have been propagated to the term. All occurrences of variables are pulled inside the `papp` terms and consecutive `papp` applications are combined by a set of reduction rules. This set includes a number of theorems, here we give only single examples for one variable:

- rewrite rules that reduce the number of `papp` applications for `SOME` terms for arbitrary numbers of variables. An example for the second of two variables:

```
# papp2_beta_right;;
val it : thm = |- papp2 (f, d) (a:(A)option) (SOME b) =
  papp1 ((\x. f (lambda i. if i = 1 then x$1 else b)),
  (\x. d (lambda i. if i = 1 then x$1 else b))) a
```

- rewrite rules that combine multiple occurrences of the same variable:

```
# papp2_same;;
val it : thm =
|- papp2 (f, d) x x =
   papp1 ((\x:real^1. f ((lambda i. x$1):real^2)),
     \x:real^1. d ((lambda i. x$1):real^2)) x
```

- rewrite rules that combine consecutive applications of `papp` possibly with different numbers of abstracted variables:

```
# papp1_papp1;;
val it : thm = |- papp1 (f1, d1) (papp1 (f2, d2) (x:(A)option)) =
   papp1 ((\x. f1 (lambda i.(f2 x))),
   (\x. d2 x /\ d1 (lambda i.(f2 x)))) x
```

The next step performed by the partial differentiation conversion extracts the function to which the `diffl` term is applied. The HOL Light `DIFF_CONV` is applied to this term. For total functions it produces a `diffl` theorem with no additional assumptions. For partial functions `DIFF_CONV` produces conditional theorems that have additional assumptions about the domain. For our example:

```
# DIFF_CONV `(\x. x pow 2 * pi + x * &c + &2 * inv x)`;;
val it : thm = |- !x. ~(x = &0) ==>
  ((\x. x pow 2 * pi + x * &c + &2 * inv x) diffl
  (((&2 * x pow (2 - 1)) * &1) * pi + &0 * x pow 2) +
  (&1 * &c + &0 * x) + &0 * inv x + --(&1 / x pow 2) * &2) x
```

Our formalization includes certain theorems about derivatives of partial functions where the derivative exists depending on some condition. For the example case the used theorem is about derivatives of functions that are not differentiable in a single point. We provide some similar theorems for inequalities which may arise in differentiating more complicated functions. The exact statement of the theorem used here is:

```
# pdiff_but_for_point;;
val it : thm = |- (!x. ~(x = w) ==> (f diffl (g x)) x) ==>
  papp1((\(x:real^1). @v. ((\x:real. f x) diffl v) (x$1)),
        (\(x:real^1). (~(x$1 = w) /\ d (x$1)) /\
     ?v. ((\x:real. if ~(x = w) then f x else @v. F) diffl v) (x$1))) =
  papp1((\(x:real^1). g (x$1)), \(x:real^1). ~(x$1 = w) /\ d (x$1))
```

The partial differentiation conversion combines the above facts to prove:

```
|- pdiff (\x. SOME pi * x * x + SOME c * x + & 2 / x) =
papp1 ((\x. (((&2 * x$1 pow (2 - 1)) * &1) * pi + &0 * x$1 pow 2) +
  (&0 * x$1 + &1 * c) + &0 * inv (x$1) + --(&1 / x$1 pow 2) * &2),
 (\x. ~(x$1 = &0)))
```

The above function can be easily simplified, and this simplification is performed by recursively calling the CAS conversion on the result (both on the function and on the domain). For our example only the function can be reduced. For the recursive call to the CAS conversion we do not include the facts about partiality to prevent looping. The conversion proves:

```
|- pdiff (\x. SOME pi * x * x + SOME c * x + & 2 / x) =
  papp1 ((\x. &2 * pi * x$1 + c + -- &2 * inv (x$1 * x$1)),
  (\x. ~(x$1 = &0)))
```

The last part of `pdiff_conv` tries to convert the term back to the original representation. As described is Section 2.3.1 this is not always possible, but it will be possible in our case. The algorithm for computing the original form examines the tree structure of the total function and reconstructs a partial function with the same structure. In our case:

```
# pconvert `(&2 * pi * (x:real^1)$1 + c + -- &2 * inv (x$1 * x$1))`;;
val it : term = `& 2 * SOME pi * x + SOME c + --& 2 * pinv (x * x)`
```

We now check if the domain of the guessed partial function is the same as the original real one. To do this we apply the CAS conversion to the guessed term with the partial function definitions and facts about them again:

```
# cas_conv it;;
val it : thm = |- & 2 * SOME pi * x + SOME c + --& 2 * pinv (x * x) =
  papp1 ((\x. &2 * pi * x$1 + c + -- &2 * inv (x$1 pow 2)),
   (\x. ~(x$1 pow 2 = &0))) x
```

The domain of the converted function is the same as the domain of the function we that was computed by differentiation[4]. Therefore we can compose this theorem with the previous result arriving at the final proved theorem:

```
|- pdiff (\x. SOME pi * x * x + SOME c * x + & 2 / x) =
   (\x. & 2 * SOME pi * x + SOME c + --& 2 / (x * x))
```

And the user is presented with the right hand side of the equation.

2.3.3 How to extend the system

In this section we will show examples that the system cannot handle automatically. We will then show how the user can add theorems to the knowledge base to add automation for simplification of those terms. Consider adding the real square root as a new partial function:

```
new_definition ‘psqrt = papp1 ((\x. sqrt (x$1)), (\x. (x$1) >= &0))‘;;
```

The original HOL Light differentiation conversion `DIFF_CONV` is able to differentiate the real square root producing a differentiation predicate with a condition:

```
# DIFF_CONV ‘\x. sqrt x‘;;
val it : thm =
|- !x. &0 < x ==> ((\x. sqrt x) diffl inv (&2 * sqrt x) * &1) x
```

The partial differentiation conversion cannot simplify the derivative of the partial square root automatically without additional facts in its knowledge base. This is because the result of the original differentiation conversion is only a condition for the function to be differentiable. It does not prove that the function is not differentiable elsewhere (namely in zero). To be able to simplify this function the user needs to prove an additional theorem that would show that the function is differentiable if and only if the variable is greater than zero, namely:

```
|- (?v. ((\x. if x >= &0 then sqrt x else @v. F) diffl v) ((x:real^1)$1))
   = x$1 > &0
```

Adding this to the knowledge base allows the partial differentiation conversion to simplify automatically the partial square root function. This means that the conversion asked for the derivative of $\sqrt{x}$ returns the partial function $\frac{1}{2\sqrt{x}}$ defined for $x > 0$, that is: $(\frac{1}{2\sqrt{x}}, x > 0)$.

[4]The two domains can be expressed in a slightly different way, thus there may be some theorem proving involved to show that they are equal. In our implementation the only thing performed is the CAS conversion, that internally tries HOL Light decision procedures for reals and tautology solving.

2.4 Concluding Remarks

The presented approach and formalized framework allow the automation of small side-conditions. Simple expressions with partial functions can be simplified transparently to the user. More complicated partiality conditions still appear in the expressions. In this case the user can prove facts that would simplify the expressions.

The approach allows to see expressions that resemble mathematics as done by engineers in proof assistants. The language for writing equations and for calculations (rewriting in HOL Light) becomes simpler.

It can be useful for formalizing partial functions that we encounter in engineering books, for example in Abramowitz and Stegun [AS64] or in the NIST DLMF project [Loz03].

It would be interesting to how easy our approach can be extended to more complicated partial operations. For example with integration it is hard to check whether the result is in the domain. Of course even then our approach gives a response, but the existential expression in the result may be hard to simplify.

It is important to note, that the standard HOL Light equality is not aware of the option type, so any objects that do not exist will be equal. Defining an equality that is not true for `NONE` is possible, but leads to additional complexity of the expressions and will not be compatible with original HOL Light any more.

We would like to add more automation. All the simplifications that we perform can be done with functions of arbitrary number of variables. Those can be proved on the fly by special conversions. Our formalization currently has all simplifications rules proved for functions with at most two optional variables. Also the `papp` definitions for more variables and facts about them are analogous to their simpler version and their definitions can be created automatically by a ML function that calls HOL Light's definition primitives.

We are looking for a policy for simplifying expressions. Currently when an expression is simplified in the total representation, but we cannot find an original partial representation, the whole conversion fails and the expression is returned unchanged. In a CAS environment with assumptions about the domains of variables the same conversions would succeed. It would be therefore desirable to suggest assumptions about variables that would allow for further simplification of terms.

It would be most interesting to extend the presented approach to address multivaluedness. Since multivalued functions are quite complicated and the underlying definitions can get very big, they are rarely treated in proof assistants. On the other hand, the treatment of multivalued functions tends to be one of the common sources of mistakes performed by computer algebra systems [JN04].

Chapter 3

Computing with classical real numbers

3.1 Introduction

Coq is a proof assistant based on dependent type theory developed at INRIA [CDT08]. By default, it uses constructive logic via the Curry-Howard isomorphism. This isomorphism associates propositions with types and proofs of propositions with programs of the associated type. This makes Coq a functional programming language as well as a deduction system. The identification of a programming language with a deduction system allows Coq to reason about programs and allows Coq to use computation to prove theorems. Below in this Chapter we will mention two important distinctions that are relevant to the theory behind Coq, namely the Prop and Set universe distinction as well as the proof vs computation distinction.

Coq can support classical reasoning by the declaration of additional axioms; however, these additional axioms will not have any corresponding computational component. This limits the use of computation to prove theorems, since Coq cannot compute the normal form of an expression where the head is an axiom. There are theories that allow program extraction from classical proofs, like A-translation, but this has not been done for proofs involving real numbers.

At least two different developments of the real numbers have been created for Coq. Coq's standard library declares the existence of the real numbers axiomatically. This library also requires the axioms for classical logic. It gives users the familiar, classical, real numbers as a complete ordered Archimedian field.

The other formalization of the real numbers is done constructively in the CoRN library [CFGW04]. This library specifies what a *constructive real number structure* is, and proves that all such structures are isomorphic. These real numbers are constructive and there is one efficient implementation where real numbers can be evaluated to arbitrary precision within Coq.

In this chapter we show how to connect these two developments of the theory of the real numbers by showing that Coq's real numbers form a real number structure in CoRN. We do this by:

- Deriving some logical consequences of the classical real numbers (Section 3.2). Specifically, we formally prove the well-known result that sentences in Π^0_1 are decidable. Bishop and Bridges [BB85] call it the *principle of omniscience* and consider it the root of nonconstructivity in classical mathematics.
- Using these logical consequences to prove that the classical real numbers form a constructive real number structure (Section 3.3).
- Using the resulting isomorphism between classical and constructive real numbers to prove some classical real number inequalities by evaluating constructive real number expressions (Section 3.4).

3.1.1 The two universes of Coq

Coq has a mechanism for program extraction [Let02]. Programs developed in Coq can be translated into Ocaml, Haskell, or Scheme. If these programs are proved correct in Coq, then the extracted programs have high assurance of correctness.

To facilitate extraction, Coq has two separate universes: the `Set` universe, and the `Prop` universe (plus an infinite series of `Type` universes on top of these two). The `Prop` universe is intended to contain only logical propositions and its values are discarded during extraction. The types in the `Set` universe are computationally relevant; the values of these types make up the extracted code. In order to maintain the soundness of extraction, the type system prevents information from flowing from the `Prop` universe to the `Set` universe. Otherwise, vital information could be thrown away during extraction, and the extracted programs would not run.

The `Prop/Set` distinction will play an important role in our work. The logical operators occur in both universes. The following table lists some logical operations and their corresponding syntax for both the `Prop` and `Set` universes.

Math Notation	`Prop` Universe	`Set` Universe
$A \wedge B$	`A /\ B`	`A * B`
$A \vee B$	`A \/ B`	`A + B`
$A \to B$	`A -> B`	`A -> B`
$\neg A$	`~A`	*not used*
$\forall x : X.P(x)$	`forall x:X, P x`	`forall x:X, P x`
$\exists x : X.P(x)$	`exists x:X, P x`	`{ x : X | P x }`

One might think that proving that classically defined real numbers satisfy the requirements of a constructive real number structure would be trivial. It

seems that the constructive requirements be no stronger than the classical requirement for a real number structure when we use classical reasoning. However, Coq's `Prop`/`Set` distinction prevents a naive attempt at creating such an isomorphism between the classical and constructive real numbers. The difficulty is that classical reasoning is only allowed in the `Prop` universe. A constructive real number structure requires a `Set`-level existence in the proof that a sequence converges to its limit (see Section 3.3.1), but the theory provided by the Coq standard library only proves a classical `Prop`-level existence. It is not allowed to use the x given by the `Prop` existential:

```
exists x:X, P x
```

to fulfill the requirement of a set existential:

```
{ x : X | P x }.
```

There may be alternative ways to prove that the classical Coq reals form a constructive real number structure by completely ignoring the classical existence and extracting a witness from CoRN, but it still remains to be seen if this is feasible. We present our original solution that transforms the classical existentials provided by the Coq standard library into a constructive existential. This solution uses Coq's classical real number axioms to create constructive existentials from classical existentials for any Π^0_1 sentence (Section 3.2).

3.2 Logical Consequences of Coq real numbers

Coq's standard library defines the classical real numbers axiomatically. This axiomatic definition has some general logical consequences. In this section we present some of the axioms used to define the real numbers and then show how they imply the decidability of Π^0_1 sentences. The axioms of the real numbers cannot be effectively realized, so a decision procedure for Π^0_1 sentences is not implied by this decidability result.

3.2.1 The axiomatic definition of the real numbers

The definition for the reals in the Coq standard library asserts a set $\mathbb{R}$, the constants 0, 1, and the basic arithmetic operations:

```
Parameter R : Set.
Parameter R0 : R.
Parameter R1 : R.
Parameter Rplus : R -> R -> R.
Parameter Rmult : R -> R -> R.
...
```

A numeric literal is simply a notation for an expression, for example `20` is a notation for:

```
(R1+R1)*((R1+R1)*(R1+(R1+R1)*(R1+R1)))
```

In addition to the arithmetic operations, an order relation is asserted.

```
Parameter Rlt : R -> R -> Prop.
```

Axioms for these operations and relations define their semantics. There are 17 axioms. We show only some relevant ones; the entire list of axioms can be found in the Coq standard library. The properties described by the axioms include associativity and commutativity of addition and multiplication, distributivity, and neutrality of zero and one.

```
Axiom Rplus_comm : forall r1 r2:R, r1 + r2 = r2 + r1.
...
```

There are also several axioms that state that the order relation for the real numbers form a total order. The most important axiom for our purposes will be the law of trichotomy. We describe the reasons for its form (in particular why it is Set-based) in next subsection:

```
Axiom total_order_T : forall r1 r2:R,
  {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

Finally, there is an Archimedian axiom (where `IZR` is the obvious injection $\mathbb{Z} \to \mathbb{R}$) and an axiom stating the least upper bound property.

```
Parameter up : R -> Z.
Axiom archimed : forall r:R,
  IZR (up r) > r /\ IZR (up r) - r <= 1.

Axiom completeness :
  forall E:R -> Prop, bound E ->
    (exists x : R, E x) -> sigT (fun m:R => is_lub E m).
```

3.2.2 Decidability of Π^0_1 sentences

It is important to notice that the trichotomy axiom uses `Set`-style disjunctions. This means that users are allowed to write functions that make decisions by comparing real numbers. This axiom might look surprising at first since real numbers are infinite structures and therefore comparing them is impossible in finite time in general. The motivation for this definition comes from classical mathematics where mathematicians regularly create functions based on real number trichotomy. It allows one to define a step function, which is not definable in constructive mathematics.

This trichotomy axiom can be used to decide any Π^0_1 property. For any decidable predicate over natural numbers P we first define a sequence of terms that take values when the property is true:

$$a_n =_{\text{def}} \begin{cases} \frac{1}{2^n} & \text{if } P(n) \text{ holds} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

We can then define the sum of this infinite sequence, which is guaranteed to converge:

$$S =_{\text{def}} \sum_{n=0}^{\infty} a_n \tag{3.2}$$

The trichotomy axiom allows us to compare S with 2. It follows that if $S = 2$ then the predicate P hold for every natural number, and if $S < 2$ then it is not the case (the case of $S > 2$ is easily ruled out). Furthermore, this distinction can be made in `Set` universe.

We formalized the above reasoning in Coq and we obtained the following logical theorem.

```
forall_dec
   : forall P : nat -> Prop,
     (forall n : nat, {P n} + {~ P n}) ->
     {(forall n : nat, P n)} + {~ (forall n : nat, P n)}
```

This statement means that:

$$\text{dec } \Sigma^0_n \longrightarrow \text{dec } \Pi^0_{n+1},$$

and since Σ^0_0 is decidable this implies that Π^0_1 is decidable. To see why this implies the decidability of particular Π^0_1 sentence, consider an arbitrary Π^0_1 sentence φ. If φ is Π^0_0, then it is decidable by the basic properties of Π^0_0 sentences. Otherwise, if φ is of the form $\forall n : \mathbb{N}.\psi(n)$ where $\psi(n)$ is decidable.The above lemma allows us to conclude that φ is decidable from the fact that $\psi(n)$ is decidable.

Constructive indefinite description

We can extend the previous result by using a general logical lemma of Coq. The constructive indefinite description lemma states that if we have a decidable predicate over the natural numbers, then we can convert a `Prop` based existential into a `Set` based one. Its formal statement can be found in the standard library:

```
constructive_indefinite_description_nat
   : forall P : nat -> Prop,
     (forall x : nat, {P x} + {~ P x}) ->
     (exists n : nat, P n) -> {n : nat | P n}
```

This lemma can be seen as a form of Markov's principle in Coq. The lemma works by doing a bounded search for a new witness satisfying the predicate. The witness from the `Prop` based existential is only used to prove termination of the search. No information flows from the `Prop` universe to the `Set` universe

because the witness found for the `Set` based existential is independent of the witness from the `Prop` based one.

Classical logic (included by `Reals`) allows us to convert a negated universal statement into an existential statement in `Prop`:

```
not_all_ex_not
   : forall (U : Type) (P : U -> Prop),
     ~ (forall n : U, P n) -> exists n : U, ~ P n
```

By combining these theorems with our previous result, we get a theorem whose conclusion is either a constructive existential or a universal statement:

```
sig_forall_dec
   : forall P : nat -> Prop,
     (forall n : nat, {P n} + {~ P n}) ->
     {n : nat | ~ P n} + {(forall n : nat, P n)}
```

3.3 The construction of the isomorphism

In this section we briefly present the algebraic hierarchy present in CoRN (it is described in detail in [GPWZ02] and [CF04]). We show that the Coq reals fulfill the requirements of a constructive real number structure, and hence they are isomorphic to any other real number structure.

3.3.1 Building a constructive reals structure based on Coq reals

The collection of properties making up a real number structure in CoRN is broken down to form a hierarchy of different structures. The first level, `CSetoid`, defines the properties for equivalence and apartness. The next level is `CSemigroup` which defines some properties for addition. More structures are defined on top of each other all the way up to a constructive ordered field structure — `COrdField`. Up to this point trichotomy is not required. Finally, the `CReals` structure is defined on top of the `COrderedField` structure. The full list of structures is given below.

CSetoid	–	constructive setoid
CSemiGroup	–	semi group
CMonoid	–	monoid
CGroup	–	group
CAbGroup	–	Abelian group
CRing	–	ring
CField	–	field
COrdField	–	ordered field
CReals	–	real number structure

To prove that classical reals form a constructive real number structure, we created instances of all these structures for the classical real numbers (called

RSetoid, RSemigroup, etc.). For example, RSetoid is the constructive setoid based on Coq real numbers. The carrier is R, while standard Coq equality (equivalent to Leibnitz equality) and its negation are used as the equality and apartness relations. The proofs of the setoid properties of these relations are simple.

The basic arithmetic operations from Coq real numbers shown to satisfy all the properties required up to COrdField. The proofs of these properties follow straightforwardly from similar properties provided by Coq's standard library. For details, we refer the reader to CoRN source files [CoR09]. We present just the final step, the creation of the CReals structure based on the ordered field.

Two additional operations are required to form a constructive real numbers structure from a constructive ordered field: the limit operation and a function that realizes the Archimedian property. The limit operation is the only step where the facts about Coq reals cannot naïvely be used to instantiate the required properties. This is because the convergence property of limits for the Coq reals only establishes that there exists a point where the sequence gets close to the limit using the Prop based quantifier, whereas CReals requires such a point to exist using the Set based quantifier. One cannot directly convert a Prop based existential into a Set based one, because information is not allowed to flow from the Prop universe to the Set universe.

The goal that remains to be proved in Coq is to show that if for any ϵ there is an index in a sequence N such that all further elements in the sequence are closer to the limit value than ϵ. The related property from the Coq library is shown as hypothesis u.

```
e : R
e0 : 0 < e
u : forall eps : R,  eps > 0 -> exists N : nat,
    forall n : nat,
     (n >= N)%nat -> Rfunctions.R_dist (s n) x < eps
______________________________________(1/1)
{N : nat | forall m : nat,
  (N <= m)%nat -> AbsSmall e (s m[-]x)}
```

In order to prove this goal, we first reduce the Set based existential to a Prop based one using the constructive_indefinite_description_nat.

Applying this lemma to the goal above reduces the problem to the following:

```
e : R
e0 : 0 < e
u : forall eps : R,  eps > 0 -> exists N : nat,
    forall n : nat,
     (n >= N)%nat -> Rfunctions.R_dist (s n) x < eps
______________________________________(2/2)
exists N : nat, forall m : nat,
  (N <= m)%nat -> AbsSmall e (s m[-]x)}
```

This now follows easily from the hypothesis. However, we are also required to prove the decidability of the predicate:

```
______________________________________(1/2)
{(forall m : nat, (x0 <= m)%nat -> AbsSmall e (s m[-]x))} +
{~ (forall m : nat, (x0 <= m)%nat -> AbsSmall e (s m[-]x))}
```

This goal appears hopeless at first because we are required to prove the decidability of a Π^0_1 sentence. However, we can use the `forall_dec` lemma from the previous section to prove the decidability of this sentence. This possible since the property:

```
P(m) := (x0 <= m)%nat -> AbsSmall e (s m[-]x))
```

is decidable. This completes the proof that the classical real numbers form a constructive real number structure.

3.3.2 The isomorphism

Niqui shows in Section 1.4 of his PhD thesis [Niq04] that all constructive reals structures are isomorphic, the proof is present in CoRN as `iso_CReals`. The constructed isomorphism defines two maps that are inverses of each other and proves that the isomorphism preserves the constants 0 and 1, arithmetic operations and limits. More details can be found in [Niq04].

In order to use the isomorphism in an effective way, we need to show that the definitions of basic constants and the operations are preserved. Since the reals of the standard library of Coq are written as `R` and CoRN reals as `IR`, we called the two functions of the isomorphism `RasIR` and `IRasR`. From Niqui's construction, one obtains the basic properties of this isomorphism:

- Preserves equality and inequalities:

  ```
  Lemma R_eq_as_IR : forall x y, (RasIR x [=] RasIR y -> x = y).
  Lemma IR_eq_as_R : forall x y, (x = y -> RasIR x [=] RasIR y).
  Lemma R_ap_as_IR : forall x y, (RasIR x [#] RasIR y -> x <> y).
  Lemma IR_ap_as_R : forall x y, (x <> y -> RasIR x [#] RasIR y).
  Lemma R_lt_as_IR : forall x y, (RasIR x [<] RasIR y -> x < y).
  ...
  ```

- Preserves constants: 0, 1 and basic arithmetic operations: $+$, $-$, $*$. In the properties listed below we do not show the dual theorems that state the same facts for opposite translation `IRasR`. Those are easy to prove using the properties of `RasIR`.

  ```
  Lemma R_Zero_as_IR : (RasIR R0 [=] Zero).
  Lemma R_plus_as_IR : forall x y,
      (RasIR (x+y) [=] RasIR x [+] RasIR y).
  ...
  ```

An important difference between the definition of real numbers in the Coq standard library and in CoRN is the way partiality is handled. Partial functions are defined as total functions for the Coq reals, but their properties require proofs that the function parameters are in the appropriate domain. For example, division is defined as a total operation on real numbers; however, all the axioms that specify properties of division have assumptions that the reciprocal is not zero. This means that the term $\frac{1}{0}$ is some real number, but it is not possible to prove which one it is.

In CoRN, partial functions require an additional argument, the domain condition. Division is a three argument operation; the third argument is a proof that the divisor is apart from zero. Other partial functions, such as the logarithm, are defined in a similar way. We prove that this isomorphism preserves these partial functions. These lemmas require a proof that the arguments are in the proper domain to be passed to the domain conditions of the CoRN functions.

- Preserves the reciprocal and division for any proof:

```
Lemma R_div_as_IR : forall x y (Hy : Dom (f_rcpcl' IR) (RasIR y)),
  (RasIR (x/y) [=] (RasIR x [/] RasIR y [//] Hy)).
```

Niqui's theorem proves the basic arithmetic operations and limits are preserved by the isomorphism. However, the real number structure does not specify any transcendental functions. The existence of the transcendental functions follows from the axiomatization, but the actual definitions used in the axiomatizations do not need to be the same. Therefore it is necessarily to manually prove that these functions are preserved by the isomorphism. This is easy if the Coq and CoRN definitions are similar, but becomes difficult if the two systems choose different definitions for the same function. We thus prove some more properties that the isomorphism preserves:

- Preserves infinite sums:
  The proof that the values of the sums are the same requires the decidability of Π^0_1 sentences and `constructive_indefinite_description_nat`. The term `prf` is the proof that the series converges.

```
Lemma R_infsum_as_IR : forall (y: R) a,
  Rfunctions.infinit_sum a y -> forall prf,
  RasIR y [=] series_sum (fun i : nat => RasIR (a i)) prf.
```

- Preserves transcendental functions: *exp*, *sin*, *cos*, *tan*, *ln*

```
Lemma R_exp_as_IR : forall x,
    RasIR (exp x) [=] Exp (RasIR x).
Lemma R_sin_as_IR : forall x,
    RasIR (sin x) [=] Sin (RasIR x).
Lemma R_cos_as_IR : forall x,
    RasIR (cos x) [=] Cos (RasIR x).
```

```
Lemma R_tan_as_IR : forall x dom,
    RasIR (tan x) [=] Tan (RasIR x) dom.
Lemma R_ln_as_IR : forall x dom,
    RasIR (ln x) [=] Log (RasIR x) dom.
```

We finally prove that the isomorphism preserves the constant π. This was more difficult because the π in Coq is defined as the infinite sum

$$\pi_{Coq} =_{\text{def}} \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}, \tag{3.3}$$

while in CoRN π is defined as the limit of the sequence

$$pi_n =_{\text{def}} \begin{cases} 0 & \text{if } n = 0 \\ pi_{n-1} + cos(pi_{n-1}) & \text{otherwise} \end{cases} \tag{3.4}$$

$$\pi_{CoRN} =_{\text{def}} \lim_{n\to\infty} pi_n. \tag{3.5}$$

Both libraries contain proofs that the sine of π is equal to zero, and additionally that it is the smallest positive number with this property. Using these properties it is possible to show that indeed the two definitions describe the same number:

```
Lemma R_pi_as_IR : RasIR (PI) [=] Pi.
```

3.4 Computation with classical reals

3.4.1 Solving ground inequalities

O'Connor's work on fast real numbers in CoRN includes a semi-decision procedure (a decision procedure that may not terminate) for solving strict inequalities on constructive real numbers. With the isomorphism it is possible to use it to solve some goals for classical reals.

Consider the example of proving $e^{\pi} - \pi < 20$ for the classical real numbers. The difference between these numbers is very small, so the proof is hard without using numeric approximations:

```
______________________________________(1/1)
exp PI - PI < 20
```

Our tactic first converts the Coq inequality to a CoRN inequality by using the fact that the isomorphism preserves inequalities. Then it recursively applies the facts about the isomorphism to convert the Coq terms on both sides of the inequality and their corresponding CoRN terms. This is done with using a rewrite database and the autorewrite mechanism for setoids. The advantage of using a rewrite database is that it can be easily extended with new facts about new functions being preserved under the isomorphism. The disadvantage of this method is that the setoid rewrite mechanism is fairly slow in Coq 8.1.

```
______________________________________(1/1)
Exp Pi[-]Pi[<](One[+]One)[*]
              ((One[+]One)[*](One[+](One[+]One)[*](One[+]One)))
```

(Recall that, in Coq, the real number 20 is simply notation for $(1 + 1) * ((1 + 1) * (1 + (1 + 1) * (1 + 1))).$)

Once the expression is converted to a CoRN expression, the semi-decision procedure from CoRN can be applied (which itself uses another rewrite database to change the representation again). This semi-decision procedure may not terminate. If the two sides of the inequality are different, it will approximate the real numbers accurately enough to either prove the required inequality (or fail if the inequality holds in the other direction). If the two sides are equal, then the search for a sufficient approximation will never terminate.

The decision procedure for CoRN takes an argument which is used for the starting precision of the approximation. Setting it to an appropriate value can make search faster, if the magnitude of difference between the sides is known *a priori*. Our decision procedure also takes this an argument and passes it on to the CoRN tactic.

We have shown the intermediate step above for illustration purposes only. The actual tactic proves the theorem in one step:

```
Example xkcd217 : (exp PI - PI < 20).
R_solve_ineq (1#1)%Qpos.
Qed.
```

Automatic rewriting is not enough to convert partial functions like division and logarithm. The additional parameters needed in CoRN are the domain conditions. The tactic itself could be called recursively to generate the assumptions. Unfortunately Coq 8.1 cannot automatically rewrite inside dependent products, making the recursive tactic more difficult to create. The Coq 8.2's new setoid rewriting system will allow rewriting in dependent products, and we expect this to greatly simplify the creation of a recursive tactic.

3.4.2 Using facts about Coq reals in CoRN

The standard library of Coq contains more properties of real numbers than CoRN. It also contains more tactics, like `fourier` for solving linear constraints. By using the isomorphism the other way, it is possible to apply these tactics while working with CoRN. Using the isomorphism this way is controversial because using the classically defined real numbers means that the axioms of classical logic are assumed.

We will show how a goal that would normally be proved by the `fourier` tactic in Coq reals can be done in CoRN. We will show it on a very simple goal, but the example is illustrative:

$$x \leq y \Rightarrow x < y + 1. \tag{3.6}$$

The goal written formally in Coq is:

```
Goal forall x y:IR, (x [<=] y) -> (x [<] y [+] One).
```

After introducing the assumptions we can apply the isomorphism to the inequalities both in the assumptions and in the goal:

```
intros x y H; rapply IR_lt_as_R_back.
assert (HH := R_le_as_IR_back _ _ H).
```

This shows the following goal:

```
1 subgoal
x : IR
y : IR
H : x[<=]y
HH : IRasR x <= IRasR y
______________________________________(1/1)
IRasR x < IRasR (y[+]One)
```

Since the isomorphism preserves all the functions in the goal and assumptions, we can apply the facts to change the terms that include the isomorphism on the top of the term to terms that include the application of the isomorphism only on variables.

```
replace RHS with (IRasR y + IRasR One)
  by symmetry; rapply IR_plus_as_R.
replace (IRasR One) with 1. 2: symmetry; apply IR_One_as_R.

1 subgoal
x : IR
y : IR
H : x[<=]y
HH : IRasR x <= IRasR y
______________________________________(1/1)
IRasR x < IRasR y + 1
```

Now the `fourier` tactic is applicable:

```
fourier.

Proof Completed.
```

A similar transformation can be performed to use other facts and tactics from the Coq library.

3.5 Related Work

Melquiond has created a Coq tactic that can solve some linear inequalities over real number expressions using interval arithmetic and bisection [Mel08]. This

tactic is currently limited to expressions from arithmetic operations and square root, but could support transcendental functions via polynomial approximations. It has the advantage that it can solve some problems that involve constrained variables.

Many other proof assistants include facts about transcendental functions that could be used for approximating expressions that involve them. However there are few mechanisms for approximating real numbers automatically since to compute effectively this has to be done either by constructing the real numbers with approximation in mind, or by using special features of the proof assistant. The latter approach is used effectively for example in HOL Light, due to a close interplay between syntax and semantics. The way a real number expression is described in HOL Light can be analyzed and it can be used to prove an approximation.

The construction of HOL Light real numbers is described in [Har98]. The approximation mechanism of HOL Light is provided in `calc_real` (part of the distribution of the prover) as the `REALCALC_REL_CONV` conversion. This conversion uses the fact that terms are transparent and decomposes a term or goal into subterms, looking for underlying underlying real number operations or constants. Implementing approximation as a conversion means that it is not available as a function inside statements of theorems, but instead while proving a goal about real number expression it is possible to ask for an approximation of a particular closed term. The conversion uses rewriting to generate a theorem that approximates a particular term.

Obua developed a computing library for Isabelle. In his PhD [Obu08] he shows examples of computing bounds on real number expressions using computation rather than deduction.

Lester implemented approximation of real number expressions in PVS [Les08]. Results of real number functions are proved to have fast converging Cauchy sequences when their parameters have fast converging Cauchy sequences. Cauchy sequences for many real number functions are effective and can be evaluated inside PVS.

3.6 Concluding remarks

In this chapter we have formalized a proof that the axioms of Coq's classical real numbers imply the decidability of Π_1^0 statements. We used this fact to prove that these classical real numbers form a constructive real number structure. Then we used the fact the all real number structures are isomorphic to use tactics designed for one domain to solve problems in the other domain. In particular, we showed how to automatically prove a class of strict inequalities on real number expressions.

The lemmas showing the decidability of Π_1^0 statements have been added to the standard library and are made available in the 8.2 release of Coq. The isomorphism and the tactics used to prove inequalities over the Coq reals have been added to the CoRN library. They will are available with the version of

CoRN compatible with Coq 8.2.

We wish to extend our tactics to solve inequalities over terms that involve partial functions. This should be easier to do when CoRN uses the new setoid rewrite mechanisms available in Coq 8.2. Currently CoRN uses the old setoid rewrite mechanism which means that the translation of expressions from one domain to another is quite slow. We would like to investigate ways that this could be made faster. We would also like to automate the translation from CoRN expressions to Coq expressions so that CoRN can have its own `fourier` tactic assuming classical logic.

Part II

Interactive formalized math on the web

Chapter 4

Web Interfaces for Proof Assistants

4.1 Introduction

4.1.1 Motivation

Nowadays people are more and more accustomed to having a connection to the Internet all the time. Thus the network and the machines on the network become a part of the computer one uses. There is a tendency to make services available just by accessing certain web pages. In this way people do not need to install software on their computers any more. Examples include web interfaces to e-mail, calendars, chat clients, word processors and maps.

Commercial services are often available through web-interfaces. On the other hand, in the scientific domain, examples are not so abundant. In particular there were no real implementations of web interfaces for proof assistants.

To use a proof assistant, one needs to install some software. Often the installation process is complicated. For example to install Isabelle [NPW02], which is one of the most popular proof assistants, on a Linux system, one needs a particular version of PolyML, a HOL heap and Isabelle itself. To use an interface to access the prover, one needs ProofGeneral [Asp00] and one of the supported Emacs versions. Installing additional libraries for proof assistants is often even more complicated; in particular getting a version of Coq that will work with CoRN [CFGW04] requires compilation from sources of both.

It happens that computer scientists prefer to stick with installed old versions of provers, not to go through the same process to upgrade. Mathematicians may even stay away from computer assisted proving altogether, just because of the complexity of installation.

We want a fast interface, that is available with a web browser. We want to access various proof assistants and their versions, in a uniform manner, without installing anything, not even plugins. The interface should look and behave like

local interfaces to proof assistants.

We want the possibility to create web pages, that show tutorials and proofs, where the user can interact with the real system. The provider of the server may install patched versions of provers, allowing an easy way for the users to try out particular features. We want libraries for proof assistants to be available centrally, so that users who want to see them do not need to download or install anything. The interface should allow developing proofs and libraries centrally, in such a way that it could be used in a *wiki*-like [Dav04] environment.

4.1.2 Our Approach

The solution is a client-server architecture with a minimal lightweight client interpreted by the browser, a specialized HTTP server and background HTTP based communication between them. The key element of our architecture is the asynchronous DOM modification technique (sometimes referred to as *AJAX - Asynchronous JavaScript and XML* or *Web application*) [W3C08]. The client part is on the server, and when the user accesses the interface page, it is downloaded by the browser, which is able to interpret it without any installation.

The user of the interface, accessing it with the browser, does not need to do anything apart from reloading the page when a modification is done on the server. Every time the user accesses a prover, the version of the prover that is currently installed on the server is used. The user can access any of the provers installed on the server, even a prover which does not work on the platform from which the connection is made.

Saving the files on the central server allows accessing them from any location, by just accessing the interface's page with a web browser. A central repository simplifies cooperation in proof development, by replacing versioning systems like CVS, which keeps a remote and a local copy, by a *wiki-like* mechanism, where the only copy is the remote one.

Our approach is presented as an architecture to create web interfaces to proof assistants, but it is not limited to them. The problems solved are relevant to creating web interfaces programs that have a state, include an *undo* mechanism, and their interfaces can be buffer oriented. Our architecture may be applied for example to buffer oriented programming languages, like Epigram [McB04].

4.1.3 Related work

There have been some experiments with providing remote access to a prover. None of them allowed efficient access without installing additional software.

LogiCoq [Pot99] is a web interface to Coq [CDT08]. It offers a window where one can insert the contents of whole Coq buffer and submit them for verification. It sends the whole buffer with standard HTTP request and refreshes the whole page. Therefore one can work efficiently only with tiny proofs.

The web interface to the Omega system [BCF+97], requires the Mozart interpreter to be installed on the user's machine. The use of the web browser

is minimal, the whole interface is written in Mozart. Installation of Mozart is possible only for certain platforms which also makes the solution limited.

There are Java applets having built-in proof assistant functionality. Examples may include G4IP [Urb98] or Logic Gateway [Got05]. We believe that limiting provers to Java applets is undesirable. First, requiring users to have working Java applet functionality adds another dependency. Second, most of the current proof assistants are implemented in functional programming languages, and make use of the features provided by them, so allowing only Java would seriously limit the implementers.

Web interfaces related to proof assistants and displaying mathematics on the web are worth mentioning. In particular:

- Helm [APCS01] - (Hypertextual Electronic Library of Mathematics) A web interface that allows visualisation of libraries available for proof assistants.

- Whelp [AGC$^+$04] - A content based search engine for finding theorems in proof assistants libraries, that supports queries requiring matching and/or typing.

- ActiveMath [MBG$^+$03] - A web-based framework for learning mathematics that uses Java applets to communicate with a central server using OMDoc [Koh00].

There are many commercial web interfaces and frameworks that use asynchronous DOM modification in non scientific domains, examples of which were mentioned in the introduction.

The novelty of our architecture in comparison with existing web interfaces for theorem provers is that it permits the creation of an interface to a prover, that can look and behave very much like the ones offered by state-of-the-art local interfaces, but is available just by accessing a page with a web browser without installing any additional software, not even plugins. Because of the architecture, the network used to transfer information does not slow down the interaction. The experiment to use asynchronous DOM modification to create an interface to a proof assistants has never been tried before.

4.1.4 Contents

In the rest of the chapter we present the techniques for creation of web interfaces, that we will use (Section 4.2) and the internals of a local prover interfaces which we try to imitate (Section 4.3), followed by the presentation of the new architecture (Section 4.4) and a description of its security and efficiency (Section 4.5). We present our prototype (Section 4.6) and give more details about how the implemetation has been performed (Section 4.7). Finally we present some concluding remarks and present a vision of a complete system built according to our architecture (Section 4.8).

4.2 Asynchronous DOM Modification

As the web is becoming more commonly used, web page designers and browser implementers add new functionality to web pages. Text files have been replaced by hyper-linked files, later including images, language-specific and mathematical characters, styles and dynamic elements. The W3C Consortium, which is the organization responsible for the standardization of the Web, defines these elements in standards, and consequently they are implemented in a similar ways in all browsers.

Since the late nineties browsers have started supporting the following technologies relevant to ProofWeb: JavaScript, DOM [LHLHW+04] (*Document Object Model*) and XmlHttp [Web08]. Combined use of these three technologies has become popular in recent years, since they allow one to create responsive web interfaces. In this chapter we refer to the combined usage of these three technologies as "*Asynchronous DOM Modification*." One can find other names describing this technique, like *AJAX* or *Web Application*.

JavaScript is a scripting programming language, created by Netscape in 1995, for adding certain dynamic functionality to pages written in HTML. It has been quickly adopted by most browsers and nowadays it is supported even by some text mode browsers like w3m and Links, and mobile phone browsers. It is very often used on Internet websites.

DOM (Document Object Model) [LHLHW+04] is an API (Application programming interface) for managing HTML and XML documents that allows modifications of their structure and content. Recent browsers support W3C DOM accessibility by JavaScript. It is often used on web pages to add dynamic elements, for example drop-down menus or images that change when the mouse moves over them.

XmlHttp [Web08] is an API accessible by web browser scripting languages to transfer data to and from a web server. It internally uses HTTP requests. XmlHttp requests are sent to the server without the knowledge of the user of the web browser. For every XmlHttp request a callback has to be provided, to be executed when the response from the server is received. The sending of the request can be optionally asynchronous. XmlHttp has been available in most browsers for some time, and has been recently described in a W3C specification draft.

Asynchronous DOM modification is a web development technique that uses the three technologies described above to create responsive web interfaces. Such interfaces are web pages, where particular events (key presses and mouse movement) are captured by JavaScript events. The minimal client part encoded in JavaScript processes the local events, like menu opening or typing in a buffer. Events that require additional information from the server are sent as asynchronous XmlHttp requests. Since the request is done in the background, it does not interrupt the user from working locally. When the response arrives, it is used to modify the DOM of the page.

In comparison with classical web pages, the usage of asynchronous DOM modification makes it possible to send minimal information to the server, to

receive only the information required, and to refresh small parts of the web page. Network overhead and page refreshing are minimized, thus creating interfaces which work many times faster than classical web-based ones. This way, the interface can closely resemble local interfaces if network latency is reasonable. In case of high network latency, asynchronous requests allow the user to work locally, while additional data is requested.

Examples of usage of the asynchronous DOM modification are: web emails and calendars which operate within a single page, maps which download required parts as they are dragged, and web chat clients. Such web interfaces are supported by all standard web browsers, in particular all Gecko based browsers, Microsoft Internet Explorer versions from 5, Opera from version 8, Konqueror from version 3.2, Safari from version 1.2 and even Nokia S60 browser from version 3. It is not supported by text mode browsers and browsers for visually impaired people.

4.3 Generic Interface for Proof Assistants

In this section we describe the internals of local interfaces for proof assistants. We chose for this ProofGeneral [Asp00] for two reasons. First, it is a prover-independent interface to proof assistants. Second, it is popular, since it is universal and since it is built on the highly configurable Emacs text editor. There is an ongoing effort to make it available also with the Eclipse environment, but the usability of the new version is limited to a subset of Isabelle.

ProofGeneral's interface provides the user with two buffers: an editable buffer containing the proof script and the prover state buffer. ProofGeneral relies on the proof assistant to process the commands incrementally. It does not distinguish tactic-mode proofs from declarative-mode proofs. State changing and non-state-changing prover commands are distinguished to select only the relevant ones as a part of the proof script and to allow queries.

The interface colors keywords according to the above distinctions, and additionally marks parts of the buffer with a background color, to indicate the status of verification. There are three parts of the buffer. Possible states include: Expression that has been accepted by the prover, expression that is now being verified, and editable non-verified expression.

ProofGeneral provides the users with a proof replaying mechanism. The prover itself has to provide an *undo* mechanism. Users may choose a point in the buffer to go to, and ProofGeneral issues a number of proof steps and *undo* steps to the prover in order to reach that point. This is necessary since some provers forget the intermediate steps of older proofs.

ProofGeneral is responsible for providing the proof script from files on the disc to the prover and saving the proof script. Other disc operations that exist in some provers, like proof compilation, program extraction or automated creation of documentation are not handled by ProofGeneral.

ProofGeneral is implemented mostly in Emacs Lisp, and is strongly tied with the editor itself. It is easy to adapt ProofGeneral to new proof assistants, by

setting a number of variables. If this is not sufficient ELisp code can be used.

Other interfaces to provers offer mostly similar functionality. In some interfaces, like PCoq [ABPR01] or IsaWin, additional visualisation mechanisms are available, for example *term annotations*. Some of these mechanisms are not available in ProofGeneral; this limitation comes from the Emacs editor.

4.4 General Architecture

In this section we describe how we imagine a complete web interface for proof assistants. The part implemented in our prototype as well as the additional experiments that we did with it are described in Section 4.6. The two core elements of our architecture are: a specialized Web server and a communication mechanism (Fig. 4.1).

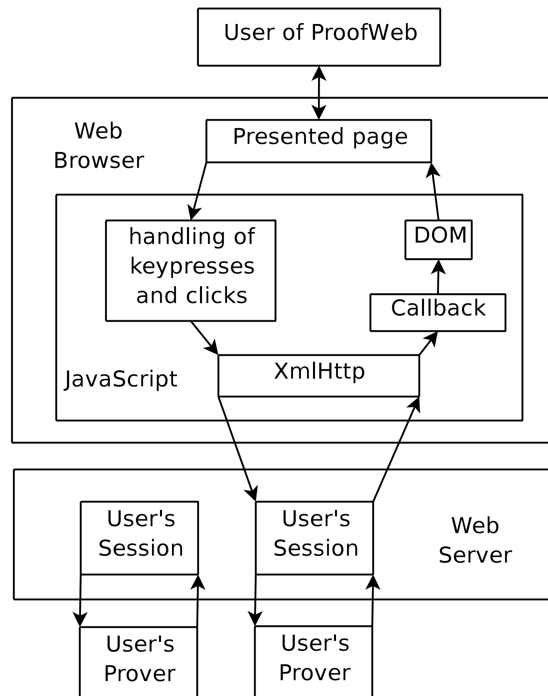


Figure 4.1: General architecture.

The Web server serves normal files and it is able to respond to special HTTP requests (see 4.4.2). The main interface is available as a normal HTML file on the server. When a user accesses the page with a browser, the page requires certain JavaScript files, which are then downloaded and interpreted by the browser. This serves as the client part.

The communication between the client part and the server is done with the mechanism described in Section 4.2. HTTP requests are created in the

background. The results are used to update the page in place. Only a small amount of information is transferred between the client and the server. The transfer is done asynchronously, making the interface responsive.

4.4.1 The Client Part

The client part offers a web page that initially presents the user with an editable buffer and an empty response buffer. (Also a menu or a toolbar is necessary for interaction, but they are normal elements of web pages). Buffers are implemented as HTML IFrames. An *IFrame* is an HTML tag that includes a floating frame within a page, that can be optionally editable. All keys that modify the IFrame are assigned to a special function. Locking of parts of the buffer is implemented by disallowing changes to locked parts of the buffer in this function.

When the user wants to verify a part of the buffer, this part is locked and sent to the server. Since the request is a background one, even if it takes a moment the user may continue working. When the response arrives, the contents of the two buffers are modified. The response may be a success, and then the part of the editable buffer is marked 'verified' and the response buffer shows the new prover state. If the command failed, the part of editable buffer is unlocked and the error shown. Parts of the editable buffer are marked, as their state changes, by using background colors, as it is done in ProofGeneral.

The interface includes a proof replaying mechanism, which behaves in a similar way to the one present in local interfaces. When the user wants to go to a particular place in the buffer, this information is passed to the server. The server sends the commands to the client's prover session and informs the interface about the results. In a similar way the interface includes a *break* mechanism that allows stopping the prover's computation.

The interface includes functionality for file interaction. Files can be loaded and saved on the server. For interoperability it is convenient if users can downloading files and uploading files from the local computer. For proof development efficiency, templates and queries may be inserted.

4.4.2 The Server Part

The server includes standard HTTP file serving functionality. With it the user's browser downloads the client part. The server can also handle special messages available for users, that have logged in. Session mechanism is used to support multiple clients even from the same IP address. A session is created when a user logs in to the system and is sustained with a cookie mechanism. Every user's session is associated with a particular prover session. The server runs provers as subprocesses and communicates with them through standard input and output. Prover sessions are terminated after a long period of inactivity (if the user did not close the page, the client part can replay the proof script from the beginning).

The special messages, mentioned above, include all communication that needs to be performed with the server. The necessary ones are: passing a given complete expression to verify to the prover, issuing an *undo* command in the prover, saving a file, loading a file, and *break* (stopping the prover computation). The commands from the client for the prover are passed first to the server, which transmits them to the prover. Prover replies are analysed by the server and only state changes are sent to the client. The state changes consist of two parts: changing of the markings of the edit buffer and the new contents of the prover state buffer.

Replies from the server are passed back to the client in an asynchronous way. This means, that the server does not answer HTTP requests from the client immediately, but when an answer from the prover is received or a timeout is reached. The server keeps a pool of provers that have been asked to process data, and waits for an answer from any of them. The waiting process does not block the server, that is, other clients' requests can be processed in the meantime.

4.5 Security and Efficiency

4.5.1 User side

All code that the user runs is interpreted within the web browser. Thus a malicious or virus infected prover can influence the client only by exploiting system or browser errors.

The efficiency of code execution on the user's side is dependent on the efficiency of the browser's internal web page and scripts interpretation, and the speed of HTML rendering.

Our experiments show that client-side DOM changes with Internet Explorer 6 are approximately twice as fast as with Mozilla Firefox 2 (still usually invisible for the user) both performed under Windows. It is hard to say whether this is due to less security checks or the worse quality of the rendered page (no anti-aliasing) in Internet Explorer.

4.5.2 Server side

In any centralized environment security, availability and efficiency of the server are important. Standard security measures include a backup server prepared to take over network traffic in case of a primary server failure and regular backing up of user files. In this subsection we will describe only the issues and solutions particular to a server that runs a web interface to a prover.

Three kinds of issues arise: security, availability and equal sharing of resources. First, exploiting bugs in our architecture could lead crackers to take control of the server. Second, in a centralized environment the only copy of files is on the server. Unavailability of the server makes users not only unable to work, but also unable to access their files. Last, when users access the same

server its resources are shared. If a particular prover uses all the memory or CPU other users are unable to work.

To provide security, the server is run in a chrooted environment (`chroot` is a Unix system call preventing a process to access any files outside of a special root directory), as a non-privileged user. The permissions include only reading server files and executing the provers. Every prover type is run as a different user (using the file `setuid` mechanism), that is allowed to read only the prover's library and has privileges to write only in a directory where the prover's proof scripts are stored. To disallow storing overly large amounts of data, filesystem quota may be used. All of those mechanisms are implemented in our prototype. We experimented with further virtualization using Xen [BDF+03], but the overhead of starting Xen domains was too big to be used practically. We expect that a non-trivial approach that uses virtualization may add to the security of the server.

For provers that allow system interaction, this functionality can be sometimes disabled. In particular, for ML based provers, dropping to the toplevel can be disabled. If the server administrator doesn't trust the prover's implementation, a secure version of the kernel can be used to disable irrelevant system calls. In this case even a system that is implemented inside ML like HOL Light can be available without changes to the prover itself.

To ensure equal sharing of resources, prover processes can be run with CPU quota and memory quota mechanisms. The scheduling policy can be changed (for example with the `nice` system call) to provide the server process with priority over prover processes. Different provers have different CPU and memory requirements, which should be taken into account while setting the limits.

When many users want to access the interface, the resources of a single server may be insufficient. It is simple to run the server on a set of machines, by calling provers as subprocesses through `ssh` on separate computers. A load balancing mechanism can be implemented.

The communication between the server part and the client part can be secured by providing the interface through HTTPS.

4.6 Prototype

We have implemented a prototype of a web interface according to the described architecture. The interface facilitates the use of Coq proof assistant with just a web browser, but it looks and behaves (Fig. 4.2) like the interfaces offered by CoqIde and ProofGeneral.

Our server is a 30kB OCaml program, that serves two HTML files and a number of JavaScript files. It additionally supports special POST requests for verifying and for undoing commands as well as for loading and saving of files. It uses the OCamlHttpd library, for web-server functionality.

Our client consists of 15kB of JavaScript and 3kB of HTML. Most of the client-side code is responsible for the locking of the buffer and recognition of Coq expressions.

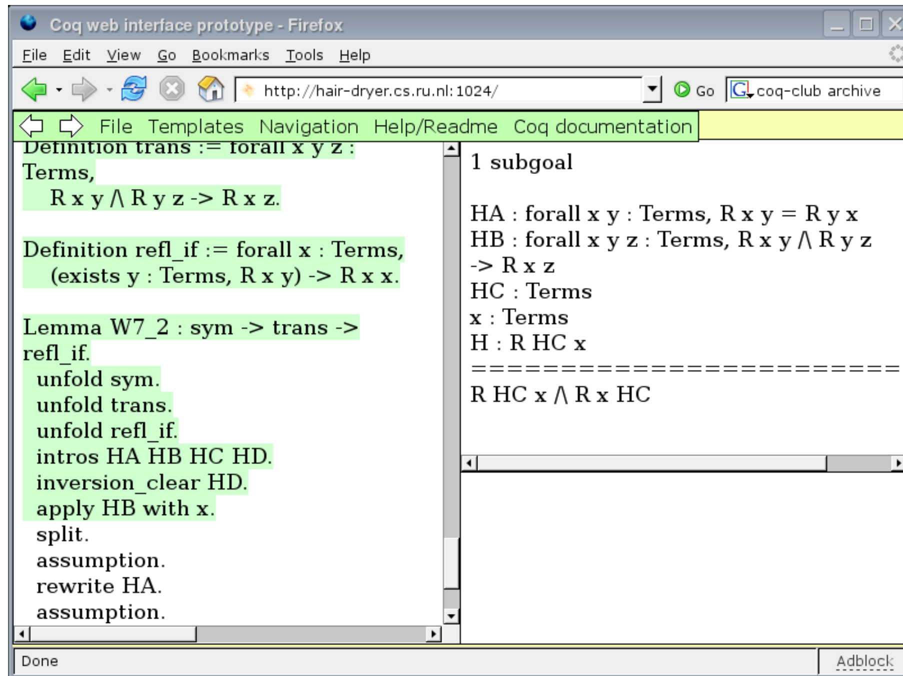


Figure 4.2: Screenshot of the prototype, that shows working with a Coq proof. The verified part of the edit buffer is colored and locked. The state buffer shows the state of the proof, there are no Coq warnings.

To secure our prototype the server is run as `nobody` in a minimal `chroot`ed environment. The prover sub-processes are `renice`d not to interfere with the main server process. Dropping from Coq to OCaml toplevel is disabled. The access to the interface was initially password protected to avoid creating prover sessions for web-spiders. Limiting the number of sessions per IP address is sufficient to avoid opening too many idle sessions for web-spiders.

Our prototype includes a `lkB` file, that is supposed to create a uniform layer that works with different browsers. We have not yet made it as portable, as the asynchronous DOM modification allows. In particular our prototype works well with Gecko-based browsers (Mozilla, Firefox, Galeon, . . .), KHTML based browsers (Konqueror, Safari) and Opera 9. It works with Internet Explorer 6 with minor problems, for example some key-bindings are missing because they are assigned to internal functions. It does not yet work with any older versions of the above. We have tested our implementation's efficiency, by trying to use the server from other locations. Although measuring responsiveness to user's actions is hard to be done objectively, our experiments show, that with reasonable network latency, its responsiveness is very good.

The prototype is a Coq web interface, but there is not much code specific to Coq. The client part includes recognition of Coq comments and whole ex-

pressions to send. The server part includes recognition of successes and failures as well as the *undo* mechanism. For all ELisp code from ProofGeneral equivalent JavaScript code can be written and regular expressions provided. Thus adapting these three things to other provers should be simple, which is why we believe that implementing an interface according to our architecture that would support different provers can be easily done.

The client part has to overcome the minor differences between browsers. In particular it includes functions that create a uniform layer for XmlHttpRequest creation, event binding, and DOM that work the same way on all currently supported browsers.

4.6.1 Possible Uses

Our interface can be used to create interactive tutorials presenting proof assistants. We have created a special proof script, that includes a slightly modified version of the official Coq tutorial. The descriptive parts have been put inside comments (including the HTML formatting), and commands to the proof assistant have been left outside comments. A user that enters such a page may just read the tutorial and execute the commands in Coq environment, but may also do own experiments with it.

Non-trivial proof scripts that use tactics are unreadable without intermediate proof states. Thus proofs presented on the web are usually accompanied with some of the proof states often automatically generated by Coqdoc or TeXmacs [AR04]. A web interface can be used (even in a read-only mode) to present such proofs interactively. In this way, the user reading the proof chooses which proof states to see.

External proof assistant libraries can be included on the server. With our server we included C-CoRN [CFGW04] (Constructive Coq Repository at Nijmegen) and Berkeley contribs [O'C05] (Goedel incompleteness theorem verified in Coq). Such libraries can be developed on the server. In such an approach visitors can always see and test the current version, without downloading and compiling the library.

Modified and experimental versions of provers usually require patching a particular version of the source of the proof assistant. Presenting such a modified version to others is easily possible with the given infrastructure. The server offered the C-zar declarative proof language extension for Coq [Cor08] before it was included in mainline branch.

4.7 Implementation

In this Section we describe the details of how our prototype has been implemented.

4.7.1 The server process

We implemented the server process as a complete web server. We used the OCamlNet [Oca] library for a basic HTTP web server that instead of processing HTTP requests sends them to our function. This function checks the parameters of the request. If the request includes a session number, the request is processed further by this session, otherwise if it is a login request, we try to authenticate and spawn a session. If it is not the case, the user is redirected to the login page.

A session provides a number of operations:

- Adding a piece of prover text to the queue
- Undoing to a particular location in the text (which can be in particular used to stop the prover)
- Checking whether there is new processed text from the prover
- Restarting a session (may be faster than undoing till the beginning, and may be additionally useful for ensuring resynchronization)
- Deleting a session

A particular session keeps a prover process as well as a queue of requests awaiting to be sent to it. The file implementing sessions also keeps a thread that checks every minute for sessions that have been contacted by the users web browser for longer than an hour and kills them.

We implemented a general interface to provers, that keeps a subprocess and is able to send input to it and read information from it. For every prover there is a number of different specialized behaviors that a particular prover module implements:

- Detecting the prompt. This allows knowing when output has ended and the prover is ready for reading new commands.
- Knowing whether a command has succeeded or returned an error. This allows knowing what is the current state of the prover in relation to the position in the buffer.
- Issuing undo commands. This works differently in different provers.
- Not allowing disabled commands.

4.7.2 The server environment

The machine that runs the server runs the standard Apache HTTP server. It is configured to redirect the special requests (requests to a particular file) to a proxy, which is the other HTTP server. This allows for keeping the main web server constantly running even with trouble with the specialized server.

The specialized server is put in a special chroot environment. This environment contains only the server, the files it needs (logins and a log file) and the proof assistant. The necessary system libraries are minimal. The standard library and other desired libraries of the proof assistant should be provided as well.

The server process is run as root in the chroot environment, and after binding the port it immediately drops root privileges. It switches to the user `nobody`, which doesn't include a home directory. The chroot includes a directory for user files, which is the only place the nobody user is able to write. The server is equipped with a special firewall that doesn't allow the user to open any network connections. Standard Unix user limits for this user are set appropriately to prevent crashing the server.

4.7.3 Client side

The client normally starts with a login page, which is a normal HTML page. When the user fills in the login, it is sent as a POST request to the specialized server. If the credentials were correct, a new specially generated HTML page is sent to the user. This page includes all necessary variables like the session number or the prover type. It also requests the other necessary Javascript files to be included: a number of common files and a file for the particular prover.

The first part of the implementation handles the buffer and the operations on it. The buffer includes four regions:

- The green region is the text already processed by the prover
- The blue region is the text that is currently being processed
- The yellow region with text that is to be processed but has not yet been sent to the prover
- The non-locked region which the user is able to edit.

Implementation takes care that the verified regions stay in this order and are not changed by the user.

The second part is responsible for taking care about the prover operations. It implements the following functions (each is a JavaScript function):

- Sending one command from the script to the prover,
- Sending all the script to the prover,
- Undoing one command,
- Restarting the session,
- Going to a particular point in the buffer.
- A general callback for processing information that came back from the prover that also tries to send more information from the yellow region of the buffer.

The last part of the client side are the prover specific scripts. For a particular prover finding the end of the next expression may not be a trivial task. For example for Coq, a single dot is an end of an expression, but two dots are not. Three dots end an expression, but a dot inside comments or inside quotes or after a module name does not.

Additionally for particular provers mapping between some of the sequences and Unicode symbols allows for nicer display.

4.8 Concluding Remarks

We presented an architecture to create simple, lightweight and fast web interfaces to proof assistants. Such interfaces are a novelty in the domain. Our solution works with modern web browsers without installing any additional software. The installation and updating of provers is done only on the server, the users do not need to do anything. It is therefore completely platform independent.

The communication mechanism uses the network minimally, making the interface comparably responsive to local ones. In comparison with other client-server solutions, the only limitation is the dependency on the web browser. Fortunately web browsers include full scripting languages, allowing implementation of nearly all possible functionality of the interface on the client side. In particular the browser's internal editors are weak in comparison with local editors. One can implement in JavaScript the handling of more key bindings to make the editor similar to a local one. Most features of state-of-the-art local interfaces for proof assistants can be imitated this way. The efficiency of an editor implemented in JavaScript would depend on the browser interpreting it. We have not been able to find any such editor.

We believe that a centralized environment, with provers accessible through a web interface, is not limited in comparison with local interfaces, and that the architecture we have presented is in the spirit of the current trends of development in computer science.

We will look at extending the architecture to a complete wiki-like architecture in Chapter 7. This requires a versioning mechanism and merging of users' changes on the server. Additionally proof displaying and searching mechanisms are mandatory. Editing conflicts can be resolved in similar way as it is done in wiki software. For example if the file was changed and a user wants to save over it, differences are presented.

The protocol we use for communication between the client side and the server side is an *ad-hoc* protocol. We considered using the general prover interaction protocol PGIP [ALW05], but it was not designed with network operation in mind. Still it is XML-based, so parts of it may even be passed by the server directly to browsers, since they are already able to parse XML. On the other hand the protocol may include too much information causing unnecessary overhead, since it was designed as a local protocol.

Other implementation features that would require significant amounts of

work, but that would give rise to a much better interface include: providing more provers, making the interface compatible with all browsers that support asynchronous DOM modifications, implementing the *break* mechanism, adding syntax highlighting, and providing better security.

Chapter 5

Teaching logic using a proof assistant

5.1 Introduction

5.1.1 Motivation

At most European universities, part of the undergraduate computer science curriculum is an introductory course that teaches the rules of propositional and predicate logic. At the Radboud Universiteit (RU) in Nijmegen this course is taught in the first year and is called 'Beweren en Bewijzen' (Dutch for 'Stating and Proving'). At the Vrije Universiteit (VU) in Amsterdam this course is taught in the second year and is called 'Inleiding Logica' ('Introduction to Logic'). Almost all computer science curricula have similar undergraduate courses.

For learning this kind of elementary mathematical logic it is crucial to work many exercises. Those exercises can of course be done in the traditional way, using pen and paper. The student is completely on his own, and in practice it often happens that proofs that are almost-but-not-completely-right are produced. Alternatively, they can be made using some computer program, which guides the student through the development of a completely correct proof. A disadvantage of the computerized way of practising mathematical logic is that a student often will be able to finish proofs by random experimentation with the commands of the system (accidentally hitting a solution), without really having understood how the proof works. Of course, a combination of the two styles of practicing formal proofs seems to be the best option. Computer assistance for learning to construct derivations in mathematical logic is desirable. Currently the most popular program that is used for this kind of 'computer-assisted logic teaching' is a logical framework called Jape [BS96], developed by Surin and Bornat at the University of Oxford.

Besides exercises there is also the issue of examination. It would be good

if the student has the opportunity to do at any moment a (part of the) logic exam by logging in to the system and be presented with a set of exercises from a database that have to be solved within a certain time. This may require human supervision to prevent cheating. We did not yet work on this, but just mention it as a possible interesting application of computer-assisted logic teaching.

5.1.2 Our contribution

This chapter describes our development, named ProofWeb. It provides functionality much like Jape (it might be considered to be an 'improved Jape-clone'). The two main innovations that our system offers over Jape and other similar systems are:

- The students work on a centralized server that is accessed through a web interface. The proof assistant will not run on their computer, but instead will run on the server.

  A first advantage is flexibility. The web interface is light: the student will not need to install anything to be able to use it, not even a plug-in. When designing our system we tried to make it as low-threshold as possible. Additionally since there is no installation required there is no danger of computer viruses. The student can work from any internet-connection at any time.

  A second advantage is that the student does not need to worry about version problems with the software or the exercises. Since everything is on the same centralized server, the students have at any time the right version of the software, exercises, and possibly solutions to exercises available, and moreover the teachers know at any time the current status of the work of the students.

- The system makes use of the proof assistant Coq. This means that students have access to all the currently available formalizations in Coq. We chose Coq because both at the RU and at the VU it is already used in research and teaching.

  An advantage of using a state-of-the-art proof assistant is again flexibility. The same interface can be used (possibly adapted) for teaching more advanced courses in logic or proof assistants.

We also include in ProofWeb:

- A large collection of logic exercises. The exercises range from very easy to very difficult, and are graded for their difficulty. The exercise set is sufficiently large (presently over 200 exercises) that the student will not soon run out of practice material. More about the exercise set can be found in Section 5.5.

- Course notes, with a basic presentation of propositional and predicate logic, and a description of how to use the system. We choose the presentation of the proofs in the system to be the same as the presentation of the proofs in textbooks. Therefore we developed both the 'Gentzen-style' and the 'Fitch-style' natural deduction variants.

5.1.3 Related work

There are already numerous systems for doing logic by computer, of which Jape is the best known. A relatively comprehensive list is maintained by Hans van Ditmarsch [vD07]. Of course many of these systems are quite similar to our system (as well as to each other). For instance, quite a number of these systems are already web-based.

The distinctive features of our system are the use of a serious proof assistant, together with a *centralized* 'web application' architecture. The work of the students remains on the web server. It can be saved and loaded back in, and the progress of the student is at all times available both to the student, the teacher and the system (i.e. the system has at all times an accurate 'user model' of the abilities of the student).

Our system has been developed for teaching logic in the natural deduction style. There also exists a school of teaching logic due to Dijkstra and Gries, called 'Calculational Logic', in which reasoning is done through rewriting with equations. The Coq system is powerful enough to support this kind of reasoning as well, but we have not developed this style of logic in our system.

5.1.4 Contents

In this chapter we present both our project and the system. We start with a short description of our project experiences in Section 5.2. Next, in Section 5.3 we present the architecture of the interface. Section 5.4 is concerned with the supporting infrastructure of tactics and exercises, and Section 5.5 with the presentation of the collection of exercises. Finally, in Section 5.6 we give an outlook on future work.

5.2 Experiences in the project

In the beginning of the project, ProofWeb was developed as a web-interface for using Coq on a centralized server. As such, the system was already used in three master courses on type theory using Coq:

1. In fall 2006: the course 'Logical Verification' at the VU [LV], taught by Femke van Raamsdonk. This is a computer science master's course about the type theory of the Coq system. The course is meant for more mature students but also recapitulates some undergraduate logic. It is therefore suitable for testing a first version of ProofWeb. Natural deduction is

taught in Gentzen style, that is, proofs have a tree-like structure, and grow upward from the conclusion of the proof.

2. In spring 2007: the course 'Type Theory' at the RU, taught by Freek Wiedijk and Milad Niqui. This course is also a master's level course about the type theory of the Coq system, and corresponds to the Logical Verification course at the VU.

3. In spring 2007: the course 'Type Theory and Proof Assistants' in the 'Master Class Logic 2006-2007', taught by Herman Geuvers and Bas Spitters. This course is similar to the previous ones, but is aimed at master's students from all over the Netherlands.

These courses were opportunities to test the interface of ProofWeb on mature students. Since the students were not using tactics that involve automation, the efficiency of the server turned out not to be a problem. At peak times around sixty students use about 2Gb memory and a fraction of a CPU.

During these courses there was not yet support for visualizing proofs. Instead the students had to do their proofs using the customary Coq proof style, which consists of building a tactic script using the standard Coq tactics. This was not problematic, since one of the aims of the courses is to learn Coq.

Initially we did not have a dedicated server, so the prototype was running on one of the group servers of the research group in Nijmegen on a non-standard port. One of the issues then was that the web-proxy at the VU did not allow the students to access pages running on non-standard ports, so they were required to turn the proxy off.

One of the assignments in the course 'Logical Verification' at the VU involves program extraction. For security reasons we did not allow running the extracted programs on the server, and therefore a mechanism allowing the students to obtain the extracted program was implemented.

In 2007, ProofWeb was used in two different undergraduate logic courses. In both courses the students used the special tactics, the display, and the database with exercises to practice natural deduction proofs.

1. In spring 2007: the course 'Beweren en Bewijzen' at the RU [Bew] taught by Hanno Wupper and Erik Barendsen. This is a computer science undergraduate course in logic using Gentzen style 'tree' proofs. See Chapter 6 for a more elaborate discussion about ProofWeb and Gentzen style natural deduction.

2. In fall 2007: the course 'Inleiding Logica' at the VU [Inl] taught by Roel de Vrijer. This is a computer science undergraduate course in logic, with natural deduction in Fitch style (see next chapter), that is, proofs have a structure of nested boxes, which structure a sequential list of proof steps. Another name for this kind of proofs is 'flag-style proofs', because often the assumptions of a subproof are written in the shape of 'flags'.

The second course in which ProofWeb was used was the course 'Type Theory' in spring 2007 at the RU. The first half of this course is basically an accelerated clone of the 'Logical Verification' course. As it turned out that initially there were only very few students who wanted to follow this course, it was decided that there would be no lectures, and that the students just would be given the course notes of 'Logical Verification' together with access to the server. They then would work on their own, with an opportunity to call for help if needed. It turns out that this worked unexpectedly well. The students just studied the lecture notes and did the exercises of the course. And even without much pressure on them in the form of requiring them to meet deadlines, they managed to keep on schedule reasonably well. The only thing that at some point confused them (after which a lecture was organized to make things clear) was the part of the course that did not correspond to Coq work: derivations in Pure Type Systems.

All in all our experience is that the system ProofWeb seems to work very well in teaching. Indeed, hardly any students used more traditional Coq interfaces like Proof General or CoqIDE.

5.3 Architecture of the interface

The interface is an implementation of the architecture for creating responsive web interfaces for proof assistants described in the Chapter 4. The architecture described there was designed as a publicly available web service. Using it for teaching required the creation of groups of logins for particular courses. The students are allowed to access only their own files via the web interface, and teachers of particular courses have access to students' solutions through the admin interface.

An example of the use of the interface in the 'Logical Verification' course can be seen in Figure 5.1.

5.4 Natural deduction for first-order logic

A first aim in the development of ProofWeb was to have an exact correspondence between the derivations on paper and the derivations in the system. The student should then work with a set of dedicated tactics, because the standard Coq tactics are too powerful. (For instance, one could solve the exercises in propositional logic using the tactic `tauto` instead of building the actual derivation.)

The standard way of working with Coq is with backward proofs, so we first naturally arrived at a set of backward tactics: every proposition (the current goal) is deduced from another proposition (the new goal) using a deduction rule. The display style that fits most naturally to this kind of proof is a proof tree (flag-style proofs are described in the next chapter). This imposes a strict way of working. The proof trees have to be constructed from 'bottom to top'. On

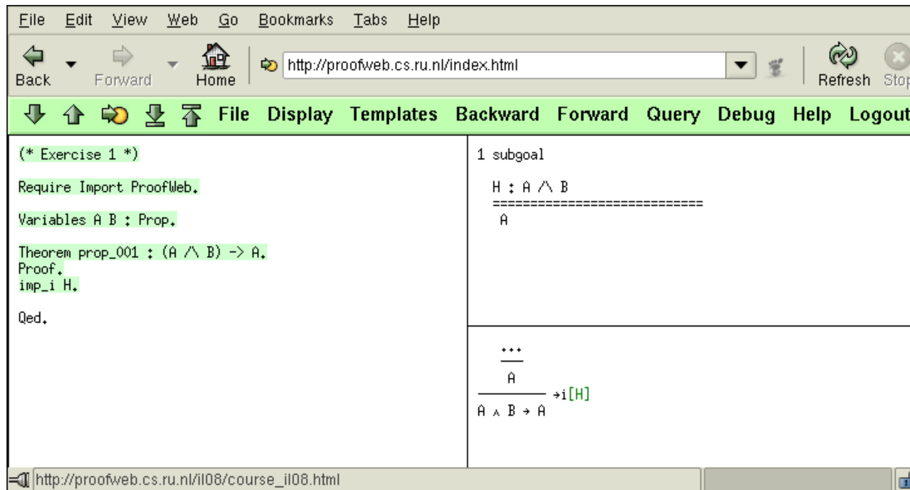


Figure 5.1: A propositional logic exercise in ProofWeb.

the one hand, this makes the construction of a deduction more difficult than on paper, because there is no possibility of building snippets of the proof in a forward way, using what is known from the hypotheses and their consequences. But on the other hand, the method forces the student to ponder the general structure of the proof before deciding by what step he will eventually end up with the current proposition. And the imposed rigidity is congenial with the aim of a logic course to encourage rigorous analytical thinking. Moreover, it becomes very clear where ingenuity comes in, such as with the disjunction elimination rule. The student is supposed to prove some proposition C. It is a creative step to find a disjunction $A \vee B$, prove this, and also prove that C follows from both A and B separately. The same goes for the introduction and elimination of negation.

As an example we present the tactic for disjunction elimination, which gives a good impression of the way additional tactics are implemented:

```
Ltac dis_e X H1 H2 :=
  match X with
  | ( _ \/ _ ) =>
   let x := fresh "H" in
   assert (x : X);
   [ idtac | elim x; clear x; [intro H1 | intro H2] ]
  end
  || fail "(the argument is not a disjunction
    or the labels already exist)".
```

If the current goal is C, the tactic `dis_el (A \/ B) G H` will create the following three new goals:

1. $A \vee B$;
2. C, with the extra assumption A with label G;
3. C, with the extra assumption B with label H.

Also, the tactic gives a nice and understandable error message. All natural deduction tactics have been given a name by using three letters of the connective's name and indicating whether the tactic implements an introduction rule or an elimination rule (and if necessary, if that is a left or a right variant or whether it is the forward tactic). We give a small example of a proof with our set of tactics (The forward tactics are denoted with the `f_` prefix and allow using the natural deduction tactic with the goal to be closed):

```
Theorem pred_076 : all x, exi y, (P(x) \/ P(y)) -> exi x, P(x).
Proof.
imp_i H.
insert G (exi y, (P(x0) \/ P(y))).
f_all_e H.
exi_e (exi y, (P(x0) \/ P(y))) y0 J.
ass G.
dis_e (P(x0) \/ P(y0)) K K2.
ass J.
f_exi_i K.
f_exi_i K2.
Qed.
```

5.4.1 Visualization

A second aim is a visual presentation of proofs as in Jape. This meant requesting the proof information from Coq and converting it to a graphic format. Coq internally keeps a proof state. This proof state is a recursive OCaml structure, that holds a goal, a rule which allows to obtain this goal from the subgoals, and the subgoals themselves. It is not just a tree structure, since a rule can be a compound rule that contains another proof state. Tactics and tacticals modify the proof state. Coq includes commands that allow inspecting the proof state. `Show` with a number allows the user to see in detail a goal that is not currently being worked on, `Show Tree` shows the succession of conclusions, hypotheses and tactics used to obtain the current goal and `Show Proof` displays the CIC term (possibly with holes).

The output of these commands was not sufficient to build a natural deduction tree for the proof. We added a new command `Dump Tree` to Coq that allows exporting the whole proof state in an XML format. An example of the output of the `Dump Tree` command for a very simple Coq proof:

```
<tree><goal><concl type="A -> A"/></goal>
  <cmpdrule><tactic cmd="intro x"/>
    <tree><goal><concl type="A -> A"/></goal>
```

```
      <cmpdrule><tactic cmd="intro x"/>
        <tree><goal><concl type="A -> A"/></goal>
          <rule text="intro x"/>
          <tree><goal><concl type="A"/><hyp id="x" type="A"/>
        </goal></tree></tree>
      </cmpdrule>
      <tree><goal><concl type="A"/><hyp id="x" type="A"/>
    </goal></tree></tree>
  </cmpdrule><tree><goal><concl type="A"/><hyp id="x" type="A"/>
</goal></tree></tree>
```

ProofWeb is able to parse the XML trees dumped by Coq and generate natural deduction diagrams (see Figure 5.2). The diagrams are rendered as HTML tables with fixed with Unicode ASCII characters, so that they can be transferred over the network efficiently. Those diagrams may be requested by the user's browser in special query requests (as described in 4.4). The diagrams are displayed in a separate frame in the interface along with the usual Coq proof state. If the user switches on the display of the diagrams, the client side requests them when no text is being processed.

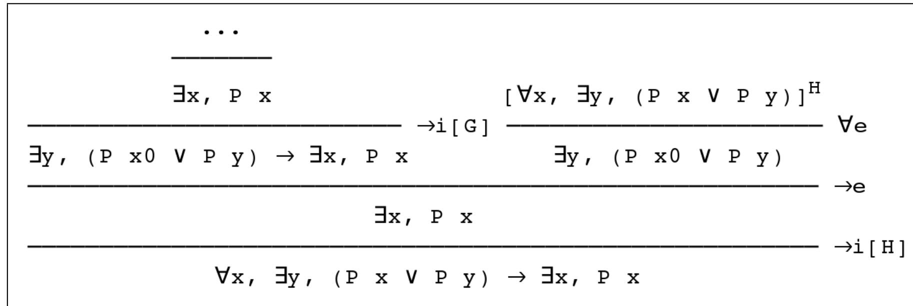


Figure 5.2: A natural deduction tree as seen on the web page.

5.5 The exercise set

As a part of the project a set of over 200 exercises was developed. If a student logs in via the web interface as a participant to a specific course, he sees the list of exercises for the course (see Figure 5.3). It gives for each exercise the name of file that holds the exercise, an indication of the difficulty, the current status of the exercise, and a button for resetting the exercise to its initial state.
The four possibilities for the status of an exercise are:

- Not touched (in grey)
- Incomplete (in red)
- Correct (in orange)
- Solved (in green)

The colors are meant to resemble the colors of a traffic light.

- The status Not touched means that an exercise has not been opened, or has been opened but has not been saved.
- The status Incomplete means that the file is incomplete or wrong. If one want to know why ProofWeb thinks there is an error in the solution, one can click the why? link next to the status. The system shows a new window that shows the error message of Coq.
- The status Correct means that the file is a correct Coq-file. However, the file is not accepted as a solution for the exercise in the course. This happens for instance if more automation (present in Coq) is used than intended for the course, for instance: by proving a propositional formula with the tactic `tauto` instead of using the tactics corresponding exactly to the logical rules used in the course. It also happens if the file is empty. If the status is Correct one can click on the why? link to find out which steps are not allowed in the course.

  This is a feature of ProofWeb is meant as a service for the teacher, but of course in addition manual verification may be required, for instance if the exercise is to give the definition of a certain object in type theory.
- The status Solved means that the file is correct Coq and moreover is accepted as a solution in the course.

File Edit View Web Go Bookmarks Tabs Help

Back Forward Home http://proofweb.cs.ru.nl/logged.html Go Refresh Stop

Predicaatlogica_061.v	Easy	Solved	Reset Predicaatlogica_061.v
Predicaatlogica_062.v	Easy	Incomplete (why?)	Reset Predicaatlogica_062.v
Predicaatlogica_063.v	Medium	Not touched	Reset Predicaatlogica_063.v
Predicaatlogica_064.v	Medium	Not touched	Reset Predicaatlogica_064.v
Predicaatlogica_065.v	Difficult	Not touched	Reset Predicaatlogica_065.v
Predicaatlogica_066.v	Difficult	Not touched	Reset Predicaatlogica_066.v
Predicaatlogica_067.v	Medium	Not touched	Reset Predicaatlogica_067.v
Predicaatlogica_068.v	Easy	Solved	Reset Predicaatlogica_068.v
Predicaatlogica_069.v	Medium	Not touched	Reset Predicaatlogica_069.v
Predicaatlogica_070.v	Easy	Solved	Reset Predicaatlogica_070.v
Predicaatlogica_071.v	Easy	Correct (why?)	Reset Predicaatlogica_071.v

Figure 5.3: Tasks assigned to students and their status.

The verification tool lexes the original task and the student's solution in parallel. The original solution includes placeholders that are valid Coq comments. An example task file on the server looks like:

```
(* Exercise 118 *)

Require Import ProofWeb.

Variables A B C : Prop.

Theorem prop_118 :
  ~~(A \/ B) -> (~~A -> ~~C) -> (~~B -> ~~C) -> ~~C.
(*! prop_proof *)
Qed.
```

Those placeholders mean that a particular place needs to contain a valid Coq term or a valid proof. For proofs the kind of proof determines the set of allowed tactics. For proofs and terms of given types the automatic verification is enough. However, there are tasks where students are required to give a definition of a particular object in type theory. For this kind of tasks manual verification by a teaching assistant of a course is required.

Below we list the tactics allowed in courses:

- Type theory and Coq courses:

  ```
  intro, intros, apply, exact, unfold, fold, elim,
  split, left, right, assumption, simpl, elimtype,
  inversion, induction, exists, reflexivity, absurd,
  inversion_clear, clear, constructor, rewrite, assert,
  destruct, replace, symmetry
  ```

- Basic logic courses:

  ```
  con_in, con_ell, con_elr, dis_inl, dis_inr, dis_el,
  imp_in, imp_el, equ_in, equ_el, neg_in, neg_el, efq,
  ass, neg_els, tnd_axiom, dn, dn_axiom, all_in, all_el,
  exi_in, exi_el, equ_eli, rewrite
  ```

- Constructive proofs:

  ```
  con_in, con_ell, con_elr, dis_inl, dis_inr, dis_el,
  imp_in, imp_el, equ_in, equ_el, neg_in, neg_el, efq,
  ass, all_in, all_el, exi_in, exi_el, equ_eli
  ```

- Propositional logic courses:

  ```
  ass, copy, exact, insert, con_i, con_e1, con_e2,
  dis_i1, dis_i2, dis_e, imp_i, imp_e, neg_i, neg_e,
  fls_e, tru_i, negneg_i, negneg_e, LEM, PBC, MT,
  b_con_i, b_con_e1, b_con_e2, b_dis_i1, b_dis_i2,
  b_dis_e, b_imp_i, b_imp_e, b_neg_i, b_neg_e, b_fls_e,
  ```

```
b_tru_i, b_negneg_i, b_negneg_e, b_LEM, b_PBC, b_MT,
f_con_i, f_con_e1, f_con_e2, f_dis_i1, f_dis_i2,
f_imp_e, f_neg_e, f_fls_e, f_tru_i, f_negneg_i,
f_negneg_e, f_LEM, f_MT, f_dis_e, f_exi_e
```

- Predicate logic additionally allows:

```
all_i, all_e, exi_i, exi_e, equ_i, equ_e, equ_e',
b_all_i, b_all_e, b_exi_i, b_exi_e, b_equ_i, b_equ_e,
b_equ_e', f_all_e, f_exi_i, f_equ_i, f_equ_e
```

5.6 Outlook

A final version of the course notes is available [KRW+08].

Some of the issues that can be worked on are the following.

- The teacher web interface allows to manage students logins and inspect the work of the students. Managing the set of exercises is at the moment only possible by logging on to the server through an ssh connection, and then listing and editing files manually. Clearly, a proper web interface for this is necessary.
- The deduction trees are currently rendered as text or HTML in IFrames, and can be optionally opened in a separate browser window to allow easy printing as PostScript or PDF. However students may need to use the trees in texts, and for that a dedicated TEX or image rendering of the trees could be implemented.
- The interface uses some web technologies that are not implemented in the same way in all browsers. It includes a small layer that is supposed to abstract over incompatible functionalities. Currently this works well with Gecko based browsers (like Mozilla, Firefox, Galeon, Epiphany and Netscape), Webkit based browsers (like Safari and Konqueror), and the Opera browser. Also, some effort has been made to make the system work reasonably well with common versions of Internet Explorer (see Section 4.6). However, it needs further attention.
- The documentation of how to install and maintain the server is in a rudimentary state. Our server currently is available to everyone who wants to experiment with our system, but a good guide that explains how to install a server would be useful.
- The system keeps a log of each interaction of each student session, is already stored on the server. Using these logs, it is possible to develop software for 'replaying' student sessions. We are currently discussing whether it is useful to develop such an extension of the system.

- The system was designed to be used in standard university courses. It might be useful to create a more complete online environment that would include introductory explanations and adaptive user profiles, therefore allowing students to learn logic without teacher interaction.

5.6.1 Beyond the project

If the development of ProofWeb is finished, a possibility is to integrate it with a system that supports the development of more serious proofs with the Coq system. One of the other projects being pursued is the creation of a so-called 'math wiki' described in Chapter 7. Here, traditional wiki technology is integrated with the same proof assistant front end that our system is based on. Combining it with en educational environment might be a long-standing goal.

Chapter 6

Merging procedural and declarative proof

6.1 Introduction

6.1.1 Procedural versus declarative proof assistants

Proof assistants are computer programs for constructing and checking proofs. In these systems one can distinguish between two quite different kind of entities that both might be considered the 'proofs' that are being checked:

- First there are the low level proofs of the logic of the system. In type theoretical systems these are the *proof terms*. In other systems they are built from tiny proof steps called *basic inferences*. Generally such proof objects are huge and constructed from a small number of basic elements.
- Then there also are the high level proof texts that the user of the system works with. Often these texts are *scripts* of commands from the user to the proof assistant. These texts are of a size comparable to traditional mathematical texts, and contain a much larger variety of proof steps. For instance both the Coq and HOL systems have dozens of *tactics* that can occur in this kind of proof.

The proof assistant does two things for the user. First it translates high level proofs into low level proofs, and secondly it checks the low level proofs obtained in this way with respect to the rules of the logic of the system.

As an example, the following 'high level' Coq proof script:

```
Lemma example : forall n : nat, n <= n.
intros.
omega.
Qed.
```

is translated to the following 'low level' proof term:

```
fun n : nat =>
Decidable.dec_not_not (n <= n) (dec_le n n)
  (fun H : ~ n <= n =>
   ex_ind
     (fun (Zvar1 : Z) (Omega5 : Z_of_nat n = Zvar1 /\ (0 <= Zvar1 * 1 + 0)%Z) =>
      and_ind
        (fun (Omega3 : Z_of_nat n = Zvar1) (_ : (0 <= Zvar1 * 1 + 0)%Z) =>
         let H0 :=
           eq_ind_r (fun x : Z => (0 <= x + -1 + - Z_of_nat n)%Z -> False)
             (eq_ind_r (fun x : Z => (0 <= Zvar1 + -1 + - x)%Z -> False)
                (fast_Zopp_eq_mult_neg_1 Zvar1
                   (fun x : Z => (0 <= Zvar1 + -1 + x)%Z -> False)
                   (fast_Zplus_comm (Zvar1 + -1) (Zvar1 * -1)
                      (fun x : Z => (0 <= x)%Z -> False)
                      (fast_Zplus_assoc (Zvar1 * -1) Zvar1
                         (-1) (fun x : Z => (0 <= x)%Z -> False)
                         (fast_Zred_factor3 Zvar1 (-1)
                            (fun x : Z => (0 <= x + -1)%Z -> False)
                            (fast_Zred_factor5 Zvar1
                               (-1) (fun x : Z => (0 <= x)%Z -> False)
                               (fun Omega4 : (0 <= -1)%Z =>
                                Omega4 (refl_equal Gt))))))) Omega3) Omega3 in
         H0 (Zgt_left (Z_of_nat n) (Z_of_nat n) (inj_gt n n (not_le n n H))))
        Omega5) (intro_Z n))
```

which then is type checked and found to be correct.

A good proof assistant should hide low level proofs from the user of the system as much as possible. Just like a user of a high level programming language should not need to be aware that the program internally is translated into machine code or bytecode, the user of a proof assistant should not have to be aware that internally a low level proof is being constructed.

It depends much on the specific proof assistant what the high level proofs look like. There are two basic groups of systems, as first introduced in [Har96b]:

The procedural systems such as Coq, HOL and PVS. These systems generally are descendants of the LCF system. The proofs of a procedural system consist of tactics operating on goals. This leads to proofs that can naturally be represented as *tree shaped* derivations in the style of Gentzen. For instance, the example Coq proof then looks like:

$$\dfrac{\dfrac{}{\texttt{n:nat} \vdash \texttt{n <= n}}\ \texttt{omega}}{\vdash \forall\texttt{n:nat, n <= n}}\ \texttt{intros}$$

The above is a screenshot from the display of our ProofWeb system. In practice it is more useful to have ProofWeb display the tree without contexts:

$$\dfrac{\dfrac{}{\texttt{n <= n}}\ \texttt{omega}}{\forall\texttt{n:nat, n <= n}}\ \texttt{intros}$$

The declarative systems. The main two systems of this kind are Mizar and Isabelle (when used with its declarative proof language Isar), but also automated theorem provers like ACL2 and Theorema can be considered to be declarative. There are experimental declarative proof languages, like the C-zar for Coq by Pierre Corbineau [Cor08] and by John Harrison for HOL Light [Wie01]..

The proofs of a declarative system are *block structured*. They basically consist of a *list* of statements, where each statement follows from the previous ones, with the system being responsible for automatically constructing the low level proof that shows this to be the case. Apart from these basic steps declarative proofs have other steps, like the `assume` step which introduces an assumption.

In declarative systems these proof steps are grouped into a hierarchical structure of *blocks*, just like in block structured programming languages. In declarative proofs these blocks are delimited by keywords like `proof` and `qed`.

Some systems might be considered not to be *fully* declarative in the sense that they still require the user to indicate *how* a statement follows from earlier statements. For instance this holds for Isabelle, where the user can (and sometimes must) give explicit inference rules. Indeed, it is common among the users of Isabelle to refer to the Isar proofs not as 'declarative' but 'structured'. However, for the purposes of this paper this distinction does not matter. In fact, the declarative proofs that we generate with our ProofWeb system also have the property that they contain an explicit tactic at each step in the proof.

Declarative proofs are similar (although more precise and, with current technology, much more fine-grained) to the language that one finds in mathematical articles and textbooks.

The contribution of this chapter is a generic method for converting a procedural proof to a declarative proof. For Coq this method has been implemented in the ProofWeb system. ProofWeb can display a high level Coq proof as a block structured list of statements. Here is how it will display the example proof:

```
1 | n: nat              assumption
2 |    n <= n           omega
3      ∀n:nat, n <= n   intros 1-2
```

The rest of the chapter details the algorithms used for this.

In Chapter 5 we described the *Web deduction for education in formal thinking* project in which we built ProofWeb. One of its features is that it allows the students to both work in Gentzen style and in Fitch style. Proofs are displayed in (almost) exactly the same way that they are shown in the textbook. We decided to have our system be exactly compatible with a popular logic textbook by Michael Huth and Mark Ryan [HR04].

The ProofWeb system can present the tree shaped proof that corresponds to the Coq proof script as a Fitch style proof. This means that it converts a procedural proof (the Coq proof script) to a declarative proof (the Fitch display). The method that it uses to do this is *generic*. It will work for converting *any* procedural proof to *any* declarative proof text, independent of the specific proof assistants involved or their logical foundations.[1]

We decided against presenting the conversion method that we used generically. Here we present just the method for the very specific situation of natural deduction proofs for first order predicate logic with equality. However, the method *is* perfectly generic. Also, our implementation already is not restricted to the small set of tactics that the users of ProofWeb are supposed to use. It will work with *any* Coq proof, providing a block structured Fitch style display of that proof.

The specifics of the first order logics that ProofWeb uses can be found in the ProofWeb manual [KRW+08]. We here just show an example for both logics in Figure 6.1. In ProofWeb flags are rendered as boxes (like in Huth and Ryan), with the right hand border of the boxes omitted to conserve space.

Declarative proofs are much more robust than procedural proofs, and for this reason can be expected to have a longer useful lifetime than procedural proofs. For this reason, development of the technology presented here might mean current formalizations get a longer useful lifetime. A current version of the procedural system can be used to export a formalisation declaratively. Keeping the declarative proof instead of the procedural one gives a much higher chance of the proof being accepted by future versions of the proof assistant.

The conversion algorithm presented here also works on proofs that have not been completed yet. In that case one gets a declarative proof with *gaps*. For instance in ProofWeb, the Fitch style display of the proof *before* the `omega` tactic is executed will be:

```
1 | n: nat              assumption
  |    ...
2 |    n <= n
3      ∀n:nat, n <= n   intros 1-2
```

ProofWeb users often use the system through this feature. They do not look at the Coq proof state (which is also available to them), but just think in terms of the incomplete Fitch style proof.

This leads us to propose a new kind of prover interface. We call it a *luxury* declarative proof assistant (after a private suggestion by Henk Barendregt). In a luxury system, the user does not see goals, but works on an incomplete declarative proof. This proof then can be modified in two ways:

- Either the user just edits the text, the common way to work in a declarative

[1] The proof might contain some statements that have no good equivalent in the target system, and the automation of the target system might not always be able to bridge the gaps between the steps, but apart from those issues, a good starting point for a formalization in the target system can always be generated.

$$
\dfrac{[\exists x(P(x) \lor \neg Q(a))]^{\mathtt{H1}} \qquad \dfrac{[P(b) \lor \neg Q(a)]^{\mathtt{H3}} \qquad \dfrac{[P(b)]^{\mathtt{H4}}}{\exists x\, P(x)}\,\exists\text{i} \qquad \dfrac{\dfrac{[\neg Q(a)]^{\mathtt{H5}} \quad [Q(a)]^{\mathtt{H2}}}{\bot}\,\neg\text{e}}{\exists x\, P(x)}\,\bot\text{e}}{\exists x\, P(x)}\,\lor\text{e}\,[\mathtt{H4},\mathtt{H5}]}{\dfrac{\dfrac{\exists x\, P(x)}{Q(a) \to \exists x\, P(x)}\,\to\text{i}\,[\mathtt{H2}]}{\exists x(P(x) \lor \neg Q(a)) \to Q(a) \to \exists x\, P(x)}\,\to\text{i}\,[\mathtt{H1}]}\,\exists\text{e}\,[\mathtt{H3}]
$$

1	$\quad \exists x(P(x) \lor \neg Q(a))$	
2	$\quad\quad Q(a)$	
3	$\quad\quad b \quad P(b) \lor \neg Q(a)$	
4	$\quad\quad\quad\quad P(b)$	
5	$\quad\quad\quad\quad \exists x\, P(x)$	$\exists$i 4
6	$\quad\quad\quad\quad \neg Q(a)$	
7	$\quad\quad\quad\quad \bot$	$\neg$e 6,2
8	$\quad\quad\quad\quad \exists x\, P(x)$	$\bot$e 7
9	$\quad\quad\quad \exists x\, P(x)$	$\lor$e 3,4–5,6–8
10	$\quad\quad \exists x\, P(x)$	$\exists$e 1,3–9
11	$\quad Q(a) \to \exists x\, P(x)$	$\to$i 2–10
12	$\exists x(P(x) \lor \neg Q(a)) \to Q(a) \to \exists x\, P(x)$	$\to$i 1–11

Figure 6.1: Example derivation in Gentzen's and Fitch's systems.

proof assistant. This is flexible but gives the user no help in writing the proof.

- Alternatively the user executes a *tactic* at a step in the proof that has not been sufficiently justified yet, i.e., for which the system has not yet generated a low level proof. The 'goal' that this tactic sees has the statement of this step as the conclusion, and all the statements before it that are in scope as the assumptions. The tactic then will generate subgoals, which will be added to the proof text as new steps in, if needed, new sub-blocks.

  Modifying a proof in this style (by executing a tactic at a not yet justified step), needs exactly the same algorithms that the conversion from a procedural proof to a declarative proof needs.

  If one 'grows' a declarative proof in such a way, it basically will consist of a merged version of all the subgoals that the proof would have gone through in the procedural system.

It is desirable that in a luxury system *both* ways of working are available simultaneously. It should not be *required* to use tactics to modify a proof.

A simple version of this luxury concept is the following. In a declarative prover the user has to formulate the appropriate `assume` steps himself, while in a procedural prover he just can type `intros`. However in a luxury prover, the `intros` command will be available, which then will generate all the needed `assume` steps automatically. Similarly, appropriate statements in the case of an induction or application of a lemma can be generated automatically by the system.

6.1.2 Approach

The conversion from a tree style proof to a block structured proof is straightforward. It consists of two phases:

- First the tree is converted to a series of nested blocks in a naive way. This is trivial. However, it does not lead to a proof that a user will want to see, as there are many duplicate lines and boxes that are not necessary.

- The second phase is to *reduce* the proof. We use a rewrite system for this that eliminates various unwanted structures from the proof:

  - If a subproof has no new assumptions nor new variables, the block for it is not needed and can be flattened into the main proof.
  - Lines that are copies of earlier lines can generally be removed, as references to those lines can be replaced by references to the earlier lines.
  - 'Cuts' also can be removed from the proofs, as the declarative proofs really have a cut (in the Gentzen sense) at every line.

Below we will give the details of this rewrite system for proofs for the specific case of first order logic. We prove it to be terminating and confluent.

Our method is designed to convert proofs preserving the level of detail present in the original proof. When building a proof using automated tactics (decision procedures), the user might be curious after the proof that those tactics constructed internally. This is analogous to the rare occasion that a compiler user wants to see the machine code that was generated by the compiler. Our method does not work well for obtaining information on this level. However, Coq allows decomposition of tactics into smaller tactics using the `info` prefix, which means that getting such information is possible even when using our approach.

6.1.3 Related Work

There have been various projects for translating proofs from a procedural proof assistant into a declarative presentation, most notably the HELM system by Asperti et al., which was further developed in the MoWGLI project [APC$^+$03, AW02]. However, those systems almost always work on the level of proof terms and not on the level of tactics. For this reason the declarative proofs that these systems produced tend to be too convoluted for human consumption.

An exception is the system by Guilhot, Naciri and Pottier where Coq proofs are considered on the level of the tactics, by converting Coq proof trees just like we do [GNP03]. However in this work the generated text is only considered a *presentation* – they call it an *explanation* – and not a proof in a formal system like Fitch-style natural deduction in its own right.

Geuvers and Nederpelt [GN04] define a translation of natural deductions in Fitch style to simply typed λ-terms (i.e., their translation goes the opposite way from ours). They present reduction relations for Fitch-style deductions that allow simpler λ-terms to be obtained. These reductions remove unnecessary subproofs, remove repeats and unshare shared subproofs. They prove that Fitch deductions are mapped to the same λ-term if and only if they are equal under these relations; which shows that there is an isomorphism between these classes.

Proof nets [Gir87] allow representing proofs in a geometrical way where the order of the application of rules as well as irrelevant features of regular natural deduction proofs can be eliminated. Geuvers and Loeb [GL06] show the correspondence between deduction graphs and proof nets and give translations from minimal propositional logic to proof nets via context nets. They also shows how an operation of cut elimination in deduction graphs can be performed after the translation to a context net.

6.1.4 Contents

The rest of the chapter is organized as follows: In Section 6.2 we define and present a translation of minimal logic tree style proofs to flag style proofs. Then in Section 6.3 we present extending our translation to more complicated logical systems. In Section 6.4 we show how obtained proofs can be optimized and what

additional simplifications can be performed when translating forward proofs (Section 6.5). We then describe our implementation of the translation and show rendered proofs in both styles in Section 6.6. Finally, in Section 6.7 we present a conclusion and future work.

6.2 Translating minimal logic tree style proofs to flag style proofs

We first will restrict ourselves to minimal propositional logic. We introduce a translation operation ($\longmapsto$) that translates a tree style proof **G** of a proposition A to a flag style proof **F**. An example of such a translation is:

$$\emptyset : \quad \cfrac{\cfrac{[A]^x}{B \to A}\,\to\text{i}[y]}{A \to B \to A}\,\to\text{i}[x] \quad \longmapsto$$

1	A (flag)		
2	B (flag)		
3	A	copy 1	
4	$B \to A$	$\to$i 2–3	
5	$A \to B \to A$	$\to$i 1–4	

This operation always preserves the conclusion, and the conclusion will be most often the part of the proof that we match, so we write it explicitly:

$$\Gamma : \left(\begin{array}{c} \vdots\,\mathbf{G} \\ A \end{array} \longmapsto \begin{array}{|c} \vdots\,\mathbf{F} \\ A \end{array} \right)$$

The translation operates in a context Γ. This context is a list of assumptions accompanied by labels that can be used in the proofs **G** and **F**. The assumptions that are discharged in the proof are no longer in the context. Sometimes for clarity we will mark assumptions available in particular branches of proofs and discharged after by additional brackets. Below we give an example of a translation of proof styles in a non-empty context:

$$[A]^x, [B]^y : \quad \cfrac{[A]^x \quad [B]^y}{A \wedge B}\,\wedge i \quad \longmapsto \quad \begin{array}{|ll} A \wedge B & \wedge\text{i } x, y \end{array}$$

We define the translation operation inductively via the translation rules in Figure 6.2. The translation rules match the conclusion and the rule used and give a rule to build the flag style proof. All new labels introduced by the translation operation are fresh identifiers. The usual presentation of flag style proofs is with line numbers and rules that reference those numbers, but in our translation we will use identifiers. An implementation may render such proofs with lines numbered in the customary way, and we do indeed provide this in our ProofWeb implementation as described in Section 6.6.

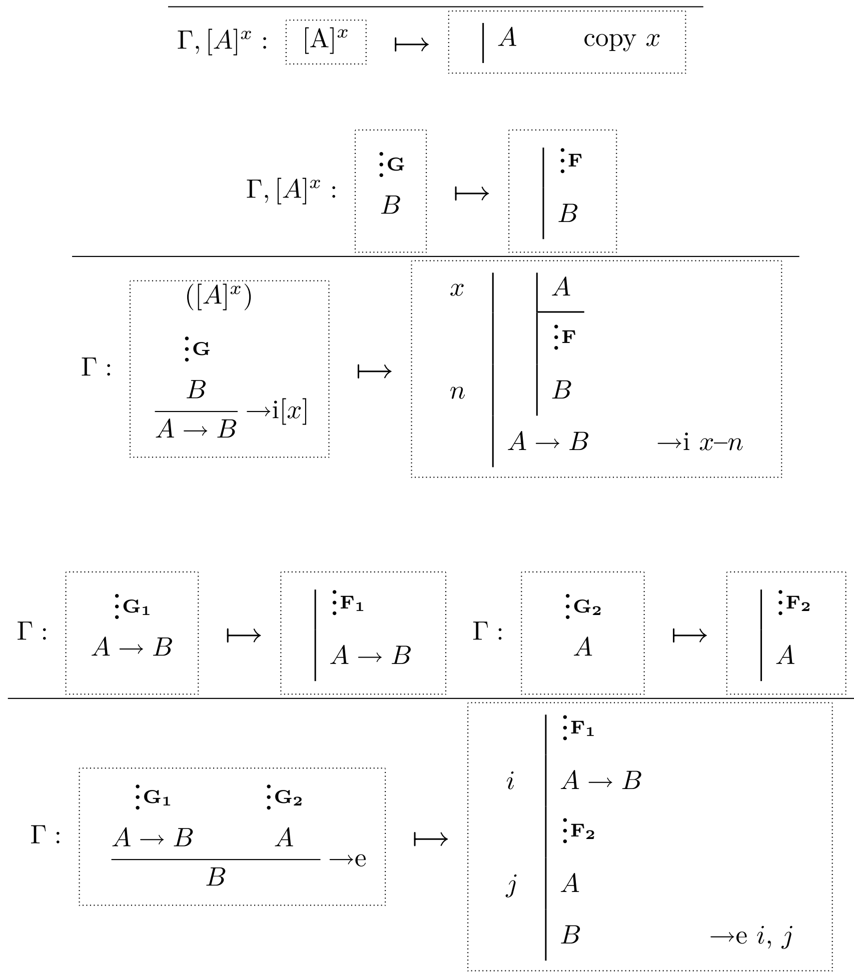


Figure 6.2: The translation rules for minimal propositional logic.

The first rule translates the use of an assumption. We replace the use of an assumption with a `copy` line, and label this line with the name of the assumption variable.

If the derivation ends with implication introduction we translate it to the implication introduction rule in the flag style. We use the name of the introduced assumption in the tree style as the label of assumption line in the flag style. The assumption $[A]$ is not in the context since it is discharged, but for readability we mark it in brackets in the tree style proof. This means that the proof **G** can use this assumption. We provide fresh identifiers for new lines.

Implication elimination is analogous. We do not need to introduce a flag for the subtree of the tree style proof. This is what makes the depth of flag proofs much lower then the depth of tree style proofs.

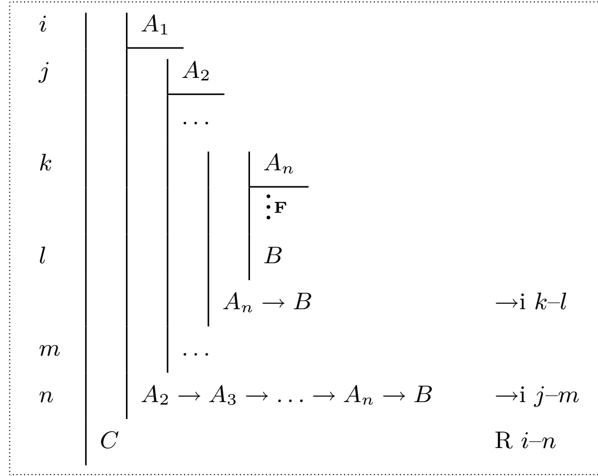
6.3 Translating proofs in more complicated logical systems

To translate a proof in tree style of an arbitrary deduction system we will first translate it to a non-optimized proof.

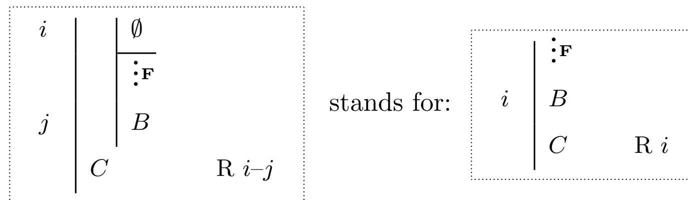
We often need to open a number of flags depending on a list of assumptions. This is why we introduce a shorthand notation. We will write flags with a *list* above the assumption line to denote opening a number of flags. The last flag is opened with the rule provided in the shorthand notation, while all other flags are introduced one by one using implication introduction:

i	$A_1, A_2, \ldots, A_n$	
	$\vdots$ F	
j	B	
	C	R i–j

This stands for:



For a list with just one assumption this is equivalent to opening one flag with just the given rule. For a flag with an empty list of assumptions no flags need to be opened:



We show the translation of a given tree style proof in terms of a general schema. This schema will be instantiated for every rule of the logic. Given a rule R that proves the formula B from the tree style proofs $G_1, G_2, \ldots, G_n$ that have conclusions $A_1, A_2, \ldots, A_n$, which discharge assumption lists (possibly empty) $S_1, S_2, \ldots, S_n$ we recursively translate all subproofs to generate the final flag style proof (Figure 6.3). The subproofs $A_1, \ldots, A_n$ can use the assumptions from their appropriate lists and this is marked in the schema by brackets. An example of instantiation of the schema for a rule for is given in Figure 6.4.

6.4 Simplification of obtained proofs

We can remove many of the copy lines by 'path compression', i.e., if a copy line is not the last line under a flag, the copy line can be removed and all further references should be renumbered to refer to the line that was copied:

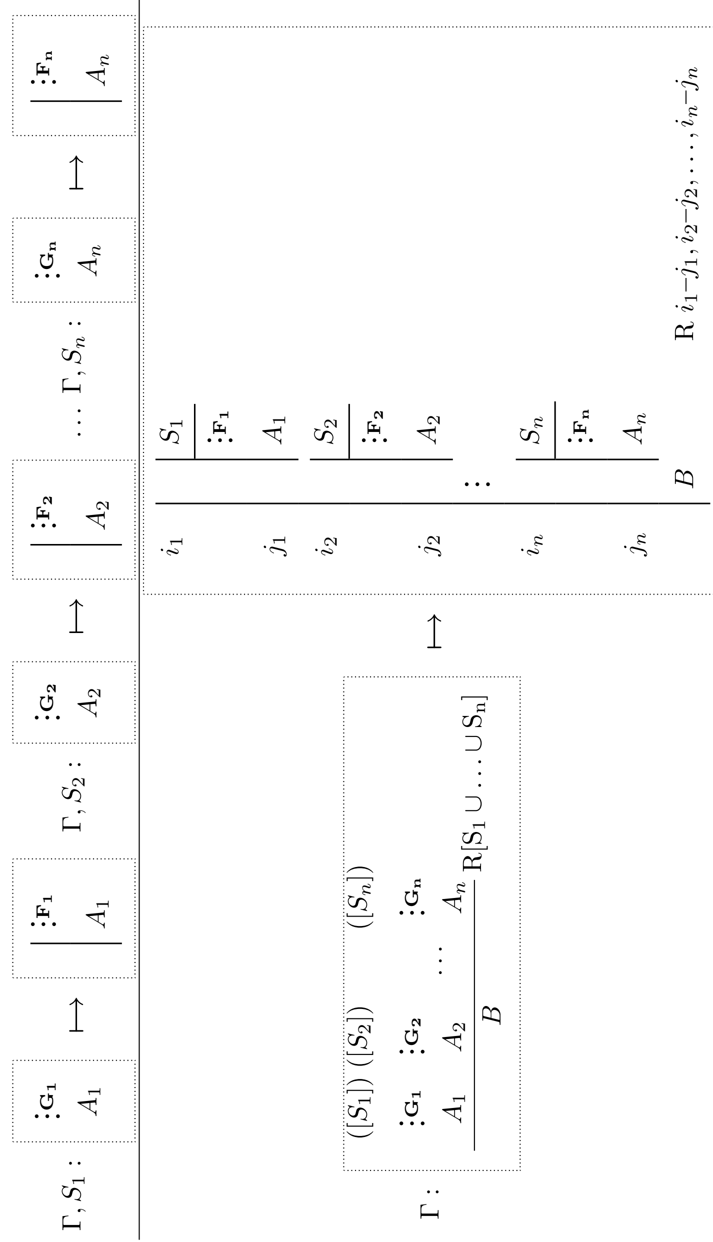


Figure 6.3: The general schema for translating a rule of the logic.

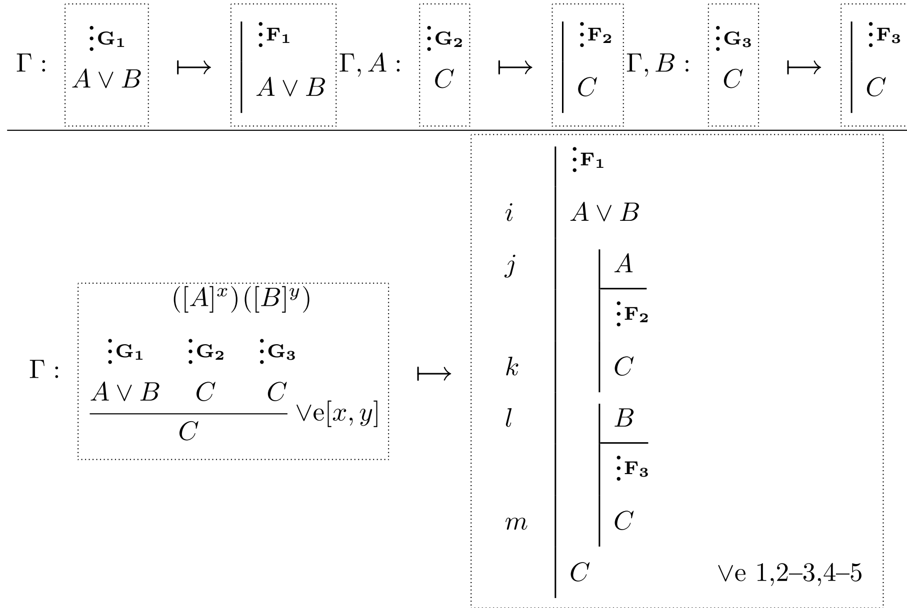
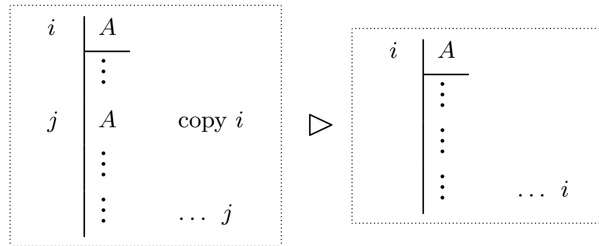
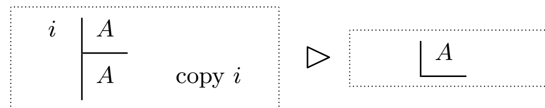


Figure 6.4: Example of the general schema instantiated for ∨-elimination.



If the copy line is the only line under a flag and is the copy of the assumption introduced under this flag, the copy line can be removed. This creates proofs that resemble customary Fitch deduction drawing style:



Theorem 6.4.1 *The use of the translation followed by performing the above simplifications on a correct Gentzen style natural deduction proof results in a flag style proof that is a correct Fitch style natural deduction proof with the same conclusion and the same rules.*

Proof [Sketch] The conclusion and the rules are preserved by all steps of translation and simplification. The simplifications do not change any of the rules or lines they operate on. The translation of any correct Gentzen rule is a correct

Fitch rule. The proof proceeds by verifying the correctness of the translation of all natural deduction rules from [KRW+08]. □

6.5 Simplification of forward proofs

One of the main advantages of flag style proofs over tree style proofs, is that the flag proof is typically almost linear, with very little nesting and therefore much easier to present on paper. For completed natural deduction derivation the proof that we obtain by translation is mostly flat, with nesting introduced only for assumptions. Our translation is also able to work with incomplete proofs. For incomplete proofs done in a backwards manner (starting from the conclusion) the tree style proof corresponds naturally to the flag style proof. This is not the case for forward proofs. For example in tree style:

$$\begin{array}{l|l|ll} i & A & & \\ \hline j & & B & \\ \cline{3-3} k & & A \wedge B & \wedge\text{i } i, j \\ & & \vdots & \\ & & C & \end{array}$$

The line labeled k is obtained by $\wedge$-introduction from lines i and j. To represent this proof in Gentzen style natural deduction we need a cut with a branch where $A \wedge B$ is an assumption:

$$\dfrac{\dfrac{[A] \quad [B]}{A \wedge B}\wedge i \qquad \dfrac{\begin{matrix}([A \wedge B]^x) \\ \vdots \\ C\end{matrix}}{A \wedge B \to C}\to i[x]}{C}\to e$$

The cut in the above proof cannot be eliminated until the proof is completed. However, this is not the case for flag style proofs, where this kind of cut can be eliminated without influence on the rest of the proof (assuming the rest of the proof is translated as well).

We want to give a mechanism that allows translating the above tree style proof with a cut to a flag style proof without a cut. The use of cut is a general technique; it is often used for inserting a subgoal that can be used further in the proof. This is why we will eliminate all the implication cuts that could have been obtained in this way. To do this we present the rewrite rule in Figure 6.5, which can be applied only if line l is not used further in the proof.

Theorem 6.5.1 *The rewrite system including the above rewrite rule terminates.*

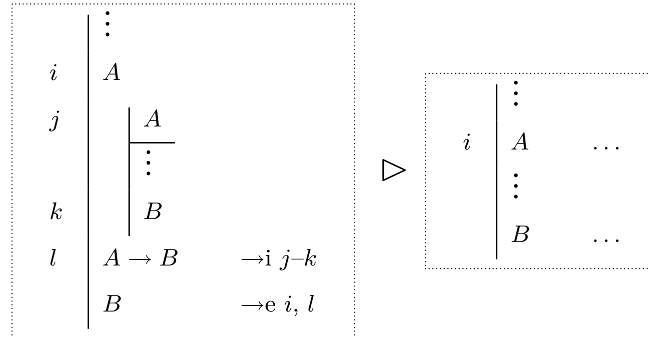


Figure 6.5: Rewrite rule for eliminating explicit cuts from a Fitch deduction.

Proof [Sketch] By induction on the number of flags. □

Theorem 6.5.2 *The rewrite system including the above rewrite rule is confluent.*

Proof [Sketch] If it is possible to apply a rule at two places in a proof, the two places are associated with two flags. Either one of the flags is under the other or they are in separate parts of the proof and thus independent. If one of the flags is under the other, it has to be inside the incomplete proof part of the rewrite rule. In that case the rewrite only moves the whole incomplete proof and thus the rewrites also are independent. □

We see in Figure 6.6, how the application of this rewrite rule makes the translation of a Coq proof from a Gentzen tree style proof into a flag style proof with a small number of nested flags.

6.6 Implementation for Coq proofs

The implementation of Coq keeps a proof tree. This is a recursive OCaml structure, that holds a goal, a rule to obtain this goal from the subgoals, and the subgoals themselves. It is not just a tree structure, since a rule can be a compound rule that contains other proof states. Tactics and tacticals modify the proof state. Coq includes commands for inspecting the proof state. `Show Tree` shows the succession of conclusions, hypotheses and tactics used to obtain the current goal and `Show Proof` displays the CIC term (possibly with holes). The output of these commands was not sufficient to transform the proof state in other formats. We added a new command `Dump Tree` to Coq that allows exporting the whole proof state in an XML format. An example of the output of the `Dump Tree` command for the Coq example from Section 6.1.1 is:

```
<tree><goal><concl type="forall n : nat, n <= n"/></goal>
  <cmpdrule><tactic cmd="intros"/>
```

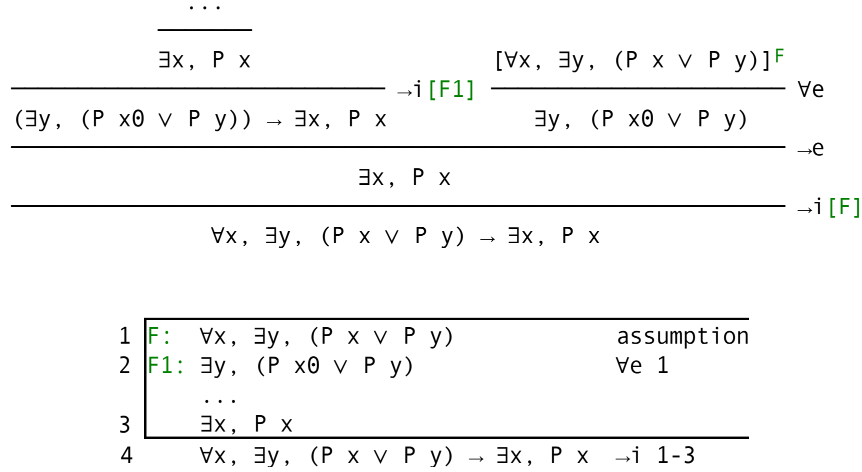


Figure 6.6: An incomplete proof in Gentzen natural deduction and its translation to a Fitch deduction, as rendered by the implementation.

```
    <tree><goal><concl type="forall n : nat, n <= n"/></goal>
      <cmpdrule><tactic cmd="intros"/>
        <tree><goal><concl type="forall n : nat, n <= n"/></goal>
          <rule text="intro n"/>
          <tree><goal><concl type="n <= n"/><hyp id="n" type="nat"/>
        </goal></tree></tree></cmpdrule>
      <tree><goal><concl type="n <= n"/><hyp id="n" type="nat"/>
    </goal></tree></tree>
  </cmpdrule><tree><goal><concl type="n <= n"/><hyp id="n" type="nat"/>
</goal></tree></tree>
```

This is the proof tree that corresponds to the incomplete Fitch proof on page 88.

The communication between ProofWeb and Coq is very narrow. The `Dump Tree` command is all that had to be added to Coq to allow our system to convert proofs, and its implementation only took a small amount of OCaml code. This code has now been integrated into the Coq code base, which means that the `Dump Tree` command is standardly available in Coq from version 8.2.

6.6.1 Transforming Coq proof state in a flag style proof

Our system is intended to be used with simple tactics that correspond to the inference rules of first order logic, so currently we forget the information generated by automated tactics (the content of compound rules). We first transform the tree to a non-optimized flag proof. For every node of the Coq tree we create a new flag. This flag first contains all the assumptions. The notation presented in the previous sections where a flag is allowed to have an arbitrary number of

assumptions is also used in the implementation; at a later step this gets translated according to the meaning of the notation. Then if a tree has subgoals, the transformed subgoals are attached. Otherwise, if the goal is not proved ellipses are attached. Finally the flag contains a line for the conclusion of the Coq node.

When rendering a flag style proof that was translated from a tree style proof done with the Coq tactics, the tactic names are printed in a special way. For tactics that match natural deduction rules, the names are changed to their natural deduction names. Furthermore we add the consecutive line numbers on the left of assumption lines and conclusion lines. We then replace references to labels with the appropriate numbers.

As an example of a flag style version of a serious Coq proof, consider the following proof from the Coq standard library:

```
Lemma leb_complete : forall m n:nat, leb m n = true -> m <= n.
Proof.
  induction m. trivial with arith.
  destruct n. intro H. discriminate H.
  auto with arith.
Qed.
```

This proof is rendered by ProofWeb as:

```
1         ∀n:nat, leb 0 n = true → 0 <= n          trivial[with,arith]
2   m:    nat                                      assumption
3   IHm:  ∀n:nat, leb m n = true → m <= n          assumption
4   H:    leb (S m) 0 = true                       assumption
5         S m <= 0                                 discriminate[H]
6         leb (S m) 0 = true → S m <= 0            intro[H] 4-5
7   n:    nat                                      assumption
8         leb (S m) (S n) = true → S m <= S n      auto[with,arith]
9         ∀n:nat, leb (S m) n = true → S m <= n    destruct[n] 6,7-8
10        ∀m:nat, ∀n:nat, leb m n = true → m <= n  induction[m] 1,2-9
```

6.7 Concluding Remarks

The future work of this chapter is to develop a *luxury* proof interface, as described in Section 6.1.1, for a serious proof assistant. The main difference with the ProofWeb system will then be that the system can also *input* a declarative proof. The declarative proofs then becomes the text that the user works on.

We implemented a rough prototype of a luxury proof language for the HOL Light system, and the approach seems to work quite well there. Currently we are redoing this system in a more systematic and structured manner. This experiment shows that our approach to converting procedural proofs into declarative proofs is not tied to any Coq specifics. It works just as well, and in exactly the same way, in a HOL environment.

A difference with ProofWeb will be to have one further rewrite rule for proofs. In the declarative language of the Mizar system there exists the `consider` statement that is used for existential elimination. If one knows that there exists an x that satisfies $P[x]$, one can write:

```
proof
  ...1
  consider x being A such that P[x] by ...2;
  ...3
  thus Q by ...4;
end;
```

This can be seen as a condensed version of

```
proof
  ...1
  proof
    let x be A;
    assume P[x];
    ...3
    thus Q by ...4;
  end;
  thus Q by ...2;
end;
```

In the case of the ProofWeb system we did *not* want the system to rewrite the latter to get the structure the former, as it would not leave Fitch-style proofs the way that student users would expect them to be. However, in a system for significant formalizations, an optimization like this will be essential.

We claim that a *luxury* proof interface – that is, an interface in which the user edits a declarative proof, but also can ask the system to extend that proof by executing tactics – combines the best of the procedural and declarative worlds. We expect that it will be straight-forward to implement such an interface using the methodology presented in this chapter.

Chapter 7

Cooperative repositories for formal proofs - a wiki based solution

7.1 Introduction

7.1.1 Motivation

Nowadays, most proof assistants follow the *interactive paradigm*: the user enters the statement of a theorem; the system checks the well-formedness of the statement. The user then enters a proof commands and the systems responds by validating the command and returning the remaining goals to be proven. This process is then iterated until the proof is complete. Thus, the resulting sequence of commands, normally called a *proof script*, has barely any meaning without the succession of *proof states* it yields. However, most formal developments only consist of the bare proof script, maybe with some comments.

Two solutions are available for people who want to understand the proofs better: HTML rendering and local execution. With web rendering, the proof scripts are processed by a documentation tool that turns the files into HTML documents and provides some facilities such as hyperlinks from symbols to their definition, indexes of symbols and searching. Some even provide pretty-printing of comments, rendering of mathematical formulae.

But to understand the proof script itself, one has to first locate and download the files containing the proofs, then install the proof assistant, and finally run the proof assistant on the file to inspect the sequence of proof states. When doing this, one loses the ability to browse the code using hyperlinks, and it can sometimes be complicated to get the proof assistant to run on one's computer in the first place.

Our work presented in Chapter 4 shows that the *Asynchronous DOM Modification* web technology (sometimes referred to as *AJAX* or *Asynchronous*

Javascript and XML [Pau05]) can be used to build a web interface for interactive proof assistants: ProofWeb. This means that users can use their favorite web browser to run proof assistant sessions, so they can themselves perform the checking of the formal proof. However this work still lacks essential features: it is not designed to support multi-file developments properly, no proper HTML rendering is implemented and there are no tools to store and retrieve multiple versions of files.

A popular web architecture which supports all those features is called a *wiki* [Cun06]. The wiki concept actually covers many implementations, but all are aimed toward a *cooperative authoring* of knowledge repositories. The key feature of a wiki system is the ability to follow an 'edit' link and be able to immediately modify and publish a new version of the page being viewed.

The popularity of wiki based solutions made us think of integrating the web interface for proof assistants within a wiki repository: the web interface would be used as the viewing and editing window for files containing proof scripts. The main difference between our work and common wiki usage is that our framework handles formal content that requires a consistent environment (i.e. file dependencies) to run interactive sessions. Thus (semi-)automated maintaining of cross-file consistency is crucial.

7.1.2 Related Work

Most proof assistants already have a more or less user-friendly way of *rendering* formalisations as a set of interlinked web pages. Some provide a standalone tool that allows users to render their own files: this is the case for Coq. Isabelle [NPW02] can create a rendered version of a theory while processing it and it also provides a way to navigate the dependency graph of multi-file developments.

The Mizar [Muz93] system has a proof repository called the MML (Mizar Mathematical Library) [BR03]. This repository is modified by human editors: duplicates are eliminated, results are moved to appropriate sections, new sections are created. This gives the MML a monolithic and consistent look [RT03]: it is handled as an encyclopedia, where new content is added with many authors but one central editing committee. There is a project of gathering the definitions and theorems from the MML to create an even smaller and more organized Encyclopedia of Mathematics in Mizar (EMM). The rendering tools provided for the library are not available for the common user to work with his local development. The MML (and its associated journal JML) is the *de facto* standard way to publish a Mizar proof.

The Logiweb System [Gru06] provides a way to submit and retrieve articles from a network of distributed repositories. It allows reliable cross-references to fixed versions of already published articles. However it still relies on a locally installed checker to verify articles before submitting them.

The HELM [APCS01] (Hypertextual Electronic Library of Mathematics) and the Whelp [AGC$^+$04] search engine give users a good rendering of distributed formal libraries along with a powerful search engine. The Matita [ACTZ07]

proof assistant offers native support for queries and browsing of these libraries.

The Logosphere [SPK+] project aims at presenting developments from different proof assistants (Nuprl, PVS, ...) using a unified framework.

The Mizar and Coq proof assistants already have wiki web sites for their documentation. The Mizar wiki [Miz08] is an official, general purpose web site whereas the Coq wiki, called Cocorico [Niq08], is a community website more dedicated to the sharing of specific knowledge about Coq usage, hints and tips.

7.1.3 The future of proof interfaces

The aim of this new web-based cooperative proof environment is to provide a complete solution for the development of formalized mathematics or software verification. It brings together the availability of a web-interface with the accessibility of a web-rendered archive.

The unique feature of this environment is that, beyond the separation between the raw editable and rendered read-only versions of the files (a characteristic of wiki environments), both of those versions can be processed by the proof assistant at the request of the user, giving him more information as to how the proof script works. Standard online formal libraries tend to treat proof scripts as unimportant (many of the rendering tools completely ignore proofs), whereas with interaction the proof contents can give the user insights on how the proof was made. This way the proof script is not any longer write-only.

Therefore, this environment provides a useful tool for specialists to communicate about proofs with a broader audience: non-specialists, general audience. It provides a simple way for article writers to give referees easy access to their formal development.

The repository is also a convenient way for proof authors to work from everywhere simply using a network access, and to learn from others' proof idiosyncrasies.

The repository can also be used for education about proof assistants and formal logics as a further extension of the system presented in Chapter 5. A permissions system can allow students to cooperate on multi-files projects and their supervisor to provide guidance.

7.1.4 Chapter contents

In the rest of the chapter we present the technologies relevant for creating a wiki for proof assistants (Section 7.2), and the components that are relevant. Then we present the global architecture of our system and discuss our library consistency policy (Section 7.3). We describe our current prototype of the wiki (Section 7.4) and discuss performance and security issues. Finally, we give our road map towards a more stable system and present our conclusion (Section 7.5).

7.2 Web Technologies

In this section we will describe what *wikis* are and what advantages they offer in comparison with static web pages. Our development also depends on *asynchronous DOM modifications* technique, that was already used for the creation of the interactive interface and was described in Chapter 4.2.

7.2.1 Wikis

Wikis are dynamic web sites that behave as static ones: they contain a number of fixed pages that can link to each other. The unique feature of wikis is that each of those pages includes an 'edit' link that displays the contents of the page in an editable textbox (or in an editable WYSIWYG box in advanced implementations). This page allows the user to actually modify the contents of the box and publish the new version on the web site, simply using a web browser.

This is what makes wikis dynamic: this online edition feature requires files to be served in a different way than just static HTML files. Usually they are stored in a database system rather than in a filesystem. Unfortunately, current proof assistants do not include the functionality to access such databases.

The wiki technology is now very popular, especially for documentation of software tools: one uses a very small, general (and somewhat imprecise) documentation which is then improved by visitors when finding inconsistencies, errors and missing items. The most famous wiki is obviously the Wikipedia web site, which aims at being an online encyclopedia where information is added by visitors. In the 14 most popular languages on Wikipedia, there are more than 100,000 articles available in each. This shows that the wiki architecture can support large amounts of data and heavy activity.

The file format used by wikis is usually a simplified markup and formatting language, tailored to make references to other pages simple to add. Wikis usually have a permissions system to forbid reading or writing for particular users. Most of them also allow modification by unregistered users. In that case the IP address of the client is used as an identifier.

Wikis also offer the possibility to explore the history of any article: what was modified, when. They allow renaming of pages, and provide indexes of available pages. They usually include tools that allow searching the page database.

Those features match our requirements for a content management system to be usable with the ProofWeb framework.

7.3 Architecture

7.3.1 Main Components

The system uses proof assistants with some of their companion utilities and some web serving utilities. The first element we need is the interactive toplevel

of a prover. It is required on the server side to be able to verify the input of the user in an incremental manner and to go to particular positions in them.

For efficiency reasons some provers allow compilation of their input files. Such files can then be quickly reused, without verifying all proofs contained in them again. For such provers we want to use the compiler on the server to generate a compiled version of proofs that are saved.

Many provers include documentation generators, that process raw prover input files and generate rendered output. The output of a documentation generator is usually HTML or PDF format. Links between files are created, different conceptual elements of the prover input are colored in different color, and sometimes mathematical formulas are rendered in a graphical way.

We need to keep a history of versions for every prover file. Usually, collaborative developments are done using version control systems. The source files are kept in the repository and each user has to build compiled or rendered versions him/herself.

Not only would we like the source prover files to be stored for all versions, but also the rendered and the compiled files (for provers that include this concept). This way users can see a rendered version of older versions of files. Referring to older versions of compiled files will be discussed in section 7.3.3.

Wikis already include some kind of versioning of the files they contain. Generally file versions are numbered in sequence. For every change, the user that made the change is stored with the file, and viewing changes is possible. The history mechanism is more limited than the ones provided by file versioning systems, but the simplicity can also be an advantage: in particular wikis do not include branches, tags, etc and the casual user does not have a good understanding of it.

A wiki infrastructure will be used for tracking changes done by users and allowing them to see the history of files and changes. It needs to store files in a way that is accessible by the prover toplevel. The wiki should allow generating indexes and searching for terms. Most wikis generate text indexes and allow searching for text only, whereas prover scripts are highly contextual.

Finally we need a web part that allows interactive edition of a proof script in a way that resembles local work. Additionally we would like to be able to step interactively over the proof regardless of whether we are in view or edit mode. The ProofWeb framework can be modified to allow those two modes.

7.3.2 Global Design

Our architecture is composed of a web server running a modified version of a wiki that redirects some requests to a ProofWeb server (Fig. 7.1).

Editors of most wikis are standard HTML textboxes, and the flat text includes special markup for marking links and elements that should be formatted in a special way in the read-only version. Recent wikis allow WYSIWYG editing in an editable IFrame. The HTML formatting introduced by user's browsers is combined with wiki links to create the read-only version. In our architecture we embed the ProofWeb editor as the editor of the wiki. This way, the user can edit

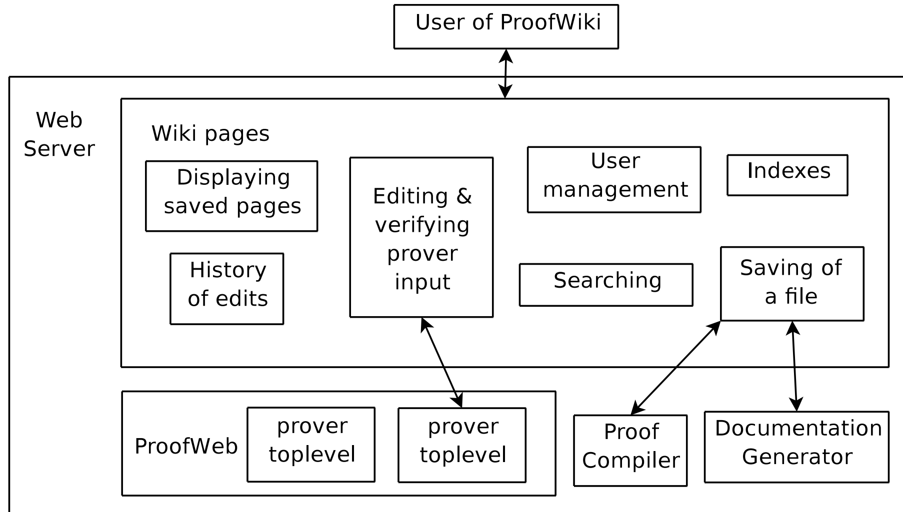


Figure 7.1: Our architecture.

the script in an interactive way seeing the output of the prover. Additionally we include a readonly version of the ProofWeb interface for the read-only version to allow examining the prover state at any point in the buffer.

The next necessary change is the way the wiki stores users files. For every saved file the wiki tries to compile it and to make a rendered version of it. The rendered version should be linked with the original, and is therefore stored in the same way the wiki stores the original in its database. Whenever the user requests the file for viewing or editing one of those two versions is used. The compiled version is stored as a standard file in the filesystem in order to make it available to the compiler and to the toplevel used in prover sessions on files that refer to this compiled file.

This change in the wiki behavior should not prevent users from storing and editing standard wiki pages in the repository. Those would include textual description of the formal content, discussions, tutorials with hyper-links to formal content.

The documentation generators of provers have to be able to generate a wiki compatible output. The format that a wiki displays is usually very close to HTML which many prover documentation generators already support. The important difference with respect to HTML is that since we will process the rendered version of the script we need to be able to distinguish active parts of the file from comments.

7.3.3 Consistency Issues

In ordinary wikis, links to a nonexistent page lead to a new editable page. This is perfectly acceptable for the usual informal content but not for a formal

development referring to another: think of it as a program missing libraries.

Moreover, we need to make sure that dependencies are always consistent. Files in the database can depend on each other, sometimes in an indirect (transitive) way. First of all, we want to require all saved files to be valid (compilable); they can still contain incomplete proofs terminated with the Coq `Admitted` keyword or its equivalents for other provers. For a valid saved file we want to ensure that the current version remains valid after changes to the files it relies on. Some provers already include compatibility verification mechanisms. Coq stores the checksums of files to ensure binary compatibility between compiled proofs. To solve the problem, we have to consider the static and the dynamic approach.

The dynamic approach is the convention that a file always refers to the latest versions of other files. It means that saving any change to a file will induce a costly recompilation of all files that depend on it. Another problem is that changing definitions deep inside a library will make many developments incompatible and correct files will stop working. Saving only valid files does not solve this problem since the objects they contain (their interface) might be modified too. This approach also makes older versions of existing files immediately obsolete.

The opposite approach is static linking, where a saved file always refers to the same version of other files. In other words, we never change a file, but rather add a new version of that file, with a fresh name. This means that the user will have to manually update the version number of files that are referred to if newer versions of those become available. The main advantage of this approach is that of integrity: provided you can safely assign new version numbers, you can enable concurrent access. Moreover, changing a file will never *break* another file. However, when changing a file deep in the library, one has to manually modify all the files in the dependency chain between that file and the files in which the changes should be reflected, which can sometimes be quite heavy.

7.3.4 Towards a hybrid approach

We believe that the static approach is a more adequate way to store older (historical) versions of a given file, whereas up-to-date files should use the dynamic dependency approach. This way, older versions of files still make sense by statically referring to older versions of files they depend on. The latest versions can remain up-to-date with their immediate dependencies by being dynamically linked to them, i.e. recompiled when new versions of those files are saved. It might happen that such a file might not be valid anymore because of changes made to its dependencies: to keep validity we have to make it link statically to the suitable previous version.

To help with this version compatibility issue, one may use a three-colour scheme (it is not implemented in our prototype):

- A file is labelled as *red* (i.e. outdated) if it depends statically on an older-than-latest version of another file.

- A file is labelled as *yellow* (i.e. tainted) if it depends only on the latest versions of other files, and one or more of those files have a yellow or red status. Yellow status thus tracks the files which are indirectly lagging behind.
- A file is labelled as *green* (i.e. up-to-date) if it depends only on the latest versions of other files, and all those files are also labelled as up-to-date (green status).

The separation between the yellow and red files comes from the fact that red files have to be manually updated to become green again (i.e. by creating a new version of them), whereas yellow files might be fixed by updating the red files that taint them.

The switching to red status can be automated by rewriting `Require` statements on-the-fly to make them refer statically to the last suitable version of the file depended on. This means that fixing a red file can give red status to yellow files that it was tainting, thus pushing the problem upwards in the dependency tree.

We imagine two ways in which renaming files can be implemented: If the user wants to export a file together with its dependencies from the repository, a mechanism can be used to convert long file names (with version number) to short ones. The case might arise where a file would refer, directly or by transitivity, to an old version of itself. We can either forbid this or generate fresh file names using standard suffixing techniques.

The procedure of updating the prover itself, although intended to be rare, will be critical. Here a decision will have to be made whether to port all possible versions or only the newest versions of each files and their dependencies. The current system is clearly not yet designed to enable such updates without putting it offline and porting files manually, but such a feature should definitely be designed and implemented.

7.4 Prototype

7.4.1 Implementation

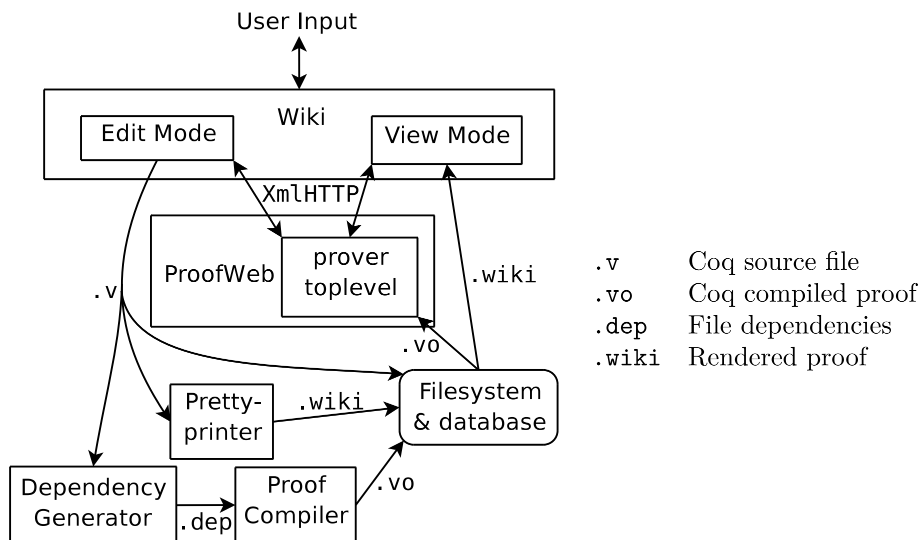
To experiment with our idea, we have created a prototype implementation based on off-the-shelf components as much as possible. We chose Coq as our target proof assistant. We used the MediaWiki code-base, the `coqdoc` documentation generator for Coq and ProofWeb for Coq (Fig. 7.2). The `coqdoc` tool was modified to generate wiki format rendered pages.

When the user opens a page of our wiki, he/she is presented with a viewing page where the usual contents area is replaced by three subframes. One frame shows the rendered version of the current document, the second one shows the current proof state and the third one displays the Coq error messages.

The user may press the ‘up’ and ‘down’ buttons to step over the proof and examine both the proof state and Coq messages. The background coloring

scheme described in Section 4.7 allows the user to keep track of the part of the script that was already processed.

The proof is rendered, that is identifiers are colored and linked to their definition, mathematical LaTeX comments are rendered, links to internal wiki pages lead to those wiki pages and links to Coq standard library objects lead to the documentation on the Coq website (Fig. 7.3).



`.v`	Coq source file
`.vo`	Coq compiled proof
`.dep`	File dependencies
`.wiki`	Rendered proof

Figure 7.2: Data flow diagram for our prototype wiki

The page also includes standard wiki elements, one of which is the 'edit' button. When the user starts editing the page, a similar page is presented, but with the raw proof script (no rendering) in a modifiable text box. The user may modify the script and use the 'up' and 'down' buttons to step over the proof in a similar way as in the view mode (Fig. 7.4). The processed part of the buffer is frozen.

When satisfied with his work, the user can save the proof. The contents of the buffer are processed in three ways:

- The raw script is saved in the database, to be used by following edits.
- The file is compiled and the corresponding `.vo` file is stored in the filesystem for processing of files that would include it using `Require` statements.
- A rendered version is generated by `coqdoc` and saved in the database to be displayed in view mode.

The data is being duplicated for the rendered version. We need all of these three versions stored. The original version is needed for further editing and feeding to the proof assistant, the rendered version is needed to be able to

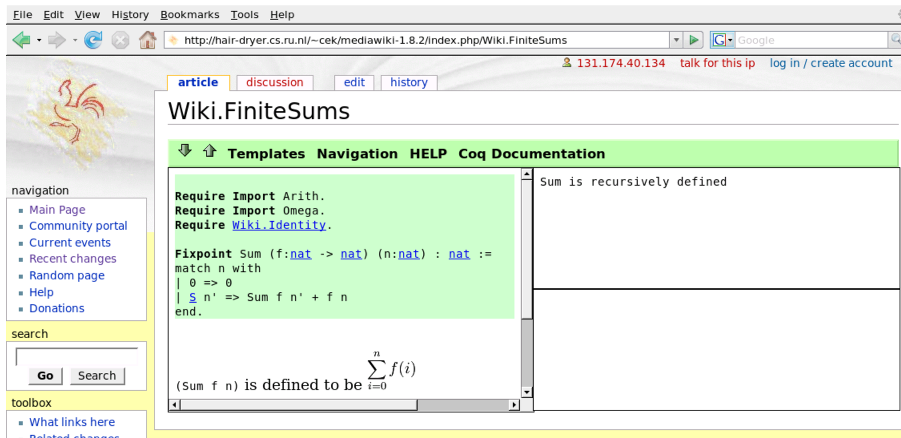


Figure 7.3: Screenshot of the prototype showing the rendered version of a a Coq file. The verified part of the edit buffer is colored. The state buffer shows the state of the prover, there are no Coq warnings.

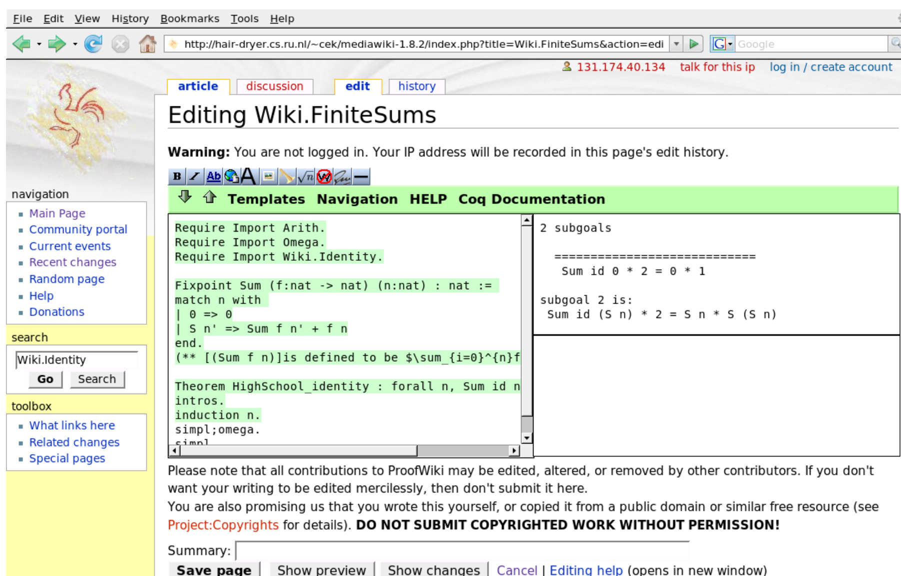


Figure 7.4: Screenshot of the prototype showing the editing of the corresponding source file. The verified part of the edit buffer is colored and frozen.

access it fast and the compiled version is necessary to be able to use the finished theory in other developments.

The user can see the history of any page as well as display the differences between the sources of any versions, using built-in MediaWiki routines. The textual search mechanism allows to query the source Coq files for any terms.

7.4.2 Security and Efficiency

The security and efficiency of the server are crucial since unavailability of the proof wiki would make the users not only unable to work, but also unable to access their own files. The security and efficiency of the architecture relies on the security of ProofWeb, the underlying wiki, the compilation and rendering processes and the communication mechanism.

The security and efficiency of ProofWeb are described in detail in Chapter 4. The solution adopted there is *sandboxing*: the ProofWeb server process is run in a chrooted environment as a non privileged user without network access. Here we add some features inside the sandbox while keeping the wiki infrastructure out of it.

The sandboxing which is a part of the ProofWeb architecture makes it reasonably safe. The efficiency is divided by the number of users, but it is straightforward to distribute prover sessions over a set of machines. In a wiki environment we are additionally running a Coq dependency generator, compiler and renderer. We run these processes in the same sandbox as the prover toplevel, so we expect them to be comparably secure. However for big formalizations performing the compilation can be costly. Especially when many files depend on each other, modification of one of them may require recompiling numerous proofs. We expect this to be the main bottleneck of a wiki for proofs. Although this can also happen with local proof interfaces, here multiple parallel sessions might overload the system for a longer period.

The proof text verification that ProofWeb does is independent from page serving performed by the web server and MediaWiki, so we can analyse the latter separately. Wikis are quite secure and efficient. At the time of writing this chapter, Wikipedia provides more than 350 servers that have more than 3 million users and 1.6 million articles without significant efficiency issues. An issue that is often a problem in wikis is vandalism. Disallowing editing by particular users or IP addresses is a common practice, and is already supported in MediaWiki. Discovering vandalism in our framework may sometimes be easier than in standard wikis, since incorrect proofs no longer compile.

The data that is being transferred to and from the wiki is usually public, still the communication mechanism can be secured by configuring the web server that serves the wiki to use HTTPS.

If the wiki is secured properly we do not expect crackers to be an important issue. However the efficiency seems to be quite fragile, in particular it seems that our architecture is quite vulnerable to denial-of-service attacks.

7.4.3 How to integrate other provers

Although our prototype has been implemented for Coq, we do not rely on any specific Coq feature. We think that extending the wiki to other provers is feasible provided the following functionalities are available: a wiki compatible documentation renderer, a dependency generator, ProofWeb support and optionally an index generator.

The renderer does not need to be sophisticated, the only mandatory feature of the renderer is distinction between active proof script from comments. Other features like syntax highlighting and links are not necessary, although they allow a more wiki-like interaction.

The wiki needs to know how to call the dependency generator of the prover, to know what files need to be updated if a particular file is modified. If the prover has a compiler, the wiki needs to know how to compile proofs. The wiki also needs to be able to identify statements that refer to other files during the interactive session.

An optional element is an index generating utility. It is needed for the wiki to distinguish concepts from the added prover's syntax. This allows not only nice index pages in the wiki, but also searching for particular prover objects, like only definitions or theorems.

Finally ProofWeb needs to be able to interact with the prover. It already provides a limited support of a number of additional provers. To extend it to a new one, the client part needs to know how to find the ends of complete prover commands and the server part needs to know how to interface with the prover process, in particular it needs to know how to check if commands succeed and how to undo. The details of extending ProofWeb to a new prover are described in Chapter 4.

7.5 Concluding Remarks

The current architecture of the prototype is not satisfying since it relies on a double storage of files: in the database, and on the disk. We are also limited by the way MediaWiki handles its name space. If we adopt the static system where files are never modified, it can be worthwhile to consider moving all the data to the file system, and adopting an architecture where we can have a better control of the name space.

The static naming will require to implement a versioning system for the substitution of `Require` statements and the distributed generation of version numbers, then the three colour scheme will be added. A mechanism for importing and exporting parts of the library will also be necessary, to allow users to have a local copy on which to work without Internet access.

A milestone in this development will be the ability to actually import the Coq standard library and official users contributions to our repository. Only then will we be able to get user feedback and report on the suitability of the repository for Coq users.

The `coqdoc` tool is able to generate index files that contain all constants occurring in the library. We could use such a feature to generate such wiki pages.

The basic textual search is very limited and proof assistants users often use query types that are far beyond the scope of textual search: find theorems about a given object or do pattern matching on theorem statements. This is sometimes implemented inside particular proof assistants, but might be achieved more generally by adapting the Whelp search engine to search our database: it will require a customisation of the indexing technology.

We could also experiment with more advanced rendering tools such as Helm and consider using MathML instead of (currently) HTML with LaTeXimages.

Although our prototype is still at a very early stage of development, our idea of combining a wiki web site with the ProofWeb interface looks definitely promising. Surprisingly, we could achieve the current result without many modifications neither to the wiki code-base, nor to ProofWeb. Most of the work was devoted to database modification and rendering.

We believe that formalising mathematics in a wiki system will foster more cooperation both within prover specific communities and between users of different provers, especially if we can make several provers coexist in the same repository. We also believe that such a project can act as a display of the work on formal proofs for a wider audience.

Chapter 8

Towards an Integrated Environment for Computer Mathematics

8.1 Vision

We started this research with a vision of formalizing the mathematics as done by engineers. We hoped that this would lead to formally verified statements from analysis and to the integration of the environments of people using mathematics and people stating theorems precisely and proving them. This was the reason to start investigating computer algebra systems. Our work has shown that this is indeed feasible, and also has also suggested that it is possible to create a system that would combine the features of computer algebra and proof assistants.

Through the course of the research our vision has changed. We now imagine that users will have a single framework for doing computer mathematics. This framework will allow computing as well as defining and proving and will be available universally via a thin client solution.

This means that all the tasks performed today with the help of computer algebra systems can be performed within this framework. The use of reflection makes it possible to effectively use computation in a certified environment. One can prove correct algorithms for computing or simplifying expressions as well as more sophisticated visualization mechanisms provided in computer algebra like graphing functions.

We also believe that access to computer mathematics should be provided universally and new results should be visible automatically. With informal steps this is done in wiki systems. We would like our system to provide all mathematical knowledge while behaving wiki-like. We think this is possible while accepting only mechanically checked proofs.

We imagine such an environment to be accessed via a thin client solution.

In the course of this thesis we investigated web-browser based environments for doing computer mathematics. This already allows users to use the full server environment without needing to care about any changes locally. Still the interface provided with a current web browser is a bit cumbersome and leaves wishes in terms of rendering of mathematics. The concurrent development of web technologies (MathML, mathematical fonts, etc) will make it possible to display mathematics in the notations users are used to.

8.2 Progress of this thesis towards the vision

The first part of this thesis presents a usable minimal framework for doing computer algebra inside a proof assistant. This shows that the decision procedures present in modern proof assistants are strong enough to build a computer algebra system that ensures absolute correctness of performed simplifications. The system is not only able to do basic computations and simplifications but is also able to handle some forms of partiality in a way similar to computer algebra. We also show how completely certified infinite precision real arithmetic can be performed in a classical setting.

The second part of the thesis aims at providing the universal access to certified mathematics repositories. We first investigated the current state of web technology and showed that it is sufficient to create a reasonably good interface to proof assistants. We explore the advantages of using a web interface instead of a local interface for showing proof assistants by providing a number of non-common provers, libraries and extensions. We then show that such an interface can be used well not only for research but also for teaching purposes.

One of the main aspects of our vision is that in the future formalizing mathematics should be done in a collaborative way. We tried to create a version of a wiki for formalizing Coq. The created prototype is quite cumbersome, but we addressed a number of important issues that come up when trying to create such an environment. We allow users to edit parts of the same repository concurrently, have schemes for tracking dependencies and provide rendered proof scripts.

8.2.1 Multivaluedness

One of our focuses was also to provide multivaluedness as expressed in computer algebra in a formal setting. We were not able to achieve it in the scope of this thesis, but we believe that a knowledge base of a computer algebra system can be extended with a formalization of multivalued functions. This would allow handling more complicated expressions like the Maple example of a complex function with multiple branches given in the introduction of Chapter 1.

Our first experiences in formalizing multivaluedness was the framework for handling partiality shown in Chapter 2. We would like to see whether a representation of multivalued functions could be done in a similar way as partiality is done in our approach. Testing this will require a lot of formalization since

there are not too many theorems in prover libraries that concern multivalued functions.

8.3 Future Work towards obtaining the Vision

We imagine the nearest future work towards a combined environment for certified computer mathematics to go in three directions:

8.3.1 Computer Algebra with Strong Semantics

First, the creation of a full-featured environment that connects the usability of computer algebra with the certifications provided by proof assistants. The prototype that we created has very few actual certified algorithms implemented as well as they come from basic fields of mathematics. Giving explicit semantics to all objects found in computer algebra and providing certified algorithms to operate on those will be a big task.

8.3.2 MathWiki

Second, the creation of a so-called 'MathWiki'. We would like to bring together the open nature of wikis with expertise in proof assistants and semantic web technologies. This would allow for building a new wiki for mathematics that would support content creation, search and retrieval of mathematical texts. The documents would have different levels of details, from high-level semantically annotated pages to detailed formalized proof scripts containing definitions, theorems and proofs in various proof assistants.

We imagine a standardized interface which supports both the high-level content and the corresponding proof assistant semantics. The background mechanism will maintain the repository in a consistent state as articles are added and revised. Searching and retrieval will be available for the whole repository using both standard text search as well as semantic searching. The MathWiki will provide an advanced environment for the collaborative creation of formalized mathematics and verified programs available simply through a web interface.

We believe that MathWiki would be very useful for various communities by providing much more coherent and precise view on high-level mathematical content for ordinary users of mathematics, as well as to popularise formally verified mathematics and facilitate cooperation between different communities by being able to make cross-links between notions in different formal systems.

8.3.3 Using a web interface for software

Many of the problems encountered when creating a wiki for proof assistants are similar to the creation of a wiki for programs. Concurrent editing of documents that depend on the theorems provided by other documents is similar to editing programming language source files that provide APIs. The issues of handling

dependencies, compilation as well as providing consistency of such a repository are definitely similar.

This will allow for providing a wiki for a number of programming languages that will allow comparing algorithms provided in different ones. A first step to creating such a repository, would be adding to MathWiki the programming languages that are closer to proof assistants, like the ones with dependent types and termination, for example Epigram and Agda.

8.4 Conclusion

To conclude, we believe that we have made significant progress in the subject of understanding the differences between proof assistant and computer algebra and bringing these two worlds together. We also investigated the relevant web technology to allow for a much wider availability of proof assistants by providing access to these via the web. The vision of a uniform environment for computer mathematics may still be too complicated to be practical, but we believe we have identified the key issues that allow for much more unified systems.

Bibliography

[ABPR01] A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau. Mathematics and proof presentation in PCoq. Rapport de Recherche 4313, Inria, Sophia Antipolis, November 2001.

[ABT04] A. Asperti, G. Bancerek, and A. Trybulec, editors. *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21, 2004, Proceedings*, volume 3119 of *Lecture Notes in Computer Science*. Springer, 2004.

[ACTZ07] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reasoning*, 39(2):109–139, 2007.

[ADG+01] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, UK, September 2001. Springer-Verlag.

[AGC+04] A. Asperti, F. Guidi, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. A content based mathematical search engine: Whelp. In J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2004.

[Agd] Agda: An interactive proof editor. `http://agda.sf.net/`.

[ALW05] D. Aspinall, C. Lüth, and D. Winterstein. A framework for interactive proof. In D. Aspinall, editor, *Proceedings of the ETAPS-05 Workshop on User Interfaces for Theorem Provers (UITP-05), Edinburgh*, page 15, 2005.

[APC+03] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical Knowledge Management in HELM. *An-*

nals of Mathematics and Artificial Intelligence, Special Issue on Mathematical Knowledge Management, 38:1–3, 2003.

[APCS01] A. Asperti, L. Padovani, C. Sacerdoti Coen, and I. Schena. Helm and the semantic math-web. In R.J. Boulton and P.B. Jackson, editors, *TPHOLs*, volume 2152 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2001.

[AR04] P. Audebaud and L. Rideau. Texmacs as authoring tool for formal developments. *Electr. Notes Theor. Comput. Sci.*, 103:27–48, 2004.

[AS64] M. Abramowitz and I.A. Stegun, editors. *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*, volume 55 of *National Bureau of Standards Applied Mathematics Series*. United States Department of Commerce, Washington, D.C., June 1964. 9th Printing, November 1970, with corrections.

[Asl96] H. Aslaksen. Multiple-valued complex functions and computer algebra. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 30(2):12–20, June 1996.

[Asp00] D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[AW02] A. Asperti and B. Wegner. MOWGLI – A New Approach for the Content Description in Digital Documents. In *Proceedings of the Nineth International Conference on Electronic Resources and the Social Role of Libraries in the Future*, volume 1, 2002.

[AZ00] A. Armando and D. Zini. Towards interoperable mechanized reasoning systems: the logic broker architecture. In A. Corradi, A. Omicini, and A. Poggi, editors, *WOA*, pages 70–75. Pitagora Editrice Bologna, 2000.

[BB85] E. Bishop and D. Bridges. *Constructive Analysis*, chapter 1.3. Springer-Verlag, Berlin, October 1985.

[BC01] H. Barendregt and A. M. Cohen. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *J. Symb. Comput.*, 32(1/2):3–22, 2001.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[BC05] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[BCC+02] S. Buswell, O. Caprotti, D.P. Carlisle, M.C. Dewar, M. Gaëtano, and M. Kohlhase. The OpenMath Standard, version 2.0, 2002. `http://www.openmath.org/cocoon/openmath/standard/`.

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. H. Siekmann, and V. Sorge. Omega: Towards a mathematical assistant. In W. McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 252–255. Springer, 1997.

[BCGH99] P. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. *Fundam. Inform.*, 39(1-2):39–57, 1999.

[BCZ98] A. Bauer, E.M. Clarke, and X. Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[BDJ+00] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, Natick, Massachusetts, 2000. A.K. Peters.

[Bew] Beweren en Bewijzen.
`http://www.cs.ru.nl/~wupper/B&B/index.html`.

[BHC95] C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: an interface between isabelle and maple. In *ISSAC '95: Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pages 150–157, New York, NY, USA, 1995. ACM Press.

[BP99] C. Ballarin and L.C. Paulson. A pragmatic approach to extending provers by computer algebra - with applications to coding theory. *Fundam. Inf.*, 39(1-2):1–20, 1999.

[BR03] G. Bancerek and P. Rudnicki. Information retrieval in MML. In *MKM '03: Proceedings of the Second International Conference on Mathematical Knowledge Management*, pages 119–132, London, UK, 2003. Springer-Verlag.

[BS96] R. Bornat and B. Sufrin. Jape's quiet interface. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 25–34. University of York, 1996.

[CDT08] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.2*. LogiCal project, 2008. Distributed electronically at: `http://coq.inria.fr/doc-eng.html`.

[CF04] L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* PhD thesis, University of Nijmegen, April 2004.

[CFGW04] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In Asperti et al. [ABT04], pages 88–103.

[CFW03] J. Carette, W. Farmer, and J. Wajs. Trustable communication between mathematics systems. In *CALCULEMUS 2003 (11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages 55–68, Rome, Italy, 2003. Aracne.

[CGG+91] B.W. Char, K.O. Geddes, G.H. Gonnet, B.Leong, M.B. Monagan, and S.M. Watt. *Maple Library V Reference Manual.* Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1991.

[CGGG83] B.W. Char, K.O. Geddes, W.M. Gentleman, and G.H. Gonnet. *The design of Maple: A compact, portable and powerful computer algebra system.* Springer-Verlag London, UK, 1983.

[CIMP03] D. Carlisle, P. Ion, R. Miner, and N. Poppelier. Mathematical Markup Language (MathML) Version 2.0 (Second Edition), 2003.

[CK07] P. Corbineau and C. Kaliszyk. Cooperative repositories for formal proofs. In Kauers et al. [KKMW07], pages 221–234.

[Cor08] P. Corbineau. Declarative proof language for Coq, 2008. `http://www-verimag.imag.fr/~corbinea/mmode.en.html`.

[CoR09] Constructive Coq Repository at Nijmegen, 2009. `http://corn.cs.ru.nl/`.

[Cun06] W. Cunningham. Design principles of wiki: how can so little do so much? In D. Riehle and J. Noble, editors, *Int. Sym. Wikis*, pages 13–14. ACM, 2006.

[Dav04] J. Davies. Wiki brainstorming and problems with wiki based collaboration. Master's thesis, University of York, 2004.

[DS97] A. Dolzmann and T. Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

[Far99] W.M. Farmer. A scheme for defining partial higher-order functions by recursion. In A. Butterfield and K. Haegele, editors, *IWFM*, Workshops in Computing. BCS, 1999.

[FS06] U. Furbach and N. Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*. Springer, 2006.

[Gir87] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[GL06] H. Geuvers and I. Loeb. From deduction graphs to proof nets: Boxes and sharing in the graphical presentation of deductions. In R. Kralovic and P. Urzyczyn, editors, *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2006.

[GN04] H. Geuvers and R. Nederpelt. Rewriting for Fitch style natural deductions. In V. van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 134–154. Springer, 2004.

[GNP03] F. Guilhot, H. Naciri, and L. Pottier. Proof explanations: using natural language and graph view, 2003. Slides for a talk at a MoWGLI presentation.

[Gon06] G. Gonthier. A computer-checked proof of the Four Colour Theorem, 2006. `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[Got05] C. Gottschall. Logic gateway, 2005. `http://logik.phl.univie.ac.at/~chris/gateway/`.

[GPWZ02] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):271–286, 2002.

[Gru06] K. Grue. Logiweb - a system for web publication of mathematics. In A. Iglesias and N. Takayama, editors, *ICMS*, volume 4151 of *Lecture Notes in Computer Science*, pages 343–353. Springer, 2006.

[Hal05] T.C. Hales. Introduction to the flyspeck project. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[Har96a] J. Harrison. HOL light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 265–269. Springer-Verlag, 1996.

[Har96b] J.R. Harrison. Proof Style. In E. Giménez and C. Paulin-Möhring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[Har98] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[Har06] J. Harrison. Towards self-verification of hol light. In Furbach and Shankar [FS06], pages 177–191.

[Hol] HOL 4. `http://hol.sourceforge.net/`.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[HT98] J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[Inl] Inleiding Logica. `http://www.cs.vu.nl/~tcs/il/`.

[Jac95] P.B. Jackson. Enhancing the nuprl proof development system and applying it to computational abstract algebra. Technical report, Cornell University, Ithaca, NY, USA, 1995.

[JN04] D.J. Jeffrey and A.C. Norman. Not seeing the roots for the branches: multivalued functions in computer algebra. *SIGSAM Bull.*, 38(3):57–66, 2004.

[Kal] C. Kaliszyk. Prototype computer algebra system in hol light. `http://www.cs.ru.nl/~cek/holcas/`.

[Kal07] C. Kaliszyk. Web interfaces for proof assistants. In S. Autexier and C. Benzmüller, editors, *Proceedings of the FLoC Workshop on User Interfaces for Theorem Provers (UITP'06), Seattle*, volume 174[2] of *Electr. Notes Theor. Comput. Sci.*, pages 49–61, 2007.

[Kal08] C. Kaliszyk. Automating side conditions in formalized partial functions. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *AISC/MKM/Calculemus*, volume 5144 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2008.

[KKMW07] M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors. *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, June 27-30, 2007, Proceedings*, volume 4573 of *Lecture Notes in Computer Science*. Springer, 2007.

[KM96] M. Kaufmann and J.S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[KO08] C. Kaliszyk and R. O'Connor. Computing with classical real numbers. *Journal of Automated Reasoning*, 2008. Submitted.

[Koh00] M. Kohlhase. Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In J.A. Campbell and E. Roanes-Lozano, editors, *AISC*, volume 1930 of *Lecture Notes in Computer Science*, pages 32–52. Springer, 2000.

[Kra06] A. Krauss. Partial recursive functions in higher-order logic. In Furbach and Shankar [FS06], pages 589–603.

[KRW+08] C. Kaliszyk, F. van Raamsdonk, F. Wiedijk, H. Wupper, M. Hendriks, and R. de Vrijer. Deduction using the ProofWeb system. Technical Report ICIS–R08016, Radboud University Nijmegen, September 2008.

[KW07] C. Kaliszyk and F. Wiedijk. Certified computer algebra on top of an interactive theorem prover. In Kauers et al. [KKMW07], pages 94–105.

[KW08] C. Kaliszyk and F. Wiedijk. Merging procedural and declarative proof. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *TYPES*, volume 5497 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 2008.

[Ler06] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54. ACM Press, 2006.

[Les01] D.R. Lester. Effective continued fractions. In *Proceedings 15th IEEE Symposium on Computer Arithmetic*, pages 163–170. IEEE Computer Society Press, June 2001.

[Les08] D. R. Lester. Real number calculations and theorem: Proving validation and use of an exact arithmetic. In O. Ait-Mohamed, editor, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2008.

[Let02] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.

[LHLHW+04] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification, Version 1. *W3C Recommendation*, 2004.

[Loz03] D.W. Lozier. Nist digital library of mathematical functions. *Ann. Math. Artif. Intell.*, 38(1-3):105–119, 2003.

[LV] Logical Verification. `http://www.cs.vu.nl/~tcs/lv/`.

[Map95] Maple form interface, 1995.
`http://www.cecm.sfu.ca/organics/help/nmpform.html`.

[MBG+03] E. Melis, J. Büdenbender, G. Goguadze, P. Libbrecht, and C. Ullrich. Knowledge representation and management in activemath. *Ann. Math. Artif. Intell.*, 38(1-3):47–64, 2003.

[McB04] C. McBride. Epigram: Practical programming with dependent types. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[Mel08] G. Melquiond. Proving bounds on real-valued functions with computations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, Lectures Notes in Computer Science, Sydney, Australia, 2008.

[MHW08] F. van Raamsdonk M. Hendriks, C. Kaliszyk and F. Wiedijk. Teaching logic using a state-of-the-art proof assistant. In *Proceedings of the ETAPS Workshop on Formal Methods in Education (ForMEd'08), Budapest*, 2008. Accepted for publication in ENTCS.

[Miz] Mizar. `http://www.mizar.org/`.

[Miz08] Mizar Development Team. Mizar wiki, 2008. `http://wiki.mizar.org/`.

[MS97] O. Müller and K. Slind. Treating partiality in a logic of total functions. *Comput. J.*, 40(10):640–652, 1997.

[Muz93] M. Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993.

[Niq04] M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud Universiteit Nijmegen, September 2004.

[Niq08] M. Niqui. Cocorico: a Coq wiki, 2008. `http://www.lix.polytechnique.fr/cocorico/`.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NuP] PRL Project: "Proof/Program Refinement Logic". `http://www.cs.cornell.edu/Info/Projects/NuPRL/`.

[Obu08] S. Obua. *Flyspeck II: The Basic Linear Programs*. PhD thesis, Technische Universitat Munchen, 2008. Submitted.

[O'C05] R. O'Connor. Essential incompleteness of arithmetic verified by coq. In J. Hurd and T.F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2005.

[O'C07] R. O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 17(1):129–159, 2007.

[Oca] Ocamlnet. `http://projects.camlcity.org/projects/ocamlnet.html`.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Berlin, Heidelberg, New York, 1992. Springer-Verlag.

[Pau83] L.C. Paulson. A higher-order implementation of rewriting. *Sci. Comput. Program.*, 3(2):119–149, 1983.

[Pau05] L. Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.

[Pot99] L. Pottier. LogiCoq, 1999. `http://wims.unice.fr/wims/wims.cgi?module=U3/logic/logicoq`.

[Pra95] V.R. Pratt. Anatomy of the pentium bug. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/-FASE on Theory and Practice of Software Development*, pages 97–107, London, UK, 1995. Springer-Verlag.

[PT98] E. Poll and S. Thompson. Adding the axioms to Axiom: Towards a system of automated reasoning in Aldor. In *Calculemus and Types '98*, July 1998. Also as technical report 6-98, Computing Laboratory, University of Kent.

[Pvs] PVS Specification and Verification System. http://pvs.csl.sri.com/.

[RT03] P. Rudnicki and A. Trybulec. On the integrity of a repository of formalized mathematics. In *MKM '03: Proceedings of the Second International Conference on Mathematical Knowledge Management*, pages 162–174, London, UK, 2003. Springer-Verlag.

[Sor00] V. Sorge. Non-trivial symbolic computations in proof planning. In *FroCoS '00: Proceedings of the Third International Workshop on Frontiers of Combining Systems*, pages 121–135, London, UK, 2000. Springer-Verlag.

[SPK+] C. Schurmann, F. Pfenning, M. Kohlhase, N. Shankar, and S. Owre. Logosphere. A Formal Digital Library. Logosphere homepage: http://www.logosphere.org/.

[Urb98] C. Urban. Implementation of proof search in the imperative programming language pizza. In H.C.M. de Swart, editor, *TABLEAUX*, volume 1397 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 1998.

[vD07] H. van Ditmarsch. Logic courseware, 2007. http://www.cs.otago.ac.nz/staffpriv/hans/logiccourseware.html.

[Vui88] J. Vuillemin. Exact real computer arithmetic with continued fractions. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 14–27, New York, NY, USA, 1988. ACM Press.

[W3C08] W3C. Web applications (webapps) working group, 2008. http://www.w3.org/2008/webapps/.

[Web08] Web APIs Working Group. The XMLHttpRequest Object. Technical report, W3C, 2008. http://www.w3.org/TR/XMLHttpRequest/.

[Wes99] M.J. Wester, editor. *Contents of Computer Algebra Systems: A Practical Guide*, chapter A Critique of the Mathematical Abilities of CA Systems. John Wiley & Sons, Chichester, United Kingdom, 1999.

[Wie01] F. Wiedijk. Mizar light for hol light. In *TPHOLs '01: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 378–394, London, UK, 2001. Springer-Verlag.

[Wie09] F. Wiedijk. Formalizing 100 theorems, 2009. `http://www.cs.ru.nl/~freek/100/`.

[Wol03] S. Wolfram. *The Mathematica Book*. Wolfram Media, Incorporated, 2003.

[WZ03] F. Wiedijk and J. Zwanenburg. First order logic with domain conditions. In D.A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2003.

Summary

Proof assistants are computer programs that help the user build mechanically checked proofs. Proof assistants allow describing mathematical concepts or creating models of computer systems and proving properties of those concepts and models. The use of proof assistants is currently limited to specialists in the domain, since to make a proof as done on paper accepted by a proof assistant, one needs to add many details.

In this thesis we present an approach to extending the usability of proof assistants in mathematics and computer science. We do it in two ways: by combining proof assistants with computer algebra systems and by providing interactive access to such systems over the web.

Computer algebra systems are computer programs that process mathematical expressions. They include general purpose mathematical utilities, not only computer algebra. Most computer algebra systems allow manipulation of symbolic expressions including automatic simplifications, performing substitutions, solving equations over various domains, calculating arbitrary precision numerical approximations and plotting graphs of functions. Most computer algebra systems are designed to be easy to use for a beginner. Computer algebra systems allow the users to enter mathematical expressions in traditional mathematical notation and output results in user friendly format. Defining structures or functions and computing with them is easy, so performing experiments inside such a system is simple. This simplicity comes with a drawback. The algorithms implemented in those systems are not formally checked and are therefore known to make mistakes.

The thesis consists of two parts. Part I deals with approaches that combine computer algebra systems with proof assistants.

In Chapter 1 we build a prototype computer algebra system built on top of a proof assistant, HOL Light. This architecture guarantees that the system will make no mistakes. All expressions in the system have precise semantics and the proof assistant will check the correctness of all simplifications according to this semantics. We designed the user interface to close to the interfaces of popular computer algebra systems. This allows the user to easily probe the underlying automation of the proof assistant.

In Chapter 2 we extend the above system with partiality. We present an approach to formalizing partiality in real and complex analysis in total frameworks that allows the side conditions to be kept hidden from the user as long as

they can be computed and simplified automatically. Assumptions about the domains of partial functions are necessary when we guarantee correctness in proof assistants. On the other hand when mathematicians write about partial functions they tend not to explicitly write these side conditions. The solution from this chapter simplifies defining and operating on partial functions in formalized real analysis in HOL Light. It allows simplifying expressions under partiality conditions in a proof assistant in a manner that resembles computer algebra systems.

Chapter 3 talks about real number approximations in proof assistants. In Coq it is possible to work with infinite precision real numbers effectively. We investigate how to use the classical theory of real numbers together with approximations computed constructively. We combine the two main Coq libraries that have a theory of the real numbers: the Coq standard library, which gives an axiomatic treatment of the classical theory of real numbers and the CoRN library from Nijmegen which defines a constructively valid theory of real numbers.

In Part II we look at interactive formalized mathematics on the web. We describe a web interface for proof assistants and we investigate the use of this interface in teaching logic and in collaborative proof development.

In Chapter 4 we describe an architecture for creating responsive web interfaces for proof assistants. We create an interface that is available completely within a web browser, but resembles and behaves like a local one. Security, availability and efficiency issues of the architecture are described.

Chapter 5 describes the system ProofWeb which uses a web interface to Coq to teach logic to undergraduate computer science students. The system makes the full power of Coq available to the students, but simultaneously presents the logic problems in a way that is customary in undergraduate logic courses. We describe a large database of logic problems and the tactics for Coq that have been developed for the inference rules of the logic.

In Chapter 6 we present an algorithm for converting tree style proofs to flag style proofs. We then present a rewrite system that simplifies the results. It has been implemented in ProofWeb. The algorithm can be used to convert arbitrary procedural proofs to declarative proofs. In ProofWeb a proof that is given as a Coq proof script (even with arbitrary Coq tactics) can be displayed both as a tree style and as a flag style proof.

In Chapter 7 we combine the web interface from previous chapters with a wiki to create an environment for the collaborative development of formal proofs. We describe a prototype based on Coq and a modified version of the MediaWiki code-base that allows modifying proofs while preserving repository consistency and rendering the provided proofs.

Samenvatting (Dutch summary)

Bewijsassistenten zijn computerprogramma's die de gebruiker helpen mechanisch geverifieerde bewijzen te construeren. Bewijsassistenten maken het mogelijk wiskundige concepten te beschrijven en modellen van computersystemen te bouwen, en eigenschappen van deze concepten en modellen te bewijzen. Het gebruik van bewijsassistenten is tegenwoordig nog beperkt tot specialisten in dit onderwerp, omdat om een bewijs zoals het op papier wordt gedaan door een bewijsassistent te laten accepteren, er vele details aan moeten worden toegevoegd.

In dit proefschrift presenteren we methoden om de bruikbaarheid van bewijsassistenten voor wiskunde en informatica te verhogen. We doen dit op twee manieren: door bewijsassistenten met systemen voor computer algebra te integreren, en door interactieve toegang tot zulke systemen via het web mogelijk te maken.

Computer algebra systemen zijn computerprogramma's die wiskundige expressies verwerken. Behalve computer algebra vallen hier ook algemenere wiskundige programma's onder. De meeste computer algebra systemen maken het mogelijk symbolische expressies te manipuleren, onder meer door automatisch vereenvoudigen, door substitueren, door het oplossen van vergelijkingen in verschillende domeinen, door het berekenen van numerieke benaderingen met een willekeurige precisie, en door het tekenen van grafieken van functies. De meeste systemen voor computer algebra zijn ontworpen om eenvoudig bruikbaar te zijn voor beginners. Computer algebra systemen stellen de gebruikers in staat wiskundige expressies in traditionele wiskundige notatie in te voeren en ze geven antwoorden in een gebruikersvriendelijk vorm. Met deze systemen is het makkelijk om structuren en functies te definiëren, en daarom is het eenvoudig om met een dergelijk systeem te experimenteren. Deze eenvoud heeft ook een nadeel. De algoritmes die in deze systemen zijn geïmplementeerd zijn niet formeel geverifieerd, en maken soms fouten.

Dit proefschrift bestaat uit twee delen. Deel I gaat over methoden om computer algebra met bewijsassistenten te combineren.

In Hoofdstuk 1 ontwikkelen we een prototype computer algebra systeem bovenop de bewijsassistent HOL Light. Deze architectuur garandeert dat het systeem geen fouten maakt. Alle expressies in het systeem hebben een welgedefinieerde semantiek, en de bewijsassistent checkt de correctheid van alle vereenvoudigingen met betrekking tot deze semantiek. We hebben ontworpen

een gebruikersinterface dat sterk lijkt op de interfaces van populaire systemen voor computer algebra. Dit maakt het eenvoudig voor een gebruiker om met de onderliggende automatisering van de bewijsassistent te experimenteren.

In Hoofdstuk 2 breiden we het bovenstaande systeem uit met partialiteit. We presenteren een aanpak voor formalisatie van partialiteit in reële en complexe analyse in een totaal framework, waarbij het mogelijk is de zij-condities voor de gebruiker verborgen te houden zo lang ze automatisch berekend en vereenvoudigd kunnen worden. Aannames over het domein van partiële functies zijn noodzakelijk wanneer we correctheid in een bewijsassistent willen garanderen. Aan de andere kant schrijven wiskundigen als ze partiële functies gebruiken deze zij-condities meestal niet expliciet op. De oplossing van dit hoofdstuk maakt het eenvoudiger om partiële functies te definiëren en er mee te werken in geformaliseerde reële analyse in HOL Light. Het maakt het mogelijk expressies met partialiteitscondities in een bewijsassistent te vereenvoudigen op een manier die lijkt op die van computer algebra.

Hoofstuk 3 gaat over het benaderen van reële getallen in bewijsassistenten. In Coq is het mogelijk om effectief met reële getallen te werken die een willekeurig grote precisie hebben. We onderzoeken hoe de theorie van de reële getallen uit de klassieke wiskunde te gebruiken is met benaderingen die constructief berekend zijn. We combineren de twee belangrijkste Coq bibliotheken die een theorie van de reële getallen bevatten: de Coq standaardbibliotheek die een axiomatische behandeling van de klassieke theorie van de reële getallen geeft, en de CoRN bibliotheek uit Nijmegen die een constructieve theorie van de reële getallen definieert.

In Deel II kijken we naar interactieve formele wiskunde op het web. We beschrijven een web-interface voor bewijsassistenten en onderzoeken het gebruik van dit interface in het onderwijs in de logica en in collaboratieve bewijsontwikkeling.

In Hoofdstuk 4 beschrijven we een architectuur voor het bouwen van responsieve web-interfaces voor bewijsassistenten. We bouwen een interface dat volledig in een web-browser beschikbaar is, maar zich gedraagt als een locaal interface. We beschrijven de veiligheid, beschikbaarheid en efficiëntie van de architectuur.

Hoofdstuk 5 beschrijft het ProofWeb systeem dat een web-interface voor Coq gebruikt om logica te onderwijzen aan studenten informatica. Het systeem maakt de volledige kracht van Coq beschikbaar voor de studenten, maar tegelijkertijd presenteert het logica-opgaven op een manier die gebruikelijk is in propedeuse logica-cursussen. We beschrijven een grote verzameling logica-opgaven, en de tactieken voor Coq die ontwikkeld zijn voor de redeneerregels van de logica.

In Hoofdstuk 6 presenteren we een algoritme om boombewijzen om te zetten in vlaggenbewijzen. Vervolgens geven we een herschrijfsysteem om het resultaat te vereenvoudigen. Dit is geïmplementeerd in ProofWeb. Het algoritme kan gebruikt worden om willekeurige procedurele bewijzen om te zetten in declaratieve bewijzen. In ProofWeb kan een bewijs dat als een Coq bewijs is gegeven (zelfs wanneer willekeurige Coq tactieken zijn gebruikt) worden weergegeven zowel als

boom- als als vlaggenbewijs.

In Hoofdstuk 7 combineren we het web-interface uit de vorige hoofdstukken met een wiki, en creëren zo een omgeving voor de collaboratieve ontwikkeling van formele bewijzen. We beschrijven een prototype gebaseerd op Coq en een aangepaste versie van de MediaWiki broncode, die het mogelijk maakt bewijzen te bewerken terwijl de consistentie van de repository en de weergave van de bewijzen behouden blijft.

Streszczenie (Polish summary)

Systemy dowodzenia twierdzeń są programami komputerowymi, które pomagają użytkownikom tworzyć mechanicznie sprawdzone dowody. Te systemy pozwalają zdefiniować pojęcia matematyczne lub modele oprogramowania, a następnie dowodzić własności tych pojęć i modeli. Systemy dowodzenia twierdzeń są współcześnie używane jedynie przez specjalistów, gdyż do pełnego zaakceptowania dowodu przez system wymagane jest dodanie wielu szczegółów.

W niniejszej pracy doktorskiej przedstawione są sposoby zwiększenia używalności systemów dowodzenia twierdzeń w matematyce i informatyce. Przedstawione są dwie metody: łączenie systemów dowodzenia twierdzeń z systemami algebry komputerowej oraz umożliwienie dostępu do tych systemów przez sieć.

Systemy algebry komputerowej są programami, które służą do przekształcania wyrażeń matematycznych. Systemy te potrafią wykonywać wiele ogólnych matematycznych operacji, nie ograniczają się jedynie do algebry komputerowej. Większość systemów algebry komputerowej pozwala na operowanie wyrażeniami symbolicznymi wraz z automatycznym ich upraszczaniem, wykonywanie podstawień, rozwiązywanie równań w różnych dziedzinach, obliczanie przybliżeń numerycznych dowolnej dokładności oraz rysowanie wykresów funkcji. Większość systemów algebry komputerowej jest zaprojektowana z myślą o początkującym użytkowniku. Pozwalają one na wprowadzanie wyrażeń matematycznych w standardowej matematycznej notacji i zwracają wynik w formacie przyjaznym użytkownikowi. Tworzenie nowych definicji struktur i funkcji jest uproszczone, przez co eksperymentowanie z takim systemem jest względnie łatwe. Takie uproszczenie ma jednak wady. Algorytmy zaimplementowane w tych systemach nie są formalnie sprawdzone, przez co systemy te mogą popełniać błędy.

Przedstawiona praca doktorska składa się z dwóch części. Część pierwsza przedstawia metody łączenia systemów dowodzenia twierdzeń z systemami algebry komputerowej.

W Rozdziale 1 przedstawiamy prototypowy system algebry komputerowej zbudowany wewnątrz systemu dowodzenia twierdzeń HOL Light. Ta architektura zapewnia, że system nie będzie popełniał żadnych błędów. Wszystkie wyrażenia w systemie mają dokładną semantykę, a system dowodzenia twierdzeń sprawdza poprawność wszystkich przekształceń zgodnie z tą semantyką. Zaprojektowaliśmy interfejs użytkownika systemu tak, aby zachowywał się podobnie do popularnych systemów algebry komputerowej. To pozwala użytkownikom na

proste eksperymentowanie z automatyką bazowego systemu dowodzenia twierdzeń.

W Rozdziale 2 rozszerzamy powyższy system o obsługę funkcji częściowych. Przedstawiamy sposób formalizacji funkcji częściowych z analizy rzeczywistej i zespolonej w środowiskach całkowitych, który pozwala na ukrycie założeń częściowości, o ile mogą one być znalezione i uproszczone automatycznie. Założenia na temat dziedzin funkcji częściowych są niezbędne, aby zagwarantować poprawność w systemie dowodzenia twierdzeń. Jednocześnie matematycy zazwyczaj nie piszą *explicite* tych założeń. Rozwiązanie przedstawione w tym rozdziale upraszcza definiowanie i operowanie na funkcjach częściowych w formalnej analizie w HOL Light. To pozwala na upraszczanie wyrażeń pod założeniami częściowości w systemie dowodzenia twierdzeń w sposób, który przypomina ten z algebry komputerowej.

Rozdział 3 dotyczy przybliżeń numerycznych liczb rzeczywistych w systemach dowodzenia twierdzeń. W systemie Coq można pracować efektywnie z liczbami rzeczywistymi o nieskończonej dokładności. Przedstawiamy możliwość użycia aksjomatycznie zdefiniowanych klasycznych liczb rzeczywistych z biblioteki standardowej Coq'a z liczbami rzeczywistymi z biblioteki CoRN, których teoria jest poprawna konstruktywnie.

W części drugiej zajmujemy się interaktywną sformalizowaną matematyką w sieci. Opisujemy interfejs strony internetowej dla systemów dowodzenia twierdzeń oraz badamy używalność tego interfejsu do uczenia logiki oraz do współpracy przy tworzeniu dowodów.

W Rozdziale 4 opisujemy architekturę interaktywnych interfejsów strony internetowej dla systemów dowodzenia twierdzeń. Tworzymy interfejs, który jest dostępny w całości poprzez przeglądarkę, choć przypomina i zachowuje się tak jak lokalny interfejs użytkownika. Opisane są także bezpieczeństwo, dostępność i efektywność przedstawionego rozwiązania.

Rozdział 5 opisuje system ProofWeb, który używa interfejsu strony internetowej do Coq'a do uczenia logiki studentów informatyki. Ten system pozwala studentom używać całej mocy Coq'a jednocześnie przedstawiając problemy logiczne w sposób zwyczajowy dla studiów logiki. Przedstawiamy bazę problemów logicznych oraz zaimplementowane taktyki dla reguł inferencji logiki.

W Rozdziale 6 przedstawiamy algorytm przekształcania drzew dowodu w dowody flagowe. Przedstawiamy także system przepisywania, który upraszcza uzyskane dowody. Takie przekształcenie zostało zaimplementowane w systemie ProofWeb. Nasz algorytm może przekształcić każdy dowód proceduralny w dowód deklaratywny. W systemie ProofWeb dowód podany jako skrypt dowodowy Coq'a (z dowolnymi taktykami) może być wyświetlony zarówno jako drzewo dowodu jak i jako dowód flagowy.

W Rozdziale 7 przedstawione jest połączenie interfejsu strony internetowej z poprzednich rozdziałów z wiki dla stworzenia pełnego środowiska do współpracy przy tworzeniu dowodów matematycznych. Opisujemy prototyp oparty na Coq'u oraz Mediawiki, który pozwala rozwijać i modyfikować dowody zachowując spójność repozytorium.

Curriculum Vitae

Born on 04.08.1981 in Warsaw, Poland.

1996-2000 High School. Warsaw, Poland.

2000-2005 M.Sc. Computer Science. Warsaw University, Poland.

2000-2004 B.Sc. Mathematics. Warsaw University, Poland.

2005-2009 Junior researcher. Radboud University Nijmegen, the Netherlands.