Generation and Presentation of Formal Mathematical Documents

Martijn Oostdijk
Oostdijk, Martijn D.


Subject headings : automated theorem proving / formal logic / lambda-calculus / computer algebra

CR Subject Classification (1998) : F.4.1, I.7.2, I.1

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
Generation and Presentation of Formal Mathematical Documents

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. R.A. van Santen, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 18 september 2001 om 16.00 uur

door

Martijn Diederik Oostdijk

geboren te Tilburg
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J. C. M. Baeten
en
prof.dr. H. P. Barendregt

Copromotor:

dr. J. H. Geuvers
Acknowledgments

I am indebted to my promotors, my co-promotor and the other members of the manuscript committee. As leader of the formal methods group in Eindhoven, Jos Baeten has provided a very pleasant environment for doing my research. I experienced Henk Barendregt’s continued interest as inspiration for further investigations. Herman Geuvers has been an excellent supervisor. Initially, in Eindhoven as the theoretician whose door was always open. Later on, after he started working in Nijmegen, as a travel companion. Many things about doing scientific research I picked up from discussions and from working together with Herman. Arjeh Cohen and Michael Kohlhase were the other two members of the manuscript committee. I am grateful for their careful reading of the manuscript and for their comments which lead to many improvements.

My colleagues and especially my roommates have been very important to me. For a short while, I shared my office with Twan Laan and Roel Bloo. Observing them taught me a lot about what it must be like to be a fourth year PhD student. After they left, I shared my office with Suzana Andova and Tim Willemse. I thank them for their support and always happy moods. The input I got from my colleagues who work or worked on type theory, has been most helpful. I thank Georgi Jojgov, Jan Zwanenburg, Kees Hemerik, Michael Franssen (not a TafkaTaM member), and Rob Nederpelt. I thank my other colleagues from the formal methods group, as well as the usual ‘meeëters’: Andre, Bart, Dragan, Elize, Francien, Ilke, Jan Joris, Jeroen, Marcella, Michel, Peter, Roel, Rogier, Ruurd, Sjouke, Thijs, Twan, and Victor. I thank the members of the Foundations group in Nijmegen for their hospitality. I worked in this group (almost) every Friday, after Herman started working in Nijmegen. I also thank my new colleagues of the ITT group in Nijmegen for their support and encouragement while I finished the thesis. I thank the members of the RIACA group in Eindhoven, where I spent some time working together with Olga Caprotti. Chapter 5, my favorite chapter in this thesis, is a direct result of this cooperation. I thank Olga for her enthusiasm and hard work. My research position in Eindhoven was funded by the research school IPA. I thank IPA, and especially Anne-Meta Oversteegen and Tijn Borghuis, for the many enjoyable activities they organized. I am especially indebted to Tijn Borghuis for commenting on some very early versions of the manuscript and for the many discussions on subjects in this thesis. In the past four years I have had the pleasure to work together with many students. Some of the projects these students worked on have helped me a lot, especially in the implementation work described in Chapter 4. I thank Ferry de Jong, Bryan Bakker, Harm van Beek, Robert
Brouwer, and Roel Körvers for their contributions.

Finally, I thank my mother and my brother, Diederik Oostdijk, for their love and support. Diederik also read the introduction and conclusions and improved the overall quality of the English.
# Contents

1 Introduction .............................. 1
   1.1 Introduction ................................... 1
   1.2 Formal Mathematics ............................... 4
   1.3 Computer Mathematics ............................. 6
      1.3.1 Theorem Provers ................................ 6
      1.3.2 Computer Algebra ............................... 8
      1.3.3 Presentation of Mathematics ..................... 10
   1.4 Conclusions ................................... 11

2 Type Theory for Theorem Provers ............... 15
   2.1 Introduction ................................... 15
   2.2 Type Theory ................................... 15
   2.3 Theorem Proving based on Type Theory ............... 17
      2.3.1 Curry-Howard-De Bruijn isomorphism .............. 17
      2.3.2 Contexts ................................... 18
   2.4 Calculus of Constructions ........................... 19
      2.4.1 Pseudo Expressions ................................ 19
      2.4.2 Conversion ................................... 20
      2.4.3 Type Assignment ................................ 22
      2.4.4 Representing Mathematics ......................... 23
   2.5 Calculus of Inductive Constructions ................... 27
      2.5.1 Pseudo Expressions ................................ 28
      2.5.2 Conversion ................................... 30
      2.5.3 Type Assignment ................................ 31
      2.5.4 Representing Mathematics ......................... 35
   2.6 Theorem Proving in Coq ............................. 38
      2.6.1 Coq Syntax ................................... 38
      2.6.2 Tactics ................................... 40
   2.7 Conclusions ................................... 42

3 The Reflection Principle ......................... 43
   3.1 Introduction ................................... 43
   3.2 Computations in Proofs ............................ 44
3.2.1 External Computations ........................................ 44
3.2.2 Internal Computations ........................................ 45
3.2.3 Oracle Computations ......................................... 45
3.3 Reflection ......................................................... 46
3.4 Propositional Calculus .......................................... 47
  3.4.1 Interpretations ............................................... 49
  3.4.2 Proof-objects ............................................... 51
3.5 Primitive Recursive Arithmetic ................................ 52
  3.5.1 Interpretations ............................................... 53
  3.5.2 Proof-objects ............................................... 56
3.6 Internal versus External Computations ....................... 57
3.7 Results .......................................................... 58
3.8 Conclusions ...................................................... 61

4 Interactive Mathematical Documents .............................. 63
  4.1 Introduction .................................................... 63
  4.2 Overview of the System ........................................ 67
    4.2.1 Formalization Phase ...................................... 68
    4.2.2 Authoring Phase .......................................... 69
    4.2.3 Presentation Phase ....................................... 70
  4.3 Terms .......................................................... 71
    4.3.1 Context and Context Item Nodes ......................... 72
    4.3.2 Basic Nodes ............................................... 72
    4.3.3 Abstraction Nodes ........................................ 72
    4.3.4 Inductive Definition Nodes ............................. 74
    4.3.5 Cases and Match Nodes .................................. 74
    4.3.6 Fixpoint Nodes .......................................... 75
  4.4 Operations ...................................................... 77
    4.4.1 Copy ...................................................... 77
    4.4.2 Syntactical Equivalence .................................. 78
    4.4.3 Currying .................................................. 78
    4.4.4 Substitution .............................................. 79
    4.4.5 Conversion ............................................... 79
    4.4.6 Matching ................................................ 80
    4.4.7 Typing .................................................... 81
    4.4.8 Expected Type ............................................ 82
  4.5 Generation of Documents ...................................... 83
    4.5.1 Mathematical Meta Texts ................................ 85
    4.5.2 Verbalization of Proof-objects .......................... 87
  4.6 Presentation of Documents ................................... 89
    4.6.1 Tree View ................................................ 90
    4.6.2 Natural Language View ................................... 90
  4.7 XML, OpenMath and OMDoc ................................... 90
Chapter 1

Introduction

1.1 Introduction

Computer Mathematics, the subject of this thesis, is a relatively new field of scientific interest. It explores the role that computers and computer science can play in the development of mathematics. In this chapter we look at a few of the promises and challenges of this new field. The remainder of this thesis is devoted to understanding some of the fundamental problems in Computer Mathematics. We attempt to determine the current state of affairs in this new field through some concrete case studies and the implementation of prototype tools.

The ultimate goal of Computer Mathematics is the ideal mathematical workspace. This is a computer environment which lets mathematicians do all of their mathematical work, which includes defining new concepts, performing symbolic computations, logical reasoning, and publishing, on a computer in a way that is as easy as doing mathematics on paper but with all of the benefits that computers bring. These benefits include:

- Checking of correctness of statements.
- Automatic symbolic computations.
- Easy sharing of work with others through electronic publishing on information networks.

Some of the problems facing Computer Mathematics will only become clear after we explain the current state of the art in this new field.

Consider our view on the current state of Computer Mathematics given in Figure 1.1. The diagram shows different versions of the mathematical reality. Near the top of the figure is an idealized mathematical reality denoted as Plato’s heaven. This thesis does not aim to answer the question of what this ideal mathematical reality is. Nor do we take a philosophical standpoint as to what this mathematical reality looks like. One can, for example, think of Plato’s heaven as the collection of mathematical notions in a mathematician’s head. We merely note that it is different from the reality called Formal
Mathematics, in the diagram directly below Plato’s heaven, which is a syntactical derivative of the real mathematical world. We come back to this level of formal mathematics below. It is important to note that formality of the mathematics is a requirement in order for computer programs to operate on this level. To the left and right of the formalized mathematics world are two Computer Mathematics systems: A computer algebra system (CAS) to the left and a (logical) theorem prover (TP), or proof assistant, to the right. These systems operate on objects in the world of formalized mathematics, although the TP usually also helps a user formalize, i.e. make the transition from Plato’s heaven to formalized mathematics. The core of the TP, however, consists of a proof checker which operates on a statement and a proof, both of which are formal mathematical objects. Another Computer Mathematics system takes care of presenting objects in the formalized mathematics world to the user. This results in so-called Interactive Mathematical Documents, near the bottom of Figure 1.1.

The individual computer mathematics systems in Figure 1.1 exist already. However, they do not yet form one workspace. Like many computer science problems, the realization of the ideal mathematical workspace boils down to finding an appropriate language $\mathcal{L}$. Such a language connects the systems at one or more levels.

![Figure 1.1: Plato’s heaven and formal mathematics.](image)

Judging from Figure 1.1, it seems that this language is already present in the form of the formalized mathematical world. However, the diagram is a bit misleading as it presents the formal mathematical world as one single component. In reality the formal
1.1. INTRODUCTION

The mathematical world consists of many different incompatible formalisms. Each computer mathematics system has its own object language, and although many of these formalisms allow to describe very similar mathematical concepts, in practice it is very hard to share mathematical content between systems on the formal mathematical level. All present day Computer Mathematics systems use some form of formal mathematics, but there is not one fixed formal system to which they all adhere.

What is this language $\mathcal{L}$? The language $\mathcal{L}$ should be universal, because we want to be able to express all of mathematics in $\mathcal{L}$. From the fact that there are so many different implementations of formal mathematics, one could conclude that a universal language $\mathcal{L}$ which supersedes all these different formalisms, does not exist. Moreover, such a language would not be very practical as [56] successfully argues.

An alternative to combining the systems on the formal mathematical level, is to provide a common user interface. It seems that the notion of document is a good interface for our target audience consisting of mathematicians. In user interface design it is considered good practice to present the user with an interface that closely resembles, and gives direct access to, the model under inspection. Although the real model is Plato’s heaven, the document model is an interface most mathematicians are familiar with. But also for technological reasons, a document style interface to Computer Mathematics systems is preferable, see for example [84].

If the systems are to be combined on the user interface level, all that is needed is a communication language which makes superficial interchange of mathematical objects possible. Such a solution does not ground the languages of the different systems in one and the same formalism, it merely provides a means of transport for expressions from one system to the other systems. Note that such a transport mechanism for exchanging expressions is also required if we combine systems on the formal mathematical level.

In this thesis both ways of combining systems are considered. Although we believe that combining systems on a formal mathematics level is necessary if logical TP systems are to be included in the ideal mathematical workspace. Also it is argued in this thesis that presentation of mathematics in the form of interactive mathematical documents is made easier if the content is represented in a language with formal mathematical structure. All traditional uses of computers in mathematics can be brought together in one system if we base ourselves on formal mathematics.

As the title of the thesis suggests, we concentrate on two things: Generation (formalization) and presentation of formal mathematics (interactive mathematical documents). Other topics, closely related to these, are computing, proving and combining of mathematical systems. Note that, while formal mathematics plays an important role, the formal mathematics itself is not the subject of this thesis. The formalization process which translates informal mathematical theories to formal mathematical documents remains of great interest to Computer Mathematics.

One could argue that all computer programs solve some mathematical problem. Yet, most programs do not have pure mathematics as application domain, they merely use mathematics as a modeling language for problems from different domains. In this thesis, we therefore focus on computer support for pure mathematics, i.e. the mathematics
practiced by mathematicians. But of course such systems can also be used to solve real world problems as long as these problems can be translated to mathematics.

1.2 Formal Mathematics

Before answering the question “What is formal mathematics?” it is necessary to provide some historical background. Note that this is not intended to be a complete overview of the history of formal mathematics. A more extensive overview of the development and history of logic and the role of type theory within this development is given in [58] and [62] respectively.

The history of formal mathematics and logic starts with Aristotle’s syllogisms. He was the first to recognize that reasoning takes place according to strict rules. The notion of mathematical proof, a tool to find mathematical truth by looking at the structure of statements, is based on these ideas. Finding a proof is difficult, but checking the validity of a proof is simply checking whether it adheres to the rules. Interest in understanding the nature of reasoning is what inspired Aristotle.

The science of mathematics developed in the following centuries with the concept of mathematical proof as a central notion. Although the notion of mathematical proof was not formal in our sense of the word formal, mathematical proofs were rigorous and helpful in demonstrating mathematical truths. Interest in formal mathematics or logic faded until the 17th century. Gottfried Wilhelm Leibniz (1646–1716) for one was looking for an artificial universal language in which problems could be stated and for which a mechanical calculation method existed to decide the problems. Essentially, if the problem domain is restricted to mathematics, he was looking for the language $\mathcal{L}$ in Figure 1.2 and a decision procedure for statements expressed in this language.

Gottlob Frege (1848–1925) was the first to formalize parts of mathematics in a logical system. He built arithmetic on top of a logical language, in other words, he implemented mathematical notions as definitions in the language itself. Not only the reasoning is done in logic (while the mathematical content of the statements is treated
In the beginning of the 20th century interest in formal mathematics was revived due to the discovery of some negative findings such as Russell’s paradox, Gödel’s incompleteness proof, Hilbert’s problems, and Brouwer’s intuitionism. An important aspect of Brouwer’s intuitionism is that it proposes to only consider mathematical principles that are constructive, which in a radical way gets rid of many inaccuracies.

After practical computers were introduced in the 1950’s, Leibniz’ dream of a universal language for stating problems, at least when we restrict ourselves to mathematics, might become a reality in the form of theorem provers. De Bruijn’s Automath [69] system is one of the first such theorem provers, based on typed λ-calculus. Note that the introduction of automatic computers radically changes the reasons why people study formal mathematics. The possibility to mechanically check the correctness of mathematical statements makes the field of formal mathematics interesting for mathematicians and computer scientists who are not in the first place interested in the logical foundations of mathematics.

Formality is a relative notion. To solve a real world problem, one builds a mathematical model which is an abstraction of the real world problem. Reasoning about such a model forces one to consider the problem in abstract terms, leaving out irrelevant details which may obscure a solution. Since such a model is formulated in the mathematical domain, we can apply mathematical methods to it, invented by generations of clever mathematicians. While building such a model formalizes the problem, it is often not formal in the strict syntactical sense used in this thesis.

Formalizing does not imply making a mathematical model of a real world problem, but encoding of mathematical content in a formal language $L$. Such a formal encoding allows manipulation of the mathematical content by computers, and is therefore a necessary condition if we want to do Computer Mathematics. The activity of formalizing informal mathematics bears some similarities to the activity of implementing software systems. in fact, it is similar to going from an informal specification of a computer program to the concrete program code. Because mathematics has been around much longer, and also because of the formal nature of mathematics, the specifications tend to be more clear than the ones in software industry. However, the 2000 years of existence of mathematics are not directed towards full formalization.

The most acute problem seems to be that formalizing means making implementation choices. Concepts in informal mathematics are reduced to the primitives of the formal language $L$. Given a concept in informal mathematics, there are often many different ways to implement it. Such a reduction is a one-way mapping therefore something is lost when we express abstract theories in $L$. Ideally, mathematics is representation independent, but in practice such fundamental implementation choices do matter.

From a verification point of view this is not really a problem. After all, mathematics is conducted on a high level, and only in the end when we want to verify the mathematics, is it formalized and reduced to $L$ to be checked. The formalization is done in such a way that correctness of the formal mathematics implies validity of the informal mathematics. However, in practice the activity of informal theory development and
formalization are not separated, and the implementation choices made during formal-
ization influence the development of the mathematical theory.

1.3 Computer Mathematics

Formal mathematics is pursued for various reasons. As explained in Section 1.2, it was
studied first out of foundational interest. When practical mechanical computing arrived
in the 1950’s, it became possible to do symbolic computation. Verification of mathemat-
ical statements using computers and typesetting of mathematical documents are also
seen as computer mathematics applications.

This leads to the three main directions in Computer Mathematics: Document prepa-
ration systems, Computer Algebra systems, and Theorem Provers. Depending on our view
of the mathematical world and the tasks that we need to perform, we pick one of these
systems:

- Firstly, the mathematical world can be viewed in a purely symbolic way. Math-
ematical objects exist and there is a notion of truth, but these are not formal. We
communicate mathematics with other mathematicians by creating documents full
of symbols.

- Secondly, the mathematical world can be viewed as consisting of objects and oper-
ations (computations) on those objects. Statements and proofs about the objects or
operations exist on a meta level and are themselves not part of the mathematical
world. This is the view taken by the CA builders.

- Thirdly, the mathematical world can be viewed as a world of statements, proofs,
and truth. Proofs are first class citizens and the statements contain symbols that
correspond to objects and operations which themselves are not part of the lan-
guage. This is the logical view of the world.

What we are ultimately looking for is a language in which we can unite these diver-
gent views on the mathematical world.

1.3.1 Theorem Provers

Statements make up mathematical reality in the TP trend, rather than computations. In
this thesis, when we talk about theorem provers, we mean interactive proof assistants
aimed at formalizing parts of mathematics. Examples of the systems that are interesting
for the sake of this thesis’ argument are Coq [11], Lego [66], HOL [44], Mizar [82], and
PVS [76]. See [10] for a comparison of these systems. Since the logic on which fully
automatic theorem provers are based is usually restricted to a decidable fragment, we
do not consider such systems based on model checking, tableaux methods, resolution,
or other proof search methods. The kind of TP systems we are interested in take proofs
seriously, a proof is something expressed in a formal system.
The goal of TP is to gain more certainty about the correctness of parts of mathematics. As we all know, computers are more precise than humans, not easily distracted, never get tired, have no ambitions, etc. Mathematicians simply do not have time to check all proofs produced by their colleagues due to the complexity of proofs and theories. The acceptance of a mathematical statement as being true has an important social aspect. Wiles’ proof of Fermat’s last theorem, for instance, is understood by very few people. Yet, most mathematicians now believe that Fermat’s last theorem is true. Is the proof really correct? A formalized version of this proof could be checked by a computer program and believing the truth of a mathematical statement, even if the proof is hard to grasp, would be possible.

The same principle applies to proofs about computer software and hardware. The difference, however, is that these computer science proofs tend to be much more elaborate, although the individual steps are easier. In other words, these proofs are less dependent on smart ideas or tricks, it is the size makes them hard.

Such a desirable universal checker for mathematics would only be possible if the whole of mathematics can be expressed in $\mathcal{L}$. For suitable $\mathcal{L}$, a proof checking algorithm can be built. The type theory described in Chapter 2 is such a checkable language. There is an important difference between checking proofs and finding proofs. Given a theorem and a proof, it can be decided whether the proof proves the theorem. In general, however, it is not possible to construct a computer program that, given a theorem, produces a proof.

Given a proof checker, the question arises whether the checker can be trusted. We may not always be convinced that all the powerful tactics that a TP provides are sound. Moreover, a TP occasionally contains a bug. See [79] for a discussion. Reasoning about the correctness of computer programs can be seen as a branch of mathematics, as suggested in Figure 1.2. As such, we can construct a correctness proof of the checker and we can apply the checker to this proof. (Provided we can find the proof.) If the answer is negative we know for sure the checker is not correct, but if the answer is positive both the checker and the proof may be incorrect.

The problem of the proof checker which is not to be trusted when checking its own correctness proof is not solvable. However, a partial solution is to simply make the checker very small by making $\mathcal{L}$ as small as possible, i.e. $\mathcal{L}$ should have very few primitives. Of course complicated things should still be expressible, i.e. $\mathcal{L}$ should be simple but powerful. In this way, the complexity is put into the mathematics expressed in $\mathcal{L}$, not the program that checks $\mathcal{L}$. The resulting checker can be proved correct manually, since it is small.

In type theoretic TPs, this issue of reliability is solved to some extent, because the TP does not only tell the user that the theorem has been proved, but it also provides a proof-object that can be type checked by the user (using his own – relatively easy to write – type checking algorithm. This is what makes Type Theory a suitable candidate for $\mathcal{L}$. Details on Type Theory are provided in Chapter 2. Type Theory reduces the number of primitives by using strong ones, such as $\lambda$ and $\Pi$-abstraction, which represent many different notions. Furthermore, the Curry-Howard-De Bruijn isomorphism makes statements
CHAPTER 1. INTRODUCTION

and proofs first class citizens. The feature of having proof-objects that can be checked independently by a relatively small and easy algorithm, is also known as the De Bruijn criterion [8], named after the founding father of the Automath project. In this project the first TPs based on type theory were implemented (in fact they were proof checkers rather than proof assistants).

So, on the one hand the De Bruijn criterion gives a higher degree of reliability to TPs. On the other hand, the criterion makes it harder to implement very powerful proof tactics (like resolution), because the system will always have to construct a complete proof term that can be (type) checked easily in a small underlying system. The De Bruijn criterion also makes it harder to implement efficient computations since data structures have to be implemented through many layers of encodings on top of other encodings.

**User interface.** From a user interface viewpoint the class of interactive theorem provers can be divided into two styles: batch (compiler like) and interactive (interpreter like). In a batch style theorem prover situation, the user prepares the input to the theorem prover offline. The input is fed into the TP only after the complete input is finished. The user interface of an interactive theorem prover resembles a Socratic dialog. The system shows the current goal to be proved and invites the user to convince the TP of its validity. In the concrete examples in this thesis, a theorem prover of the latter category is used, but this is not essential.

On the surface an interactive theorem prover is nothing more than a direct feedback command line system, just like the user interface of a CAS. The user enters a command (containing some mathematical expressions) and the system yields an answer (usually consisting of a mathematical expression). However, the context, the “mathematical state” of the system if you will, plays a much more important role in TP than in CA. Another notable difference is that in a TP there is no pretty printing based on mathematical conventions, except on the lowest logical level. The user implements mathematical notions on top of the logical system and it is therefore the user’s responsibility to provide a solid presentation of his or her mathematical theory. Some TP systems, for that reason, provide a mechanism to implement user-specified pretty printing.

### 1.3.2 Computer Algebra

General purpose Computer Algebra Systems (CAS) have existed since the 1960’s. See [35] for an exhaustive introduction to the field of Computer Algebra. In the context of this project, it will suffice to make some remarks on the characteristics of general purpose CASs as compared to theorem provers.

CASs are designed to perform symbolic computation more quickly and more accurate than people. A CAS is basically a collection of rewriting algorithms and a programming language to combine these algorithms. It is important to realize, however, that these algorithms operate on specialized datatypes, representing the symbolic expressions, which are normally not present in general purpose programming languages. The
implementers of a CAS make a choice which datatypes to include. Although the user can add his own algorithms, he or she is limited by what mathematical notions that can be expressed by the available datatypes. The user interface, especially the pretty-printing methods for displaying the result, are also an essential feature of many CAS.

**Datatypes.** First, instead of using the integers provided by the computer hardware, which are limited in range, a CAS generally implements a datatype for integers of arbitrary range. Rational numbers are usually implemented as pairs of nominator and denominator. Furthermore, there is usually a type for polynomials implemented as a list of coefficients. Algebraic elements and functions are stored as the polynomials of which they are roots. Other datatypes include vectors, matrices, infinite sums (power series), and integrals.

**Algorithms.** The most common algorithm is *simplification*. When a class of objects has a canonical representation, simplification is the reduction of an expression to its canonical representation. Simplification can be used to decide equality for that class. It is also used to keep intermediate results small, although in some cases simplification should be avoided since the canonical forms may expand into very large terms. Other algorithms include finding roots of polynomials, factorization, Euclidean GCD, Chinese Remainder Theorem, Buchberger’s algorithm, formal integration. To the casual user, a CAS is nothing more than a collection of algorithms. Most CASs have an internal programming language to combine the algorithms.

**User interface.** The user interface is a key ingredient of most CA systems. Attention to user interface issues has contributed greatly to the acceptance of these systems in the mathematical community. The system does not basically bother the user with the internal representation. The user enters commands corresponding to the algorithms, such as “simplify” or “factorize”. These commands are applied to argument expressions. For some systems the input may even consist of just an expression, in which case the implicit command is to simplify the expression. The semantics of a command is not always clear as one command may activate different algorithms in different circumstances depending on the type of the expression. Usually “simplify” means “reduce to canonical normal form”, and “solve” means “find a value for some indeterminate parameter”. The input language in which these expressions are specified is simple and applicative. It is as close as possible to conventional mathematical notation, for instance:

\[
\text{Int}(\sin(x^5), x)
\]

Values can be assigned to variables, so that a context is built in which variables represent mathematical objects. Variables do not need to be declared or even have a type. The above example expression is pretty-printed as follows in Maple:
Even though this is text mode Maple (a GUI interface to Maple also exists), it abstracts already a great deal from the internal representation. The difference between the input language and the pretty-printed output language apparently forms no difficulty.

The demands put on \( \mathcal{L} \) from a CA viewpoint are that one should be able to express notions equivalent to the ones expressible in the datatypes of a general purpose CA system, and computations should be efficient on the representations in \( \mathcal{L} \). A language for communicating expressions between CASs is the OpenMath language [74]. Such a language works precisely because most systems share roughly the same datatypes, although in slightly different representations. A drawback of CA systems is that the datatypes and operations are not grounded in a formal framework, there is no formal connection between datatypes representing the same mathematical concepts, and the operations that translate between those representations cannot be proved correct except on a meta-level. This may be hard because the system is built for speed.

### 1.3.3 Presentation of Mathematics

Presentation systems for mathematics have been around for a long time. The most notable of these systems are the \( \text{T}\TeX \) [59] and \( \LaTeX \) [63], document preparations systems for publishing mathematical texts. These systems are mostly used for publishing on paper. Presentation of mathematics using new media, such as the World Wide Web, is more difficult since the current standards, ASCII and HTML, do not support even the most basic mathematical notation. However, work is underway to build new open standards that do support such notation, for instance MathML [21], and also closed standards such as PDF [2].

**User interface.** The \( \text{T}\TeX \) and \( \LaTeX \) systems mentioned above have a batch interface, which means the user prepares an input file and runs the system on it to produce the document. However, more advanced graphical user interfaces are also possible, for example Mathpad [83], which provides a structure editor like interface and can produce documents in a variety of formats among which \( \LaTeX \).

**Representation.** Even if we only want to present mathematics, it is most useful to use formal mathematics. At first glance, presentation of mathematics is not really concerned with formal mathematics. However, readers interested in mathematical content want more than just to read symbols. Different views are possible when content is formal. Even though from a presentation point of view the formality was initially perceived as a
major problem, it turns out that the storage of formal content actually makes interactive presentation possible.

There are differences in how the three present day Computer Mathematics systems, which have been discussed (TP-, CA-, and Presentation systems), represent the mathematical content. If one implements a system for correctly type-setting mathematical formulas, such as the \( \text{\LaTeX} \) system, it suffices to store only superficial presentation information. For example, type-setting the expression \((x^y \cdot x^z)\) requires knowledge within the system about bounding boxes and baselines, but not necessarily about the meaning of the symbols or even their arity or types.

More structure is needed when one wants to implement symbolic manipulation systems such as computer algebra systems. Consider for example a computer algebra system transforming the expression from the above example \((x^y \cdot x^z)\) into \(x^{(y+z)}\). In order to perform this operation there is no need to know the exact semantics of the symbols, except for the rewrite rule that is applied here. It is, however, crucial to know the exact syntactic structure of the expression. Internally the manipulations take place on a tree-like datastructure which captures the syntactic structure and which can be pretty-printed in a more appealing format. Note that there is a distinction between content and presentation here.

Still more structure is needed in theorem proving systems. After all, not only the exact syntactic structure of expressions, but also some semantical properties need to be specified before one can prove anything about an object. In many theorem provers this is achieved by allowing the user to define objects entirely in the logical language of the system. This has the one advantage that these systems are very general and able to deal with any mathematical theory. The drawback is that all of the underlying mathematics has to be formalized inside the object language of the system, before one can use a theory. It also means that the presentation of mathematics remains extremely close to the representation of the mathematical content. This is a drawback of many theorem provers: the content is not presented well to the end user. A more fundamental question arises here: Since the mathematical concepts are represented by concepts in the underlying logical system, can the resulting it still be presented “representation free”?

## 1.4 Conclusions

In Computer Mathematics we use computers to manipulate mathematical objects. Three forms of Computer Mathematics already exist:

1. For a Theorem Prover the formalized world is implemented through the object language of the logical system underlying the TP. Potentially all of mathematics can be reduced to this object language. A problem with a formalized world that is implemented like this is that everything must be expressed inside this object language. Such encodings may cause slow computations and result in mathematical content that is hard to present naturally.
2. A Computer Algebra system performs symbolic computations on the objects in a formalized world. The formalized world of a CAS differs from the formalized world of a TP in that it is implemented by the programmer of the CAS through a number of datastructures. These datastructures are geared towards efficient computations. This makes the CA world less flexible than the TP world where all mathematical notions are defined in the object language. However, most CA systems come with a Turing complete programming language, so in principle the user can extend the system with new datatypes and algorithms.

3. Document preparation and presentation tools, in their present form implement a very shallow formalized world. Potentially a more formal representation may help to produce truly interactive mathematical documents.

The problems studied in this thesis are:

**Formalization of Mathematics.** How to formalize mathematical theories in a TP? There are different reasons why formalization may be difficult. Formal mathematical discourses are given in much more detail than informal mathematics, and implementation choices have to be made.

**Computations and proofs.** Computations have to be encoded in this object language, unless we do them on the meta level using tactics. In either case the resulting computations are not particularly efficient. Formalization also restricts our freedom to choose a representation which allows for more efficient computation. How to do efficient computations in TP? Can we do proof automation using those computations? Can we use computation engines such as CAS in TP?

**Presentation of mathematics.** If we reduce all mathematics to the formal language of a theorem prover, we make certain choices. Therefore, we add implementation details and we lose the possibility to implement things differently. Can mathematics once “reduced” to formal mathematics be presented as “real” informal mathematics?

An attempt to answer these questions is given in the remainder of this thesis which is organized as follows.

- Chapter 2 presents a gentle introduction to type theoretical theorem proving. We argue that type theory is a good candidate for the language $\mathcal{L}$ we are looking for. The concrete type theory that is introduced in Chapter 2 is the calculus of inductive constructions, the system behind the Coq theorem prover.

- Chapter 3 investigates computations in theorem provers, with an application to automated theorem proving for certain classes of problems. This is done using the reflection method, which is described in that chapter.
1.4. CONCLUSIONS

- In Chapter 4 focuses on presentation of mathematics using so-called interactive mathematical documents. We describe an implementation of a tool which presents formalized mathematical theories as interactive natural language mathematical documents.

- Chapter 5 demonstrates that theorem provers and computer algebra systems can be combined to get efficient albeit formal proofs. The example in this demonstration is based on Pocklington’s criterion, a criterion for testing primality of numbers.

- Chapter 6 presents the conclusions of this thesis. Here we come back to the questions raised in the current chapter, and see what we can do in the future to make the ideal mathematical workspace a reality.
Chapter 2

Type Theory for Theorem Provers

2.1 Introduction

In the previous chapter a class of computer mathematics applications, the so-called theorem proving assistants, is sketched. TPs help to formalize informal mathematical theories into some formal logical language. The TP can then be applied to verify proofs of statements about the objects in the mathematical theory.

The ideas on TPs put forward in Chapter 1, where we compared this class of systems to CASs and presentation systems, are general and apply to any theorem prover. This generality is desirable, since we do not want to restrict ourselves to one specific formalism, especially since the new field of Computer Mathematics is still under development. It does not seem appropriate to make too concrete commitments to any one framework or technology. However, in the next chapters we do want to apply today’s theorem proving technology to some real case studies, if only to get an idea what is missing and to be able to suggest improvements. This means that we have to focus on some particular theory or tool. In this chapter we make the choice to only consider theorem proving based on type theory, and more particularly the type theory of the theorem prover Coq [11]. This chapter provides a gentle introduction to the field of type theory from a theorem proving point of view. We describe a large part of the type theoretical framework underlying the Coq theorem prover called the Calculus of Inductive Constructions (CIC) [77]. Furthermore, we show the expressive power of CIC.

2.2 Type Theory

In type theory everything revolves around judgments which are statements of the form:

\[ M : A \]

Where \( M \) is a term and \( A \) is its type. This judgment is pronounced as “\( A \) is the type of \( M \)”, or “\( M \) inhabits \( A \)”. In type theory \( M \) and \( A \) are expressions which may contain free
variables. Type information about those variables is collected in a context $\Gamma$ and so CIC judgments are of the form:

$$\Gamma \vdash M : A$$

We will come back to contexts below.

The classical use of type assignment systems is categorization of objects in classes of similar objects. Such a categorization is very useful to prevent mistakes in computations. When doing, for example, everyday calculations, people make mistakes in how they calculate, but also mistakes in what they calculate. For example, in physics, when computing the speed of an object we measure the distance it travels and divide this number by the time that elapses during the measurement. To each of these numbers, speed, distance, and time, we assign a unit, meters per second, meters, and seconds for example. These units correspond to types. By doing the computation on the units, we see that the equation (speed equals distance divided by time) makes sense. In a way, the type checking computation is isomorphic to the computation, the type check is an “abstract interpretation” of the actual computation. Therefore, if a type check is successful, we have partial correctness.

The ability to check for partial correctness has contributed a lot to the popularity of type theory in the theory and practice of programming languages. Some mistakes can be detected at compile time with a minimal amount of effort on the programmer’s part. All that is expected from the programmer is that he gives explicit type information about variables and functions or procedures in the program source code. Put differently, in programming, type theory plays an important role in preventing a programmer from writing down program constructions that make no sense.

The role of type theory in formalizing mathematics, as done in a type theoretical proof assistant, is similar but takes place on a much more fundamental level. Here we also want to prevent a user from writing down mathematical constructions that make no sense. However, the type system is something which cannot be seen apart from the object language, it is an integral part of this language. The role of type theory in such theorem proving assistants is better understood as making a pseudo language more precise, weeding out the pseudo expressions that we do not want in.

This two-stage style of defining the object language makes the description of the language very compact. Together with some other tricks, notably the Curry-Howard-De Bruijn isomorphism described below, this makes it fairly easy to build a verification algorithm. Therefore type theory is trustworthy. We will see in this Chapter that it is also quite expressive, but in its pure form not very efficient for concrete computations. However, efficiency can be traded for conciseness. In other words, type theory is a good candidate for the language $L$ we were looking for in the previous chapter.

In this chapter we limit ourselves to describing the type theory underlying the Coq theorem prover. That is, we describe a large part of CIC, the Calculus of Inductive Constructions. CIC is based on the Calculus of Constructions, which is described first. Since we will use Coq to formalize some case studies in the other chapters, our focus will be on how to formalize mathematics in CC and CIC. This is done in Coq by encoding
the mathematical primitives directly in the calculus. Therefore the goal of this chapter is to show that CIC is powerful enough to allow this.

The outline of this chapter is as follows. In the next section we discuss some general ideas about type theory and theorem proving based on type theory. Next, in Section 2.4, we describe the calculus of constructions CC and in Section 2.5, the calculus of inductive constructions. Both these sections present the calculus itself as well as many illustrative examples showing how mathematics can be encoded in type theory. In Section 2.6 we make some remarks about concrete Coq syntax and its relation to the CIC notation we use in this chapter. Finally, in Section 2.7 we provide the conclusions of this chapter.

### 2.3 Theorem Proving based on Type Theory

This section gives a short introduction to theorem proving based on type theory. It introduces the basic ideas that make type theoretical theorem proving interesting: The Curry-Howard-De Bruijn isomorphism and contexts. These ideas apply to any type theoretical theorem prover, in this chapter we restrict ourselves to part of the Coq system. The calculus of Coq is called the Calculus of Inductive Constructions.

As outlined above, it is common in the field of type theory to define concepts in two stages. First, an easy context-free definition of a large set of pseudo elements. Second, restrictions on pseudo elements to weed out the not so interesting elements. A typing system can be seen as such a filter, assigning types to only the meaningful pseudo terms. But a typing system is more. It also categorizes the pseudo elements, and gives them a partial meaning. A type system for CIC is presented in Section 2.5, where we present the terms of CIC and describe their use in formalizing mathematical notions.

#### 2.3.1 Curry-Howard-De Bruijn isomorphism

The most important idea behind applying type theory to theorem proving is the ‘propositions as types’ or ‘proofs as terms’ correspondence, originally due to Curry, Howard [52] and De Bruijn [69]. This interpretation encodes proofs as typed λ-terms. Under this interpretation a statement “$M : A$” can be read in two ways:

- $M$ is an element of the set denoted by $A$,
- $M$ is a proof of the proposition denoted by $A$.

This means that checking the validity of a proof is as easy as checking the type of a term. In the case that $M$ denotes a proof, one can (in general) really construct a natural deduction style derivation starting from the proof term $M$. Whether this is possible depends on the specific type theory, but for many well-known logics an isomorphic typed λ-calculus has been defined: there is a bijection between natural deductions in the logic and proof terms in the typed λ-calculus. We shall illustrate this correspondence between logic and typed λ-calculus later by some examples. The main consequences of this approach towards theorem proving are that
• Proof checking is Type checking,

• Interactive Theorem Proving is the interactive construction of a term of a given type.

The Proof Assistant Coq is an interactive theorem prover based on type theory: the implemented typed $\lambda$-calculus is a version of constructive higher order logic with powerful inductive types. The system Coq provides the user with powerful tactics to interactively construct a proof term. In this construction process, the system guarantees the type correctness. An important distinction to be made – which is a basic philosophy behind type theoretic provers like Coq – is the one between

1. Checking an alleged proof: this is easy, comparable with checking the syntactic correctness of a computer program,

2. Constructing a proof for a given formula: this is hard (undecidable in general), comparable with constructing a program which satisfies a specification.

In type theoretic provers, the first task is performed by a type checking algorithm, the second task is performed interactively with the user.

2.3.2 Contexts

A very important concept in Type Theory is the notion of context. Mathematics consists not only of expressions or formulas. The formal parts of a mathematical document depend on each other through mechanisms of assumption and definition. The term calculus described below on its own is not enough to represent this feature of mathematics. The notion of theory is very important. This is why we introduce contexts. A context is a list of context items. Each context item is either an assumption or a definition.

An assumption $x : A$ is used to declare a new symbol with its type. It states a name $x$ for the new symbol and it states the type $A$. The symbol can be a mathematical object whose existence is assumed or it can represent an axiom which is assumed to hold, in the latter case $A$ is a proposition and $x$ is a name for the axiom.

A definition $x := M : A$ is used to declare a new symbol $x$ with its type $A$, but a definition also states a defining term $M$ for the symbol.

The new symbol declared by an assumption or a definition may be used in the other context items in the remainder of the context. Formally contexts are generated by the following grammar.

Definition 2.3.1 Abstract syntax (in BNF) for pseudo contexts. The empty context is denoted as $\varepsilon$. Let $S$ be a countable set of individuals. The set $T$ of terms will be defined later.

$$
\Gamma ::= \varepsilon \mid \Gamma, S : T \mid \Gamma, S := T : T
$$
Not every pseudo context generated by the above grammar is valid. There are a number of restrictions, mostly dealing with variables. A context is not valid if variables from $S$ are declared more than once in the same context or if they are used before they have been declared. We will not make these restrictions precise here.

The set of terms $T$ is given in the next two sections, where we choose two concrete type systems: CC and CIC.

### 2.4 Calculus of Constructions

The Calculus of Constructions is usually given using a two stage definition. First, an abstract syntax for pseudo expressions. Second, restrictions on the set of pseudo expressions to allow only those pseudo expressions representing potentially meaningful mathematical terms. Such a two stage definition is much more powerful than just a context free grammar. The restrictions are usually presented using a type system, assigning types to those pseudo expressions that are potentially meaningful.

Notice that CC is a system in Barendregt’s cube and therefore it is also a PTS [7]. In fact CC is the strongest system in the cube. This means that we may use results proven about PTSs in general. Below we will mention such properties without proof.

#### 2.4.1 Pseudo Expressions

The abstract syntax for pseudo expressions is given first, the type system is given below in Section 2.4.3.

**Definition 2.4.1** Abstract syntax for pseudo expressions. Let $V$ be a countable set of individuals.

$$
T ::= V | S | \text{Set} | \text{Prop} | \text{Type} | TT | \lambda V : T . T | \Pi V : T . T
$$

The calculus of constructions contains the usual expressions found in typed lambda-calculi: Variables to be bound by $\lambda$-abstraction and $\Pi$-abstraction, symbols defined in the context, sorts $\text{Set}$, $\text{Prop}$, and $\text{Type}$, for datatypes, propositions, and higher order types respectively. The sorts are described below in Section 2.5.4. Function application, $\lambda$-abstraction to form functions, and $\Pi$-abstraction to form function types. Function application is a binary operator, but using *currying* functions of higher arity can be emulated. Functions are formed by $\lambda$-abstracting a variable over a term.

$\Pi$-abstraction forms dependent product types, a form of generalized function types. A $\Pi$-type can be seen as a function type when it is not dependent, in which case the type will be denoted using the $\to$ symbol, see Notation 2.4.2. The only type-forming operator in this language is $\Pi$, which comes in four flavors, depending on the type of the *domain* (the $A$ in $\Pi x : A . B$) and the type of the *range* (the $B$ in $\Pi x : A . B$). Intuitively,
a \Pi\text{-type should be read as a set of functions. If we depict the occurrences of } x \text{ in } B \text{ explicitly by writing } B(x), \text{ the intuition is:}

\Pi x : A . B(x) \approx \prod_{a \in A} B(a) = \{ f \mid \forall a \in A[f a \in B(a)] \}.

This means \Pi x : A . B is the dependent function type of functions taking a term of type A as input and delivering a term of type B in which x is replaced by the input. We therefore immediately recover the ordinary function type A \rightarrow B as a special instance. Some more notational conventions are introduced.

**Notation 2.4.2** Some notational conventions for expressions of CC.

- In case \( x \notin \text{FV}(B) \), we write \( A \rightarrow B \) for \( \Pi x : A . B \). We call this a non-dependent function type.

- We omit parentheses in repeated non-dependent function types by letting them associate to the right, for example \( A \rightarrow B \rightarrow C \) denotes \( A \rightarrow (B \rightarrow C) \).

- We also omit parentheses in repeated applications by letting them associate to the left, for example \( MN \rightarrow P \rightarrow Q \) denotes \( (MN)P\rightarrow Q \).

- We use shorthand notation for repeated abstractions, for example \( \lambda x, y : A . M \) denotes \( \lambda x : A . (\lambda y : A . M) \), and in the meta-language \( \lambda \vec{x} : \vec{A} . M \) denotes \( \lambda x_1 : A_1 \ldots \lambda x_n : A_n . M \).

- \( M \equiv N \) denotes that \( M \) and \( N \) are the same term, or can be obtained from each other by renaming bound variables. \( M \) and \( N \) are said to be \( \alpha \)-convertible, see [6].

**2.4.2 Conversion**

By defining a reduction relation on the set of pseudo expressions, we model computations. We need this reduction relation to define conversion of terms. In the type assignment system in Section 2.4.3 this relation is used to make sure that computationally equivalent types have the same inhabitants.

**Definition 2.4.3** Reduction of pseudo expressions.

1. A term is a reducible expression, or redex, if it is of the form \( (\lambda x : A . M) N \). A redex can be contracted to its contractum: \( (\lambda x : A . M) N \rightarrow_{\beta} M[N/x] \).

2. The one-step reduction relation \( \rightarrow_{\beta} \) is defined by taking the compatible closure of contraction.

   - if \( M \rightarrow_{\beta} N \), then \( ZM \rightarrow_{\beta} ZN \),
   - if \( M \rightarrow_{\beta} N \), then \( MZ \rightarrow_{\beta} NZ \),
   - if \( M \rightarrow_{\beta} N \), then \( (\lambda x : A . M) \rightarrow_{\beta} (\lambda x : A . N) \),
   - if \( M \rightarrow_{\beta} N \), then \( (\Pi x : A . M) \rightarrow_{\beta} (\Pi x : A . N) \),
3. The reduction relation \( \rightarrow_\beta \) is defined by taking the reflexive, transitive closure of \( \rightarrow_\beta \).

\[
M \rightarrow_\beta M,
\]

if \( M \rightarrow_\beta N \), then \( M \rightarrow_\beta N \),

if \( M \rightarrow_\beta N, N \rightarrow_\beta L \), then \( M \rightarrow_\beta L \)

4. The conversion relation \( =_\beta \) is defined by taking the symmetrical, transitive closure of \( \rightarrow_\beta \).

\[
if M \rightarrow_\beta N, then M =_\beta N,
\]

if \( M =_\beta N \), then \( N =_\beta M \),

if \( M =_\beta N, N =_\beta L \), then \( M =_\beta L \)

There are only finitely many redexes in a term, but by contracting redexes new redexes may be formed. If a term has no redexes, it is said to be in normal form.

**Definition 2.4.4 (Normal form)** A term is a normal form if it does not have a redex as subterm.

If a term is convertible to some normal form, then this normal form is unique. This follows directly from the Church Rosser or “diamond” property, the following theorem.

**Theorem 2.4.5 (Church Rosser)** If \( M \rightarrow_\beta N_1, M \rightarrow_\beta N_2 \), then for some \( N \) one has \( N_1 \rightarrow_\beta N \) and \( N_2 \rightarrow_\beta N \).

A proof can be found in [7].

**Theorem 2.4.6 (Normalization)** If \( M \) has a normal form, then iterated contraction of the leftmost redex leads to that normal form.

A proof can be found in [6].

Every typeable expression has a normal form. Based on Theorem 2.4.5 and Theorem 2.4.6 we can check for convertibility of two terms \( M \) and \( N \). One algorithm to do this, is to reduce \( M \) and \( N \) to their normal forms and see whether they are \( \alpha \)-equivalent.

In order to check whether two terms are convertible, it may not always be necessary to reduce both terms to normal form. Weak head normal forms are useful if we want to check whether two terms are convertible without reducing both terms to normal form. Essentially a term is in weak head normal form if its head symbol stays the same, no matter which redexes are contracted.

**Definition 2.4.7 (Weak head normal form)** The terms that are in weak head normal form are generated by the following abstract syntax:

\[
W ::= (V \ T^*) \mid \Pi V : T \cdot T \mid \lambda V : T \cdot T
\]

Again, by reducing the leftmost redex, a term can be brought into WHNF. We can now test for equivalence of two terms \( M \) and \( N \) by reducing both terms to WHNF. If the resulting terms have different head symbols, then the terms are not equivalent. If the head symbols do match, then we check for \( \alpha \)-equivalence. If the terms are not \( \alpha \)-equivalent we proceed by reducing the subterms.
2.4.3 Type Assignment

These type assignment rules can be found in [7]. They filter out terms that are not well formed. We first give the part of the rules that specify the Calculus of Constructions. The rules for the Calculus of Inductive Constructions, are in Section 2.5.3. Rules are either judgments, or draw a conclusion judgment from several judgments premises (some rules also have side conditions, for instance the conversion rule).

Definition 2.4.8 Typing rules for expressions of CC. Assume $\Gamma$ is a valid context, and let $s_1$, $s_2$ range over the sorts \{Set, Prop, Type\}.

\[
\begin{align*}
\text{(Set)} & \quad \varepsilon \vdash \text{Set} : \text{Type} \\
\text{(Prop)} & \quad \varepsilon \vdash \text{Prop} : \text{Type} \\
\text{(var)} & \quad \Gamma, x : A \vdash x : A \\
\text{(weak)} & \quad \Gamma \vdash M : B \\
\Gamma, x : A & \vdash M : B \\
\text{(Π)} & \quad \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \\
\Gamma & \vdash \Pi x : A.B : s_2 \\
\text{(λ)} & \quad \Gamma, x : A \vdash M : B \\
\Gamma & \vdash \lambda x : A.M : \Pi x : A.B \\
\text{(app)} & \quad \Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A \\
\Gamma & \vdash MN : B[N/x] \\
\text{(conv)} & \quad \Gamma \vdash M : A \quad \Gamma \vdash A' : s \\
\Gamma & \vdash M : A', \ A =_{\beta} A'
\end{align*}
\]

If, using the rules, a derivation tree can be constructed with the judgment $\Gamma \vdash M : A$ as conclusion, and only axioms and the variable rule (with $\Gamma$ as context) as leaves, then $M$ is said to be typeable with type $A$ in context $\Gamma$. We list some important properties of the type assignment system from Definition 2.4.8.

Lemma 2.4.9 (Typeability of subterms) If $M$ has a type, then every subterm of $M$ has a type as well.

Theorem 2.4.10 (Subject Reduction) If $\Gamma \vdash M : A$ and $M \rightarrow_{\beta} N$, then $\Gamma \vdash N : A$.

Theorem 2.4.11 (Strong Normalization) If $\Gamma \vdash M : A$, then $M$ has a normal form.
We already saw in Theorem 2.4.6 that finding the normal form, if it exists, is computable. Theorem 2.4.11 shows that for typeable terms such a normal form always exists. This provides an algorithm to check the side condition $A =^\beta A'$ of the conversion rule.

**Theorem 2.4.12** Type checking in CC is decidable.

Type checking and type inference are in principle equally difficult, but for purposes explained in Chapter 4 (presentation of an already checked context) we only need a type inference algorithm which may assume the terms are typeable. This means that our type inference algorithm does not need to check some of the side conditions.

Type-checking is easy. The type-assignment derivation system above is easily transformed into an algorithm for type-assignment. We need this in Chapter 4 in order to determine how to verbalize an object.

### 2.4.4 Representing Mathematics

Mathematical concepts can be assumed in the context using assumption, but they can also be constructed in CC. For the construction we use *impredicative encodings*. That is, we define concepts using a quantification over the type to which the new concept belongs.

#### Sorts

The rules of Definition 2.4.8 induce a structure on the expressions, dividing them in three classes of typeable expressions: the *sorts* level, the *types* level, and the *terms* level. In Figure 2.1 some examples of sorts, types, and terms are displayed. Every expression in the calculus of Coq falls in one of these three levels. Most objects defined in a typical formalization of a mathematical document live in the terms or types level.

![Figure 2.1: Sorts, Types, and Terms.](image-url)

in the calculus of Coq falls in one of these three levels. Most objects defined in a typical formalization of a mathematical document live in the terms or types level.
Coq has three sorts called Set, Prop, and Type. Sorts serve as types for expressions on the types level. The intended meaning of Prop is the type of propositions, while Set is for datatypes. Type is inhabited by higher order types such as Prop and Set, the type of predicates $A \rightarrow \text{Prop}$, etc.

Inhabitants of sorts (apart from the sorts Set and Prop themselves) live on the types level. Examples of expressions living on the types level are the type of booleans bool and the type of natural numbers nat. Both are in sort Set. An example of a type living in Type is $\text{nat} \rightarrow \text{Prop}$ which is the type of unary predicates over the natural numbers. Note that theorems live on the same level as ‘ordinary’ types, but are characterized by the fact that they inhabit Prop.

Expressions which inhabit types are called terms and live on the lowest level. Terms are the objects that do not have inhabitants themselves. Terms that inhabit a proposition (a type inhabiting Prop) are called proof-objects.

Note that these three levels are not syntactical classes, they are induced by the typing relation. Other interpretations are possible. One can collapse the Set and Prop sorts by just doing logic in Set for example. However, the libraries of definitions and lemmas that come with Coq assume this interpretation, and so do we when we create the interactive documents.

Datatypes

As a first example, consider how the natural numbers can be constructed in Coq. We define the type of natural numbers as an expression of type Set, with a zero element and a successor function which can be used to construct elements of the type, and an iterator which can be used to eliminate elements of the natural numbers by specifying recursive functions over the type. Very similar to the iterator, we can also construct an induction scheme.

Example 2.4.13 Church numerals with constructors, addition-, multiplication functions.

\[
\begin{align*}
\text{nat}_C &:= \Pi A : \text{Set}. A \rightarrow (A \rightarrow A) \rightarrow A \\
\text{O}_C &:= \lambda A : \text{Set}. \lambda x : A. \lambda f : A \rightarrow A. x \\
\text{S}_C &:= \lambda n : \text{nat}_C. \lambda A : \text{Set}. \lambda x : A. \lambda f : A \rightarrow A. (f (n\ f\ x)) \\
\text{plus}_C &:= \lambda x : \text{nat}_C. \lambda y : \text{nat}_C. (x \text{nat}_C y \text{S}_C) \\
&\quad : \text{nat}_C \rightarrow \text{nat}_C \rightarrow \text{nat}_C \\
\text{mult}_C &:= \lambda x : \text{nat}_C. \lambda y : \text{nat}_C. (x \text{O}_C (\text{plus}_C\ x)) \\
&\quad : \text{nat}_C \rightarrow \text{nat}_C \rightarrow \text{nat}_C
\end{align*}
\]

Note that the Church numerals act as iterators, for example in the definition of $\text{plus}_C$ the “numeral” $x$ is applied to a value $y$ a unary function $\text{S}$. The $\text{S}$ function is iterated $x$ times on the argument $y$ to get the results of the function. So, the numerals themselves express the recursion principle.
While this yields an implementation of the natural numbers which is quite elegant and natural, there are some drawbacks to the Church numeral representation and impredicative encoding of datatypes in general. First of all, the Church numerals are an encoding. There is no way to hide the implementation details. For instance, each Church numeral is a polymorphic function, but the real natural numbers are not functions. When building mathematical theories on top of this implementation of the natural numbers, care has to be taken only to rely on properties of the natural numbers not on properties of the implementation.

Second, due to the specific encoding of numbers as functions, certain properties are not provable in CC. For example injectivity of the successor function $S_C$ would need the extensionality rule. Of course extensionality could be assumed as an axiom, but this is a rather heavy axiom to add just to prove injectivity of $S_C$.

Third, the encoding enforces an inherently inefficient implementation of certain recursive functions when reducing an application of such a function on concrete arguments. The classical example is the predecessor function which has a linear time complexity in its argument. Before we give a definition of the predecessor function, we need to introduce the Cartesian product with pairing function and projection functions.

Example 2.4.14 Cartesian product with pairing function and projection functions.

\[
\text{prod} := \lambda A, B : \text{Set}. \Pi C : \text{Set}. \,(A \to B \to C) \to C \to \text{Set} \to \text{Set}
\]

\[
\text{pair} := \lambda A, B : \text{Set}. \lambda a : A. \lambda b : B. \lambda C : \text{Set}. \lambda f : A \to B \to C. (f a b)
\]

\[
\text{fst} := \lambda A, B : \text{Set}. \lambda p : (\text{prod} A B). p A (\lambda a : A. \lambda b : B. a)
\]

\[
\text{snd} := \lambda A, B : \text{Set}. \lambda p : (\text{prod} A B). p B (\lambda a : A. \lambda b : B. b)
\]

Pairing is necessary in order to give the following definition of the predecessor function. Note that pairs are constructed by applying a variable $f$ to the components of the pair which is $\lambda$-abstracted. Access to the components is achieved by substituting a function for the variable $f$, i.e. applying the pair to an appropriate function.

Using pairing we can define the predecessor function by iterating its argument over the type $\mathbb{N} \times \mathbb{N}$ with initial value $(0, 0)$ and function $(n, m) \mapsto (m, m + 1)$. The predecessor function is defined as the first component of the resulting pair.

Example 2.4.15 Predecessor function on the Church numerals.

\[
\text{pred}_C := \lambda n : \text{nat}_C. (\text{fst } \text{nat}_C n) \text{nat}_C \text{nat}_C (n (\text{prod } \text{nat}_C \text{nat}_C) (\text{pair } \text{nat}_C \text{nat}_C O_C O_C))
\]
Propositional Logic

Similar to the encoding of datatypes, propositional logic is also done in CC using impredicative encodings.

**Example 2.4.16** How to do propositional logic in CC.

1. Implement implication $A \rightarrow B$ simply as the non-dependent function type $A \rightarrow B$.
2. Use impredicative encodings for the other connectives:
   
   \[
   \begin{align*}
   \text{True} & := \Pi A : \text{Prop}. A \rightarrow A \\
   \text{False} & := \Pi A : \text{Prop}. A \\
   \text{not} & := \lambda A : \text{Prop}. A \rightarrow \text{False} \\
   \text{and} & := \lambda A, B : \text{Prop}. \Pi C : \text{Prop}. (A \rightarrow B \rightarrow C) \rightarrow C \\
   \text{or} & := \lambda A, B : \text{Prop}. \Pi C : \text{Prop}. (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C
   \end{align*}
   \]
3. Terms representing the introduction and elimination rules are easily constructed.

The encoding reflects the constructive interpretation of the connectives. Note, for instance, that the definition of $\text{and}$ is very similar to the definition of Cartesian product we give above in Example 2.4.14.

Predicate Logic

Predicates are defined as functions with codomain $\text{Prop}$. We define quantifiers using impredicative encodings.

**Example 2.4.17** How to do quantifiers in CC.

1. As usual, we implement universal quantification $\forall x : A. B$ as $\Pi x : A. B$,
2. Existential quantification through a second order encoding.
   
   \[
   \text{ex} := \lambda A : \text{Set}. \lambda P : A \rightarrow \text{Prop}. \Pi C : \text{Prop}. \Pi a : A. ((P a) \rightarrow C) \rightarrow C
   \]
3. Terms representing the introduction and elimination rule are easily constructed.

**Example 2.4.18** Division and primality predicates.

\[
\begin{align*}
\text{Divides} & := \lambda n, m : \text{nat}. \exists k : \text{nat}. (\text{mult} n k) = m \\
\text{Prime} & := \lambda n : \text{nat}. (n > 1) \land \forall d : \text{nat}. (\text{Divides} d n) \rightarrow (d = 1 \lor d = n)
\end{align*}
\]
2.5. CALCULUS OF INDUCTIVE CONSTRUCTIONS

Equality

A general polymorphic equality known as *Leibniz equality* can be constructed as follows. It can be easily verified that this equality is reflexive, symmetric, and transitive.

**Example 2.4.19** Leibniz equality.

1. 

\[
\text{eq}_L := \lambda A : \text{Set}. \lambda x, y : A. \Pi P : A \rightarrow \text{Prop}. (P x) \rightarrow (P y)
\]

2. Terms representing proofs for reflexivity, transitivity, symmetry, and elimination are easily constructed.

The above examples show the power of CC. Note that CC complies to the De Bruijn criterion, its description is very simple, consisting of a small abstract syntax, one reduction rule, and a handful of typing rules. Yet, it is powerful enough to allow construction of many mathematical concepts. However, the power of CC requires a lot of encoding. This has two important drawbacks: First, it is not efficient when doing concrete computations. Second, when trying to present or communicate the mathematics we depend a great deal on the representation as impredicative encodings. The first drawback makes our work in Chapters 3 and 5 harder, as we rely on efficient computations in the theorem prover Coq. The second drawback makes presentation of mathematics, like we do in Chapter 4, harder, as the representation may be not abstract enough to allow good presentation. For example, we would like to treat the Church numerals as primitives when we present a mathematical theory based on arithmetic, but we cannot guarantee that their representation will not be used in the theories built on top of this representation. The same thing holds for the logical connectives. Therefore we introduce inductive types by adding new primitives to the calculus.

2.5 Calculus of Inductive Constructions

A basic notion in logic and set theory is induction: when a set is defined inductively, we understand it as being 'built up from the bottom' by a set of basic constructors. Elements of such a set can be decomposed in 'smaller elements' in a well-founded manner. This gives us the principles of *proof by induction* and *function definition by recursion*, which in the spirit of the Curry-Howard-De Bruijn isomorphism are the same thing.

From a foundational viewpoint, there is no need to add inductive types explicitly to the language. We can use impredicative encodings and in this way define numbers, propositional and predicate logic with equality. Yet, for efficiency and convenience reasons it is good to have inductive types. More importantly, inductive types allow definitions of a concept which really only capture the properties of that concept. Using an impredicative encoding allows multiple interpretations, for example presentation of “proofs by induction” (as we do in Chapter 4) becomes harder.
If we want to add inductive types to our type theory, we have to add a definition mechanism that allows us to introduce a new inductive type, by giving the name and the constructors of the inductive type. The theory is able to automatically generate a scheme for proof-by-induction and a scheme for primitive recursion. It turns out that this can be done very generally in type theory, including very many instances of induction. Here we shall use a variant of the inductive types that are present in the system Coq [11] and that were first defined in [32].

2.5.1 Pseudo Expressions

To facilitate inductive types, we add three new constructions for building pseudo expressions: ind, constr, case, and fix.

Definition 2.5.1 Grammar for pseudo expressions. Let \( V \) be a countable set of individuals.

\[
T ::= V | S | Type | Set | Prop | TT | \lambda V : T.T | \Pi V : T.T | \langle T \rangle \text{ case } T . \{ T^* \} | \text{fix}_N V : T.T | \text{ind } V : T.\{ T^* \} | \text{constr}_N T
\]

The asterisk (*) denotes a finite list of terms. Meta variables of such lists will be given in “vector notation” such as: \( \vec{M} \).

We already encountered the CC primitives \( \lambda, \Pi \), and application in Section 2.4.1. What makes CIC the calculus of inductive constructions are case, fix, ind, and constr. These primitives deal with inductive types, constructors of inductive types, case analysis on inductive types, and recursive functions over inductive types.

Inductive types are formed using the ind primitive. It contains the universe the inductive type lives in and a type for each constructor. The \( i \)th constructors of an inductive type can be selected using the constr primitive with index \( i \). In Example 2.5.2 below, the type of natural numbers nat is defined inductively with two constructors. For clarity, the constructors are explicitly given names O for constructing the zero number and S for constructing a successor number.

Example 2.5.2 Natural numbers. Define nat using an inductive type with constructors O and S.

\[
nat ::= \text{ind}_X : \text{Set.} \\
\{ X, X \rightarrow X \} \\
O ::= (\text{constr}_1 \text{nat}) : \text{nat} \\
S ::= (\text{constr}_2 \text{nat}) : \text{nat} \rightarrow \text{nat}
\]
The **case** construction eliminates values of inductive types by doing a case distinction, one case per constructor. It takes a term and a list of values. An example is given in Example 2.5.3. In the case distinction in this example for each of the two constructors of nat one **guard** term is present. If \( n \) is built with the \( \mathbb{O} \) constructor, then \( m \) is the result. If \( n \) is built with the \( \mathbb{S} \) constructor, say \( m \) equals \( (\mathbb{S} p) \) then the result is computed using \( p \). The fact that this \( p \) may be used is indicated in the second guard term by specifying a \( \lambda \)-abstraction over \( p \). The expression between angled brackets in front of the case term is used to specify the type of the case term. In this case the type is nat. The general mechanism for checking well-formedness of case terms will become clear when we explain the typing rules in Section 2.5.3.

The fix primitive makes recursive functions possible. Although restrictions will be presented below, in Section 2.5.3, which make general recursive functions impossible, using fix functions can be specified as least fixpoints over inductive types. In a fix expression, a fixpoint variable is abstracted, representing a recursive version of the fixpoint term itself, which may be applied in the body of the term. The operational semantics of fix becomes clear in Section 2.5.2. With the combination of case and fix, recursive functions can be defined over inductive types in the style of functional programming. For example in Example 2.5.3 the addition function is defined using a fix and a case.

**Example 2.5.3** The addition function on nat defined with recursion to its first argument.

\[
\text{plus} := \text{fix}_1 \; f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}.
\lambda n, m : \text{nat}. (\lambda n : \text{nat}. \text{nat}) \text{case } n : \text{nat}.
\{ m, \\
\lambda p : \text{nat}. (\mathbb{S} (f \; p \; m)) \}
\]

Coq puts an elimination predicate in the context for every inductive type. The elimination predicate is not primitive but is defined automatically in terms of case and fix. We do not specify how this predicate is generated in general, the reader is referred to [42].

**Example 2.5.4** The elimination predicate for nat is generated automatically by Coq (defined in terms of case and fix).

\[
\text{natelim} := \quad \lambda P : \text{nat} \rightarrow \text{Prop}.
\lambda h_0 : (P \; \mathbb{O}). \lambda h_1 : (\Pi n : \text{nat} . (P \; n) \rightarrow (P \; (\mathbb{S} \; n))).
\quad \text{fix}_1 \; f : (\Pi n : \text{nat} . (P \; n)). \\
\lambda n : \text{nat}. (\lambda n : \text{nat} . (P \; n)) \text{case } n : \text{nat}.
\{ h_0, \\
\lambda m : \text{nat} . (h_1 \; m \; (f \; m)) \}
\quad \Pi P : \text{nat} \rightarrow \text{Prop}.
\quad (P \; \mathbb{O}) \rightarrow (\Pi n : \text{nat} . (P \; n) \rightarrow (P \; (\mathbb{S} \; n))) \rightarrow \\
\quad \Pi n : \text{nat} . (P \; n)
\]


A similar term can be defined for the Set and Type sorts. However, when defining functions over an inductive type it is customary to use case and fix, while in proving something over an inductive type the elimination predicate is used. We take this into account when we study the verbalization of proof-objects in Chapter 4, where the elimination predicates are treated as primitive.

2.5.2 Conversion

The case and fix constructions each have their own reduction rules. The syntactic conditions under which an expression involving these constructions reduces are a little harder to formulate. Furthermore, we need to specify new compatibility rules for the relations.

**Definition 2.5.5** Reduction of pseudo expressions.

1. Reducible expressions and contraction.
   \[(\lambda x: A.M) N \rightarrow_{\beta_r} M[N/x]\]
   \[\langle Q \rangle \text{ case} (\text{constr}, I): A \{B_1, \ldots, B_n\} \rightarrow B_i\]
   \[(\text{fix}_k f: B.M) \bar{P} ((\text{constr}, I)\bar{N}) \rightarrow_{\iota} M[(\text{fix}_k f: B.M)/f] \bar{P} ((\text{constr}, I))\]

2. We extend the one-step reduction relation with the following compatibility rules:
   - If \(M \rightarrow_{\beta_r} N\), then \(ZM \rightarrow_{\beta_r} ZN\),
   - If \(M \rightarrow_{\beta_r} N\), then \((\lambda x: A.M) \rightarrow_{\beta_r} (\lambda x: A.N)\),
   - If \(M \rightarrow_{\beta_r} N\), then \((\Pi x: A.M) \rightarrow_{\beta_r} (\Pi x: A.N)\),
   - If \(A \rightarrow_{\beta_r} A'\), then \((\text{ind}_X: A.\{\vec{G}\}) \rightarrow_{\beta_r} (\text{ind}_X: A'.\{\vec{G}\})\),
     similar for each of the \(C_i\),
   - If \(M \rightarrow_{\beta_r} N\), then \((\text{constr}_i M) \rightarrow_{\beta_r} (\text{constr}_i N)\)
   - If \(M \rightarrow_{\beta_r} M'\) then \((\text{case}_M: A.\{\vec{G}\}) \rightarrow_{\beta_r} (\text{case}_M': A.\{\vec{G}\})\),
   - If \(A \rightarrow_{\beta_r} A'\) then \((\text{case}_M: A.\{\vec{G}\}) \rightarrow_{\beta_r} (\text{case}_M: A.\{\vec{G}\})\),
     similar for each of the \(G_i\),
   - If \(A \rightarrow_{\beta_r} A'\), then \((\text{fix}_k f: A.B) \rightarrow_{\beta_r} (\text{fix}_k f: A'.B)\),
   - If \(B \rightarrow_{\beta_r} B'\), then \((\text{fix}_k f: A.B) \rightarrow_{\beta_r} (\text{fix}_k f: A.B')\),

3. The reduction relation \(\rightarrow_{\beta_r}\) is defined by taking the reflexive, transitive closure of \(\rightarrow_{\beta_r}\).

4. The conversion relation \(\equiv_{\beta_r}\) is defined by taking the reflexive, transitive, symmetrical closure of \(\rightarrow_{\beta_r}\).

Like in CC, the above definition is also easily turned into an algorithm, as leftmost outermost reduction always leads to a normal form.

The definition of weak head normal forms is different from the CC version, in order to take the two new redexes into account.
Definition 2.5.6 (Weak head normal form) The terms that are in weak head normal form are generated by the following abstract syntax:

\[
M ::= (V \mathbf{T}^*) \mid \Pi V : T. T \mid \lambda V : T. T \mid <T> \text{case } M \{ T^* \} \mid (\text{fix}_i V : T. T) V^* M
\]

\[
W ::= M \mid \text{constr}_i T^*
\]

2.5.3 Type Assignment

As with CC, the type assignment rules determine which pseudo expressions are well-formed and which are not. The new primitives \text{ind}, \text{constr}, \text{case}, and \text{fix} are used for introducing inductive types, access to the constructors of inductive types, case analysis over terms of inductive types, and specification of recursive functions over inductive types. It is important to make the type assignment rules in such a way that non-terminating computations cannot be specified, as non-termination leads to inconsistency of the logic. Preventing specification of non-terminating computations makes the syntactic restrictions, present in the form of side conditions, rather involved. The presentation of the side conditions is taken from Gimenez [42].

In addition to the typing rules from Section 2.4.3, we give four more rules to assign types to the new primitives. We only give rules for the new constructions since the other rules do not change, but it is understood that the conversion rule is adapted to use the new conversion relation $=_{\beta \iota}$. Note that these rules have side conditions and not just simple judgments as assumptions:

- The \text{ind} rule uses a condition called “form of constructor w.r.t. $X$” which is explained below.

- The \text{constr} rule needs to make sure that the index $i$ of the constructor we want to select is within the bounds, i.e. does the inductive type have at least $i$ constructors?

- The \text{case} rule uses “Left/right substitution”, denoted with $S$, which is explained below.

- The \text{fix} rule uses “guarded by destructors”, denoted with $D$, which is explained below.

These side conditions prevent construction of infinite datastructures and computations.
Definition 2.5.7 Typing rules for ind, constr, case, and fix. In the following rules let \( I \equiv \text{ind}X : s.\{\vec{C}\} \) and \( N \equiv \lambda \vec{x} : \bar{T}.\lambda y : (I \bar{P}).M. \)

\[
\text{(ind)} \quad \frac{\Gamma \vdash A : Type \quad \Gamma, X : A \vdash C_i : s}{\Gamma \vdash \text{ind}X : A.\{\vec{C}\} : A}, \ C_i \text{ is form of constructor w.r.t. } X
\]

\[
\text{(constr)} \quad \frac{\Gamma \vdash \text{ind}X : A.\{\vec{C}\} : s}{\Gamma \vdash (\text{constr}_i (\text{ind}X : A.\{\vec{C}\}))) : C_i[(\text{ind}X : A.\{\vec{C}\})/X]}, \ 1 \leq i \leq |\vec{C}|
\]

\[
\text{(case)} \quad \frac{\Gamma \vdash Q : \Pi \vec{z} : \bar{Z}.(I \vec{z}) \rightarrow s \quad \Gamma \vdash M : (I \bar{P}) \quad \Gamma \vdash G_i : S(C_i, I, Q, (I)_i)}{(Q) \text{ case } M : (I \bar{P}) \{\vec{G}\} : (Q \bar{P} M)}
\]

\[
\text{(fix)} \quad \frac{\Gamma \vdash N : B}{\Gamma \vdash (\text{fix} f : B.N) : \Pi x : A.B}, \ D(f, k, x, M)
\]

We now present in more details the restrictions that act as side conditions to the above rules.

Restrictions on ind. The restriction on ind formation is called “forms of constructor”, and specifies what the type of a constructor of an inductive type may be. The notion of strictly positive occurrence is needed to define “forms of constructor” below.

Definition 2.5.8 (Strictly positive occurrence) Let \( P \) be a term. A variable \( X \) occurs strictly positively in \( P \) if \( P \equiv \Pi \vec{z} : \bar{M}.(X \bar{N}) \) and \( X \not\in \text{FV}(M_i) \) and \( X \not\in \text{FV}(N_i) \).

“Forms of constructor” specifies what the types of the constructors of an ind expression may look like.

Definition 2.5.9 (Forms of constructor) Let \( X \) be a variable. The terms which are a form of constructor with respect to \( X \) are generated by the abstract syntax:

\[
\text{Co ::= } (X \bar{N}) \mid P \rightarrow \text{Co} \mid \Pi x : M.\text{Co}
\]

with the following restrictions on \( X : X \not\in \text{FV}(N_i), X \text{ is strictly positive in } P, X \not\in \text{FV}(M) \).

Restrictions on case. For case formation, the guards need to correspond to the number and shape of constructors of the inductive type over which a case analysis is performed. Each of the “guards” of the case analysis corresponds to a constructor of the inductive type. This can be checked by performing a so-called “Left/right substitution” on the constructors. The guard should be typeable with the result of the substitution after applying it to the type of the constructor.
**Definition 2.5.10 (Left/right substitution)** Let $C$ be a form of constructor with respect to $X$, we define the Left/right substitution of $I$ for $X$ in $C$, $S(C, I, Q, R)$, with induction over the form of constructor $C$.

\[
S((P \to C), I, Q, R) = \Pi y: P[I/X], S(C, I, Q, (Ry)) \\
S((\Pi x: M.C), I, Q, R) = \Pi x: M, S(C, I, Q, ( Rx)) \\
S((X \bar{N}), I, Q, R) = (Q \bar{N} R)
\]

**Restrictions on fix.** The restrictions on fix formation are necessary to prevent the formation of non-terminating functions. Non-terminating functions would cause inconsistency, so we have to avoid them at all cost. Restrictions on fix formation is called “guarded by destructors”. But we first need the following two definitions.

In recursive calls the function may only be applied to terms that are structurally smaller than the term we are matching against. The arguments to the function in a recursive call are presented in the form of variables which are $\lambda$-abstracted in the terms of a case construct. Therefore we define the notions of pattern variables and components.

**Definition 2.5.11 (Component)** Let $x$ be the $k$th $\lambda$-abstraction of $N$. A component of $x$ is a term $(z \bar{P})$ with $z$ being a pattern variable of the case analysis which protects the application.

Checking that the fixpoint variable is only applied to a component in its $k$th argument is not enough. The notion of recursive position is really needed. Gimenez [42] has a counter example of non-terminating $f$ if one does not have RP.

**Definition 2.5.12 (Recursive position)** Let $C \equiv \Pi \bar{x}: \bar{M}.(X \bar{N})$ be a form of constructor with respect to $X$. We say that $j$ corresponds to a recursive position of $C$ if the variable $X$ appears in the term $M_j$. We denote this property by $RP(j, C)$.

Finally, we can define “guarded by destructors”, which is the side condition which restricts the formation of fix terms. Informally a term $(\fix_k f : B.N)$ should meet four requirements.

- $f$ may occur in $N$ only as the head of an application.
- Any application of $f$ must be guarded by a case analysis on the $k$th $\lambda$-abstraction of $N$, say $x$ is the name of the variable that is abstracted.
- The $k$th argument in an application of $f$ must be a component of $x$.
- Only recursive components of $x$ are allowed to be the argument of a recursive call.

The “guarded by destructors” condition is computed using the following recursive definition. The $D(f, k, x, M)$ in Definition 2.5.7 is actually $D_0(f, k, x, M)$ and we leave out $f, k, x$ because they never change.
Definition 2.5.13 (Guarded by destructors) Let $k$ be a positive integer. Let $M$ be a term, (intuition: $M$ is the body of a fix term, after removing all $\lambda$-abstractions). Let $f$ and $x$ be two variables and $V$ a set of variables. The $D(f, k, x, M)$ condition on fix expressions is computed as follows.

- If $f \notin \text{FV}(M)$, then $D_V(M)$ holds without conditions.
- If $M \equiv \lambda z: P.Q$, then $D_V(P) \land D_V(Q)$.
- If $M \equiv \Pi z: P.Q$, then $D_V(P) \land D_V(Q)$.
- If $M \equiv \text{fix}_p f: A.N$, then $D_V(A) \land D_V(N)$.
- If $M \equiv \text{ind}X: A.\{\vec{C}\}$, then $D_V(A) \land D_V(\vec{C})$.
- If $M \equiv \langle Q \rangle \text{ case } N: S.\{\vec{G}\}$, then
  - if $N \equiv (z \vec{P})$ for some $z \in V \cup \{x\}$, then
    * $S \equiv (I \vec{R}) \land I \equiv (\text{ind} X: A.\{\vec{C}\})$.
    * $D_V(Q) \land D_V(S) \land D_V(\vec{P})$.
    * if $C_i \equiv \Pi \vec{y}: \vec{T}.(X \vec{K}), G_i \equiv \lambda \vec{y}: \vec{T}[I/X].E$, then $D_U(E)$, where $U = V \cup \{y_j|\text{RP}(j, C_i)\}$.
  - else, $D_V(Q) \land D_V(N) \land D_V(S) \land D_V(\vec{G})$.
- If $M \equiv (N \vec{P})$, then
  - If $N \equiv f$ with $|\vec{P}| \geq k$, then $P_k \equiv (z \vec{Q})$ with $z \in V \land D_V(\vec{P})$
  - else, $D_V(N) \land D_V(\vec{P})$.

Every term specified with fix and cases can be rewritten using (primitive recursive) recursors, and vice versa. We do not give the translation algorithm, interested readers are referred to [42]. As a result the calculus is as powerful as a calculus with explicit recursors. However, the current presentation of recursive functions is more user friendly since a user can specify recursive calls just like in a general functional language. The drawback of this presentation is that, to a casual user it may not always be clear why a specification of a recursive function is not allowed.

Inductive types give more efficient computations and make the specification language more precise. In CIC we can really capture for example the notion of natural number, while in CC the Church numerals defined in Example 2.4.13 have “other” properties. Both of these properties (efficiency and precision) are important for the work presented in the other chapters in this thesis. However, the inductive types come at a certain price. Compared to CC, CIC has many more restrictions to check in the typing rules of the new primitives. This makes the type checking algorithm more complex and thus less trustworthy. The question arises if CIC still conforms to the De Bruijn criterion.
2.5. REPRESENTING MATHEMATICS

2.5.4 Representing Mathematics

We demonstrate the power and precision of CIC with some examples of mathematics that can be formalized. Before we give examples, please note that we make a choice here on how to formalize mathematics in CIC. This is an interpretation of the sorts. Different interpretations are possible. The presentation in Chapter 4 depends on the choice we make here.

Data-types

The inductive types make creating basic datatypes such as booleans, natural numbers, lists, and trees easy. Specifications is made as friendly as in most functional languages. We have already encountered the natural numbers in Example 2.5.2. The booleans are probably an even simpler example of an inductive type.

Example 2.5.14 Booleans.

1. Define bool using an inductive type with constructors true and false.

\[
\begin{align*}
\text{bool} & := \text{ind } X : \text{Set.} \\
& \{ X, X \} \\
\text{true} & := (\text{constr}_1 \text{bool}) : \text{bool} \\
\text{false} & := (\text{constr}_2 \text{bool}) : \text{bool}
\end{align*}
\]

2. The elimination predicate is generated automatically by Coq (defined in terms of case and fix).

\[
\begin{align*}
\text{boolelim} : & \Pi P : \text{bool} \rightarrow \text{Prop.} \\
& (P \text{true}) \rightarrow (P \text{false}) \rightarrow \\
& \Pi b : \text{bool}. (P b)
\end{align*}
\]

The booleans and natural numbers are non-dependent inductive types. With dependent inductive types we can construct very powerful notions. As an example, consider lists of length \( n \). Here we have an example of a type \((\text{list } A n)\) depending on a term \( n \).

Example 2.5.15 Lists of length \( n \) with constructors for the empty list and adding an element.

\[
\begin{align*}
\text{list} & := \text{ind } X : \text{Set} \rightarrow \text{nat} \rightarrow \text{Set} \\
& \{ \Pi A : \text{Set}.(X A O), \\
& \quad \Pi A : \text{Set}.\Pi n : \text{nat}.(X A n) \rightarrow \text{nat} \rightarrow (X A (S n)) \} \\
\text{nil} & := (\text{constr}_1 \text{list}) : \Pi A : \text{Set}.(\text{list } A O) \\
\text{cons} & := (\text{constr}_2 \text{list}) : \Pi A : \text{Set}.\Pi n : \text{nat}.(\text{list } A n) \rightarrow \text{nat} \rightarrow (\text{list } A (S n))
\end{align*}
\]
For example the following recursive function can be specified over list. The constant function takes arguments \( v \) and \( n \), and yields a list containing \( n \) values \( v \). So, the term \((\text{constant}(42)(5))\) computes the list \([42, 42, 42, 42, 42]\), which has type \((\text{list nat}(5))\).

**Example 2.5.16**  The list of variable length filled with one \text{nat} value.

\[
\text{constant} := \text{fix}_2 f : \text{nat} \to \Pi n : \text{nat} \cdot (\text{list nat} n).
\]

\[\lambda v : \text{nat}. \lambda n : \text{nat} \cdot (\lambda n : \text{nat} \cdot (\text{list nat} n))\text{case } n : \text{nat} \cdot \{ (\text{nil nat}), \lambda m : \text{nat} \cdot (\text{cons nat } m (f v m) v) \} \]

: \text{nat} \to \Pi n : \text{nat} \cdot (\text{list nat} n)

**Propositional Logic**

The logical connectives can be built using inductive types. In these inductive types the constructors behave like introduction rules, and the elimination predicates behave like elimination rules.

**Example 2.5.17**  How to do propositional logic in CIC.

1. Implement implication \( A \to B \) again simply as the non-dependent function type \( A \to B \).

2. Define the other connectives using inductive types.

\[
\text{True} := \text{ind} X : \text{Prop} \cdot \{X\}
\]

\[
\text{False} := \text{ind} X : \text{Prop} \cdot \{\}
\]

\[
\text{not} := \lambda A : \text{Prop} \cdot A \to \text{False} : \text{Prop} \to \text{Prop}
\]

\[
\text{and} := \text{ind} X : \text{Prop} \to \text{Prop} \to \text{Prop}.
\]

\[
\{\Pi A, B : \text{Prop} \cdot A \to B \to (X A B)\}
\]

\[
\text{or} := \text{ind} X : \text{Prop} \to \text{Prop} \to \text{Prop}.
\]

\[
\{\Pi A, B : \text{Prop} \cdot (X A B), \Pi A, B : \text{Prop} b \to (X A B)\}
\]

We use infix notation \( \neg A \), \( A \land B \), and \( A \lor B \) for \((\text{not } A)\), \((\text{and } A B)\), and \((\text{or } A B)\) respectively.

3. The constructors behave like introduction rules.

\[
\text{id} := (\text{constr}_1 \text{True}) : \text{True}
\]

\[
\text{andintro} := (\text{constr}_1 \text{and}) : \Pi A, B : \text{Prop} A \to B \to (\text{and } A B)
\]

\[
\text{orintro} := (\text{constr}_1 \text{or}) : \Pi A, B : \text{Prop} A \to (\text{or } A B)
\]

\[
\text{orintror} := (\text{constr}_2 \text{or}) : \Pi A, B : \text{Prop} B \to (\text{or } A B)
\]
4. The elimination predicates are generated automatically by Coq (in terms of fix and case).

\[
\begin{align*}
\text{falseelim} & : \Pi P : \text{Prop.} \text{False} \to P \\
\text{andelim} & : \Pi A, B, P : \text{Prop.} (A \to B \to P) \to A \land B \to P \\
\text{orelim} & : \Pi A, B, P : \text{Prop.} (A \to P) \to (B \to P) \to A \lor B \to P
\end{align*}
\]

The constructors of those inductive types behave like introduction predicates. The elimination predicates are defined in terms of case and fix.

Predicate Logic

As in CC, predicates in CIC are functions with codomain Prop. The existential quantifier is defined using an inductive type.

**Example 2.5.18** How to do quantifiers in CIC.

1. As usual, we implement universal quantification \( \forall x : A.B \) as \( \Pi x : A.B \),

2. Existential quantification is defined inductively.

\[
ex := \text{ind} X : \Pi A : \text{Set.} (A \to \text{Prop.}) \to \text{Prop.} \\
\{ \lambda A : \text{Set.} \lambda P : A \to \text{Prop.} \Pi x : A. (P x) \to (X A P) \}
\]

3. The constructor of ex behaves like the introduction rule.

\[
exintro := (\text{constr}, \text{ex}): \\
\Pi A : \text{Set.} \Pi P : A \to \text{Prop.} \Pi x : A. (P x) \to (ex A P)
\]

4. The elimination predicate is generated automatically by Coq (in terms of fix and case).

\[
exelim : \Pi A : \text{Set.} ; P : A \to \text{Prop.} ; P_0 : \text{Prop.} (\Pi x : A. (P x) \to P_0) \to (ex A P) \to P_0
\]

Equality

Equality is defined using an inductive type such that it is the smallest reflexive relation of type \( (\Pi A : \text{Set.} A \to A \to \text{Prop.}) \). It is equivalent to Leibniz equality, which is defined by quantifying over all predicates.

**Example 2.5.19** Inductive equality.

1. Define equality using an inductive type.

\[
eq := \text{ind} X : \Pi A : \text{Set.} A \to A \to \text{Prop.} \\
\{ \Pi A : \text{Set.} \Pi x : A. (X x x) \}
\]
2. The elimination predicate is generated automatically by Coq (defined in terms of fix and case).

\[
\text{eqelim} : \Pi A : \text{Set}. \Pi x : A. \Pi P : A \rightarrow \text{Prop}. \\
(P x) \rightarrow \Pi y : A. ((\text{eq} x y) \rightarrow (P y))
\]

Example 2.5.20  Proof-objects showing reflexivity, symmetry, and transitivity of the equality relation.

\[
\begin{align*}
\text{eqrefl} & \coloneq (\text{constr}_1 \text{eq}) : \\
& \quad \Pi A : \text{Set}. \Pi x : A. (\text{eq} x x)\\
\text{eqsym} & \coloneq \lambda A : \text{Set}. \lambda x, y : A. \lambda H : (\text{eq} x y). \\
& \quad (\text{eqelim} A x (\lambda a : A. (\text{eq} a x)) (\text{eqintro} A x y H) : \\
& \quad \Pi A : \text{Set}. \Pi x, y : A. (\text{eq} x y) \rightarrow (\text{eq} y x)\\
\text{eqtrans} & \coloneq \lambda A : \text{Set}. \lambda x, y, z : A. \lambda H : (\text{eq} x y). \lambda H_0 : (\text{eq} y z). \\
& \quad (\text{eqintro} A y (\lambda a : A. (\text{eq} a x)) H z H_0) : \\
& \quad \Pi A : \text{Set}. \Pi x, y, z : A. (\text{eq} x y) \rightarrow (\text{eq} y z) \rightarrow (\text{eq} x z)
\end{align*}
\]

The above examples demonstrate the possibility of encoding powerful mathematical concepts in CIC. Of course, encoding mathematical theories in CIC is still a non-trivial task. However, the inductive types make direct encoding of, for example, algebraic structures much easier than the indirect impredicative encodings of CC. For examples, see the FTA project [40].

2.6 Theorem Proving in Coq

The notation used above to present CIC is geared towards easy presentation of the metatheory. In Chapter 3 and Chapter 5 we develop some theories in the Coq system, which are presented in Coq notation. The version of Coq that is used is V6.3.1.

The purpose of this section is to relate the notation used in the meta-theory of this chapter to the Coq notations in the other chapters. It also explains Tactics.

The reader is warned that this section does not constitute a Coq manual, we merely provide a way to relate concrete Coq notation (in this thesis presented in typewriter script with the CIC notation used in the formal definitions in previous sections of this chapter. See [11] for the Coq manual instead.

2.6.1 Coq Syntax

We provide a mapping of CIC primitives to Coq notation, starting with the CC primitives and the logical connectives. Although the mapping is specified through examples, we hope that the general idea is clear.
2.6. THEOREM PROVING IN COQ

Notation 2.6.1 Coq notation for \( \lambda \) - and \( \Pi \)-abstraction.

- \( \lambda x : A . B \) corresponds to \([x:A]B\)
- \( \Pi x : A . B \) corresponds to \((x:A)B\)
- \( A \rightarrow B \) corresponds to \(A->B\)
- \( \text{(not } A \text{)} \) corresponds to \(\neg A\)
- \( \text{(and } A B \text{)} \) corresponds to \(A\land B\)
- \( \text{(or } A B \text{)} \) corresponds to \(A\lor B\)
- \( \text{(ex } A P \text{)} \) corresponds to \((\text{EX } x : A \mid (P \ x))\)

Combined definition of inductive type and names for its constructors. When using the Coq theorem prover in practice, the \text{ind} construction can only be used in a definition. Such an inductive definition adds also the constructors of the type to the context.

Notation 2.6.2 Coq notation for inductive definitions. The CIC notation

\[
\begin{align*}
A & := \text{ind} \ X : \text{Set} \cdot \{F_1(X), F_2(X)\} \\
C_1 & := \text{constr}_1 A \\
C_2 & := \text{constr}_2 A
\end{align*}
\]

corresponds to the Coq notation

\[
\begin{align*}
\text{Inductive } A : \text{Set} := C_1 : F_1(A) | C_2 : F_2(A).
\end{align*}
\]

The Coq notation for \text{case} is very similar to the CIC notation. There exists however a much nicer notation for doing case analysis: The \text{Cases} macro. With \text{Cases} one can specify case distinctions using pattern matching like in functional languages. Internally everything is done in terms of \text{case}, but there is an intuitive operation semantics for \text{Cases}: The value corresponding to the first pattern that matches the term is the outcome of the \text{Cases} expression. From a presentation point of view \text{Cases} is preferable over \text{case}, which is why we use \text{Cases} as a primitive in Chapter 4.

Notation 2.6.3 Coq notation for case distinction. Suppose the following CIC notation for a \text{case} term is valid. I.e. \( A \) is an inductive type with three constructors, say \( C_1, C_2, \) and \( C_3. \) Let \( P_i \equiv (C_i \bar{x}) \) for \( i = 1, 2, 3. \)

\[
\langle Q \rangle \text{case } X : A. \begin{cases} 
\lambda \bar{x}_1 : \vec{B}_1.G_1 \\
\lambda \bar{x}_2 : \vec{B}_2.G_2 \\
\lambda \bar{x}_3 : \vec{B}_3.G_3 
\end{cases}
\]

corresponds to the Coq notation

\[
\langle Q \rangle \text{Cases } N : A \text{ of } \\
| \begin{array}{l} 
P_1 \Rightarrow G_1 \\
| P_2 \Rightarrow G_2 \\
| P_3 \Rightarrow G_3 
\end{array} \text{ end}
\]
If $Q$ is a non-dependent function type, a Coq user can just specify the codomain and leave out the abstracted dummy variables. In some cases, when Coq can derive $Q$, it may be left out completely. Even more complicated patterns are possible. However, this is all syntactic sugar, internally every `Cases` is translated in terms of `case` expressions. We do not specify how the `Cases` macro is translated to `case` internally.

With regard to fixpoints, there are three notations for `fix` in Coq. There are two inline notations: `fix/i f` (the index $i$ indicates which variable is the recursion variable) and `fix f[x:A]` (last variable in $x$ indicates which variable is the recursion variable). And there is one `Fixpoint` definition notation in which the fixpoint variable $f$ is added directly to the context.

**Notation 2.6.4** Coq notation for fixpoint terms and fixpoint definitions. The CIC notation

$$ f := \text{fix}_{3} f : B. \lambda x, y, z, p, q : A.M $$

may be denoted in Coq as

$$ \text{Fixpoint } f \ [x, y, z:A]: B := \ [p, q:A]M. $$

Other forms of Coq’s syntactic sugar are: User specified grammar rules, which we will make use of in this thesis, and implicit arguments, which are useful for leaving out arguments that can be computed. For example we would like to use the notation $(3) = (5)$ instead of $(\text{eq nat}(3)(5))$. The `nat` argument for the polymorphic equality relation can be derived by the type checker and may therefore be marked as an implicit argument, making the notation more pleasant for the reader.

### 2.6.2 Tactics

Creating a concrete proof-object, i.e. a $\lambda$-term, can be a tedious task. Although the Curry-Howard-De Bruijn isomorphism shows that this task is in principle equivalent to constructing a derivation, from a user’s perspective, constructing such an object directly requires too much detailed knowledge of the concrete formalism. Formalizing informal mathematics should be done on a level as abstractly as possible. Therefore many type theoretical theorem provers have facilities to construct proof-objects interactively with the user. During the interactive construction of a proof-object it contains holes which represent the subterms that still have to be constructed.

There are two possible solutions for this problem. In the first solution the user interface of the system is made in such a way that concrete proof-objects can be constructed interactively through a structure editor. The user selects one of the holes and chooses from a menu a head symbol for the corresponding subterm. The system restricts the choice of primitives based on the intended type of the subterm. Depending on the arity of the head symbol, a number of new holes appear. Initially there is one hole for the whole proof-object. This is implemented for example in the ALF proof-editor [67, 3] and its successor Alfa.
In the second solution a language of tactics is provided with constructs on a higher level of abstraction. A tactic script written in this language can be translated to the concrete logical object language. During the translation details are filled in by doing for instance non-trivial matching, tautology checking, and search in the (local) context. Many systems feature such a high level language based on tactics.

Coq implements the second solution through a high level language of tactics and tacticals. A formal definition of the concept of tactic as used in Coq is outside the scope of this thesis. The Coq tactics can be thought of as mappings which transform a term with holes into another term with holes, without changing the type of the term. From the perspective of the user, tactics operate on the current goal, changing it to zero or more subgoals. The notion of typeability of terms with holes is not well understood and is subject of active research. This is why Coq rechecks the type of generated concrete \( \lambda \)-terms after all holes are filled in. Another interesting question is: Which set of tactics makes up a good mathematical vernacular (see for example [46]). We will not go into these matters. Instead, we discuss some of the often used Coq tactics from the perspective of the user. This is done to give an impression of the level on which the tactics language operates, we do not intend to give a complete overview. See the Coq manual [11] for such an overview.

- **Intro** is used to push a \( \Pi \)-abstracted variable into the local context. For the proof-object under construction this means that the head symbol becomes a \( \lambda \).

- **Apply** is used to apply a lemma from the global context, or an assumption from the local context to the current goal. For the proof-object under construction this means that the head symbol becomes an application. **Exact** is a special version of **Apply** for cases where the lemma or assumption has exactly the current goal as type. For the CC fragment of CIC, **Intro** and **Apply** are sufficient to construct all possible proof-objects. Note, however, that the **Apply** tactic is more abstract than concrete CIC application, as **Apply** perform a form of matching.

- **Elim** is used to eliminate inductive types. When applied to a term of inductive type, it creates new subgoals replacing in the current goal all occurrences of that term with the possible values of the inductive type. For the proof-object under construction this means that the head symbol becomes an application of the elimination predicate (which is defined in terms of **Fix** and **Case**). **Case**, **Split**, **Left**, **Right**, and **Induction** are all elimination tactics similar to **Elim** which are used in special cases.

- **Simpl** and **Change** are used to invoke the conversion rule. The proof-object under construction does not change as a result.

- **Discriminate**, **Injection**, and **Inversion** are convenience tactics dealing with inductive types.
• Auto, Tauto, Assumption, Trivial, and Omega are some of the tactics that do automatic proof search.

2.7 Conclusions

In order to be able to reason about concrete formalizations of parts of mathematics, this chapter introduces a candidate universal language $\mathcal{L}$ which is based on typed lambda-calculus. The use of type theory in theorem proving shows some differences with the use of type theory in programming languages. In programming type theory is intended to catch mistakes made by the programmer. The programming language can be seen apart from the type theory. In the languages introduced in this chapter, however, the types are an integral part of the language, and help to define the language in more detail. We show that type theory has some of the properties we need in a universal language for mathematics: it is powerful, it conforms to the De Bruijn criterion and is thus trustworthy. We restrict ourselves to type theoretical theorem provers which use the ‘propositions as types’ correspondence to encode mathematical content.

The Calculus of Constructions is a first attempt. Since it is a system from the $\lambda$-cube and a pure type system, we can use many of meta-theoretical results. The CC is very powerful and yet very simple, it naturally adheres to the De Bruijn criterion. Inductive datastructures are possible through an impredicative encoding. However, when doing concrete computations, this is very slow; although the best alternative with respect to the De Bruijn criterion.

Inductive types can be added to CC, which leads to the Calculus of Inductive Constructions. This is the system underlying the Coq theorem prover. There is a trade off between efficiency and user friendliness on the one hand and conciseness and trustworthiness on the other. Adding new primitives with new reduction rules makes the conversion behavior much more efficient. This will be useful in Chapters 3 and 5. However, in order to allow the user to define only terminating functions in a natural way, the side conditions in the type system get very complex. By natural we mean that recursive functions are specified using pattern matching and recursive calls in the body of the function definition, rather than using explicit recursors. We list the side conditions as used in Coq.

The calculus of inductive constructions is quite powerful. Using examples we suggested that many mathematical notions can be constructed in the calculus. In Chapter 5 some more formalization examples can be found.

Proof-objects are first class citizens. They are easily accessible when we want to export proofs to other systems, or when we want to present proofs. We make use of this in Chapter 4.

Finally, we provided concrete Coq notations and related them to the notations we used in describing CIC. Throughout the rest of the thesis we will use the Coq notation introduced in this chapter.
Chapter 3
The Reflection Principle

3.1 Introduction

Computations are important in real life and even more so in computer science and mathematics. Contrary to popular belief, performing computations, even symbolic ones, is a bureaucratic activity that requires little or no intelligence at all. Hence we are able to construct computing machinery that can perform computations and save us a lot of work. We already introduced the class of CA systems in Chapter 1. They are systems optimized for doing efficient symbolic manipulations on formal mathematical expressions. We also looked at theorem provers, systems that are geared towards mathematical reasoning, rather than for computations.

The question we attempt to answer in this chapter is: How do computations relate to proofs? A question that is connected to this is: How do theorem provers deal with computations? Often, the activity of proving is very similar to performing computations. This is reflected for example in the Curry-Howard-De Bruijn isomorphism.

Provability, the problem of finding proofs, is difficult (in general undecidable), while checking a proof is an easy computation (see Chapter 2). Some provability problems are decidable, which means one can construct an algorithm to find the proof. Based on this insight our idea is to replace a proof obligation by an algorithm and a correctness proof of the algorithm. What we want is to automatically prove statements from classes where the provability problem is decidable through an algorithm, but we do not want to make the theorem prover more complex so that it stays compliant to the De Bruijn criterion.

This chapter is based on earlier work in [73]. The chapter is organized as follows: Section 3.2 lists different options for dealing with computations in theorem provers. Section 3.3 introduces the reflection principle, an internal method for provability automation through internal computations. Sections 3.4 and 3.5 describe two examples of the reflection method implemented in the Coq system. Section 3.6 compares our internal method to external methods to do automatic proving. Section 3.7 lists some results of the case studies and Section 3.8 presents the conclusions.
3.2 Computations in Proofs

Some theorem provers, especially the type theory based ones, have computations built in. That is, these provers contain a small functional programming language, often just typed $\lambda$-calculus. Moreover, a computation via $\beta\iota$-reduction in TT is for free, in the sense that given a program $P$ and input $d$, if $(P \, d)$ evaluates to $v$, then the statement $(P \, d) = v$ requires no proof. This idea is called Poincaré’s Principle and is embodied in CIC through the conversion rule.

In Chapter 2 we encountered the conversion rule and mentioned that it allows computations in the object language CIC. In the current chapter we investigate how computations relate to the process of formalization of mathematical theories, and specifically to the process of proving tautologies.

Below we introduce the notions of external and internal computations. As an example look at the less than or equal to inequality. We give three definitions of this relation on the natural numbers, which are (in Coq) provably equivalent. However, computationally they behave quite differently. The three definitions illustrate three different ways to do computations in TP: External, Internal, and Oracle computations.

3.2.1 External Computations

The first option for doing computations in a TP is the external method. Here we let an external program control the TP by generating either a proof-object or a tactic script which generates a proof-object. For example, the tacticals language can be used for this task. Even though the tactic language is part of the TP, we still call this external computing.

Consider as an example how in Coq the less or equal relation $\leq$ is defined with an inductive type.

```coq
Inductive le [n:nat] : nat->Prop :=
    le_n : (le n n)
| le_S : (m:nat)(le n m)->(le n (S m))
```

This means that a proof of a concrete inequality, say $3 \leq 5$, is constructed from the constructors $\text{le}_n$ and $\text{le}_S$. By applying $\text{le}_S$ two times, the goal changes to $3 \leq 3$ which is proved by one application of $\text{le}_n$. An automatic proof procedure can be specified for example in the tactical language of Coq:

```coq
Repeat (Apply le_n Orelse Apply le_S).
```

This tactical will solve any concrete inequality of this form. A drawback is that the generated proof-object gets rather large if the numbers in the goal are far apart. Most TPs use the external computation model. The advantage of this method is that the external system may be specified in an arbitrary programming language. A disadvantage is that the resulting proof-objects grow larger as the computations get larger.
3.2.2 Internal Computations

The second option for doing computations in a TP is the *internal method*. Here we use the built-in programming language that is present in some TP. For example, in Coq we can use the CIC to specify recursive functions.

This option is illustrated by giving an alternative, computational, definition of the less or equal relation.

```coq
Fixpoint le [n,m:nat]: Prop :=
  Cases n m of
    0 y => True
  | (S x) 0 => False
  | (S x) (S y) => (le x y)
end.
```

In a way, this definition looks like the inductive one above, however it actually computes the witness \( k \) itself. This means that a proof of \( 3 \leq 5 \) is trivially \( \text{Id} \), as the term \( 3 \leq 5 \) itself is convertible to \( \text{True} \). Therefore, the tactics that prove our goal are:

```
Exact I.
```

Note that the resulting proof-object (\( I \)) does not contain a trace of the computation. This is due to the conversion rule. Disadvantage of this method is that the built-in programming language of the TP is usually very limited in features, compared to general purpose external programming languages.

In some TPs one can only use \( \lambda \)-calculus to specify the computations. In Coq the `Fixpoint` and `Cases` constructions really help to specify computations on a higher level of abstraction. However, compared to the external method, where any programming language is allowed, the internal programming language of Coq seems rather limited.

3.2.3 Oracle Computations

The third option for doing computations in a TP is the *oracle method*. This is really a combination of the internal and external methods. Hopefully this yields the advantages of both methods, while eliminating the disadvantages.

Again we look at the less or equal relation. Another possible way, perhaps the most obvious one, to define this relation on the natural numbers, is to define it using an existential quantifier and the `plus` function. The `plus` function is defined using a fixpoint in Chapter 2.

```coq
Definition le := [n,m:nat](EX k:nat | (plus n k)=m).
```

When proving a concrete inequality, again we consider as example \( 3 \leq 5 \), a witness \( k = 2 \) has to be provided. Essentially, the proof-object consists of the witness. So, the tactics that prove our goal are:
The new goal is then to prove $3 + 2 = 5$, which is trivial since we have a proof that the equality relation is reflexive. Note that the conversion rule is used here, i.e. we are performing an internal computation to solve this goal. In principle $k$ could be found by an external oracle, and the proof is uniform in $k$, which constitutes another automatic proof procedure.

Summarizing, there are three options for computations: External computations, internal computations, and oracle computations. We focus on internal computations in the current chapter. External and oracle computations are discussed in Chapter 5.

### 3.3 Reflection

This method presented in this chapter automatically proves statements from certain decidable classes of propositions. Two examples are given: Propositional calculus and first order primitive recursive arithmetic. The methods we employ are fitted for type theoretical theorem provers as they rely on internal computations, and consist of replacing proof obligations by computations. For example, the proposition $\text{Prime}(61)$ can be verified by a computer program which checks all potential divisors of 61. By doing these computations, it can be seen that there are no proper divisors of 61. From this, it is concluded that 61 is prime.

In informal mathematical proofs, propositions like $\text{Prime}(61)$ are seldom proved. They are not considered to be “mathematically interesting” and verification is normally left to the reader. However, when constructing formal proofs using an automated proof system based on type theory, such as Coq, the user is forced to find proofs for all claimed propositions, including propositions like $\text{Prime}(61)$. The ability to prove these propositions automatically, allows users of these systems to concentrate on formalizing the important, mathematically interesting parts of a theory.

The method presented here is based on two main ideas. The first idea, goes by different names. It is called computational reflection in [48] dating back to original work by Howe in [53], who called it reflection. It is called two level approach in [13]. The idea is to interpret a class of propositions on three different levels: a syntactical level, a propositional level, and a computational level. The syntactical level makes it possible to relate the computational level to the propositional level by proving that a decision algorithm (on the computational level) indeed has the intended effect (on the propositional level). The second idea, called Poincaré’s principle in [8], states that propositions which can be verified by a computation are easy; i.e., no proof is required. This principle is incorporated in Coq through the so-called conversion rule: types that are computationally equal (convertible) are not distinguished. Poincaré’s principle is crucial for the use of computational reflection in theorem provers, as it allows to replace a large proof-object (laborious to generate externally) by a small proof-object plus a computation (mechanical).
In the second example, in Section 3.5, the combination of these two ideas allows us to replace a proposition from primitive recursive arithmetic (the propositional level) with a computation (the computational level) involving characteristic functions of primitive recursive predicates. The latter can be resolved using the conversion rule. Proving that this replacement is indeed allowed, involves lifting the original proposition to the syntactic level and translating it to the computational and propositional levels. It is proved that these two translations conform with each other: the translation to the computational level evaluates to true if and only if the translation to the propositional level is provable.

The results in this chapter show that it is possible to add powerful proof tactics to Coq and at the same time comply with the De Bruijn criterion as discussed in Chapter 1. The ‘reflection principle’ can be summarized as follows: The basic idea is to encode a specific syntactic class of formulas as an inductive type $\text{form}$. We write $[\cdot]$ for the decoding function, giving for every formula $a : \text{form}$ a proposition $[a]$. A given (powerful) proof procedure can (in the simplest case) then be defined as a function $F$ of type $\text{form} \rightarrow \text{form}$. Now, if we can prove this procedure to be correct inside Coq, i.e. if we prove $\forall a : \text{form} ([a] \leftrightarrow [Fa])$, then we can replace a proof obligation $[a]$ by a proof obligation $[Fa]$ (which will in general be easier).

### 3.4 Propositional Calculus

Certain propositions are too trivial to prove. In informal mathematics such propositions are never proved, even if a formal proof does not correspond to a single inference step. However, in the context of type theoretical proof assistants, we are forced to provide inhabitants for all statements. The method presented in this chapter provides proof-objects for these statements which can be read as “this statements belongs to a decidable theory, and therefore a real proof is omitted”.

In type theoretical proof assistants like Coq, mathematical statements are formalized as terms of type $\text{Prop}$. Proofs of such statements are then formalized as proof-objects with as type their proposition. The logical system which Coq implements is called the *Calculus of Inductive Constructions*. The objects are lambda-terms which are constructed using $\lambda$-abstraction, $\Pi$-abstraction, application of terms, inductive definitions, and primitive recursion. See Chapter 2.

One way to do proof automation is using the *external method*. External means that a mathematical statement $P$ is formalized as input for an external computer program which acts as a black box. The program yields as output a proposition $P$ and a proof-object $\text{prf}_P$. Coq can then verify that $\text{prf}_P$ inhabits $P$. An obvious drawback to the external method is that we do not know what the proof-object $\text{prf}_P$ looks like. In type theory, proofs do matter. One may desire that $\text{prf}_P$ corresponds to the trivial proof and not to some large machine generated proof.

Proof automation using the *internal method* (also called the *reflection principle* in [48] or the *two level approach* in [8]) involves proving only one theorem stating the correctness
of the decision procedure for all formulas of some fragment of the logical language is effectively decidable and then showing that the statement \( P \) by using the decision procedure is valid. More precisely, we encode a fragment of the meta language as an inductive type \( \text{form} \). We define an interpretation \( [\_\_] \) from \( \text{form} \) to \( \text{Prop} \) and a decision procedure \( (\_\_) \) from \( \text{form} \) to \( \text{bool} \) and we prove the correctness of \( (\_\_): \forall x:\text{form}. [x] \leftrightarrow (\llbracket x \rrbracket = \text{true}) \). We then have a proof-object \( \text{ok} \) inhabiting this correctness theorem. For the equality we could take Leibniz equality or the inductively defined equality of Coq, but in the examples in Sections 3.4 and 3.5 we use the recursively defined equality on the booleans.

The proof-object \( \text{ok} \) and the translations \( [\_\_] \) and \( (\_\_) \) are defined recursively over the inductive type \( \text{form} \). This means that proving \( (\_\_ P) \) is trivial (for true statements \( P \)) because \( (\_\_ P) \) will simply reduce to \( \text{true} \). So, the term \( (\_\_ P) = \text{true} \) reduces to \( \text{true} = \text{true} \) and any proof-object of the latter statement is also an inhabitant of the former. Combining this trivial proof with \( \text{ok} \) yields a proof of \( \llbracket P \rrbracket \) which is what we set out to find.

As an example of the internal method described in the previous section consider the language of propositional logic. Formulas of this language are built from propositional variables using different connectives as given in the following abstract syntax.

\[
\begin{align*}
\mathcal{B} & ::= A_0 \mid A_1 \mid \ldots \\
\mathcal{F} & ::= \mathcal{B} \mid \neg \mathcal{F} \mid \mathcal{F} \land \mathcal{F} \mid \mathcal{F} \lor \mathcal{F} \mid \mathcal{F} \rightarrow \mathcal{F}
\end{align*}
\]

We implement the abstract syntax in Coq using an inductive type \( \text{form} \). The propositional variable \( A_n \) is represented by \( \text{(f\_bas n)} \). For the connectives we have constructors \( \text{f\_not}, \text{f\_and}, \text{f\_or} \) and \( \text{f\_imp} \).

Inductive \text{form}: Set :=  
| \text{f\_bas}: \text{nat} \rightarrow \text{form} 
| \text{f\_not}: \text{form} \rightarrow \text{form} 
| \text{f\_and}: \text{form} \rightarrow \text{form} \rightarrow \text{form}
3.4. PROPOSITIONAL CALCULUS

\[ f_{\text{or}} : \text{form} \rightarrow \text{form} \rightarrow \text{form} \]
\[ f_{\text{imp}} : \text{form} \rightarrow \text{form} \rightarrow \text{form} \]
\[ f_{\text{all}} : (\text{nat} \rightarrow \text{form}) \rightarrow \text{form}. \]

The \textit{f_all} constructor can be viewed as a second order universal quantifier which binds a propositional variable. So actually the language described by \textit{form} is larger than the language of propositional logic, it is second order propositional logic. The \textit{f_all} constructor allows us to use Coq variables (of type \textit{nat}) instead of defining a type for propositional variables. For example, the propositional formula \( A \rightarrow A \) is represented by the term

Example 3.4.1

\[ \varphi = (f_{\text{all}} [a:\text{nat}] (f_{\text{imp}} (f_{\text{bas}} a) (f_{\text{bas}} a))) \]

Note that what we have here is really the universal closure of the first order proposition scheme: \( \forall A. A \rightarrow A \), rather than \( A \rightarrow A \).

3.4.1 Interpretations

Propositional Interpretation

The translation \([-]\) interprets syntactical formulas as propositions of type \textit{Prop}. It is called \textit{isVal} in the Coq code below. It is defined recursively over the \textit{form} type and has been given two extra arguments: a natural number \( l \) representing the number of bound variables the translation has encountered up till now and a valuation \( \sigma \) assigning to natural numbers values of type \textit{Prop}. For closed formulas the function is called initially with \( l = 0 \) and \( \sigma = \lambda x : \text{nat}. \text{False} \).

\[
\begin{align*}
\text{Fixpoint isVal \[p:form\]} & : \text{nat} \rightarrow (\text{nat} \rightarrow \text{Prop}) \rightarrow \text{Prop} := \\
[1:nat][s:nat->Prop]\quad \text{Cases } p \text{ of} \\
| (f_{\text{bas}} n) & \Rightarrow (s \ n) \\
| (f_{\text{not}} a) & \Rightarrow \sim (\text{isVal } a \ l \ s) \\
| (f_{\text{and}} a b) & \Rightarrow (\text{isVal } a \ l \ s) \ \& (\text{isVal } b \ l \ s) \\
| (f_{\text{or}} a b) & \Rightarrow (\text{isVal } a \ l \ s) \ \lor (\text{isVal } b \ l \ s) \\
| (f_{\text{imp}} a b) & \Rightarrow (\text{isVal } a \ l \ s) \rightarrow (\text{isVal } b \ l \ s) \\
| (f_{\text{all}} f) & \Rightarrow (a:\text{Prop}) \\
& \quad (\text{isVal } (f l) (S l) (\text{extendp } s l a))
\end{align*}
\]

end.

The intuition is to recursively replace syntactic connectives by the Coq versions of the connectives while replacing the propositional letters (represented by natural numbers) by fresh propositional variables (of type \textit{Prop}).

\[
[f_{\text{all}} f]_\sigma^l = \forall \alpha : \text{Prop}. [f l]_{\sigma[l\rightarrow \alpha]}^{l+1}
\]
The extendp function extends its argument $s$ such that the resulting valuation maps the natural number $l$ to the proposition $a$. So, whenever isVal encounters a $f\_all$ constructor, it replaces it by a “real” universal quantifier and alters the valuation $s$ in such a way that the recursive application of isVal will replace all corresponding $f\_bas$ occurrences with $a$.

\[
\text{Definition extendp: } (\text{nat} \rightarrow \text{Prop}) \rightarrow \text{nat} \rightarrow \text{Prop} \rightarrow (\text{nat} \rightarrow \text{Prop}) := \\
[s: \text{nat} \rightarrow \text{Prop}] [n: \text{nat}] [a: \text{Prop}] [m: \text{nat}] \\
\text{Case } (b\_eq n m) \text{ of } a \,(s \,(m)) \text{ end.}
\]

When applied to the formula $\varphi$ in Example 3.4.1, isVal yields $(a: \text{Prop}) a \rightarrow a$. This most likely is how one would have formalized $A \rightarrow A$ directly as a term of type Prop instead of via the form level.

**Boolean Interpretation**

The $([-])$ translation interprets syntactical formulas as booleans. It is called checkVal in the Coq code below. Like isVal it is defined recursively over form. This translation is constructed in such a way that it reduces to true or false by an internal computation. The main difference between $([-])$ and $[[-]]$ lies in the type of the valuation $s$ which now maps natural numbers to values of type bool. For closed formulas the function is called initially with $l = 0$ and $\sigma = \lambda x: \text{nat}. \text{false}$.

\[
\text{Fixpoint checkVal } [p: \text{form}]: \text{nat} \rightarrow (\text{nat} \rightarrow \text{bool}) \rightarrow \text{bool} := \\
[l: \text{nat}] [s: \text{nat} \rightarrow \text{bool}] \\
\text{Cases } p \text{ of } \\
\quad (f\_bas n) \Rightarrow (s \,(n)) \\
\quad (f\_not a) \Rightarrow (b\_not \,(\text{checkVal } a \,(l \,(s)))) \\
\quad (f\_and a b) \Rightarrow (b\_and \,(\text{checkVal } a \,(l \,(s))) \,(\text{checkVal } b \,(l \,(s)))) \\
\quad (f\_or a b) \Rightarrow (b\_or \,(\text{checkVal } a \,(l \,(s))) \,(\text{checkVal } b \,(l \,(s)))) \\
\quad (f\_imp a b) \Rightarrow (b\_imp \,(\text{checkVal } a \,(l \,(s))) \,(\text{checkVal } b \,(l \,(s)))) \\
\quad (f\_all f) \Rightarrow (b\_and \,(\text{checkVal } (f \,(l))) \,(S \,(l)) \,(\text{extendb } s \,(l) \,(\text{true}))) \,(\text{checkVal } (f \,(l))) \,(S \,(l)) \,(\text{extendb } s \,(l) \,(\text{false}))) \\
\end{cases}
\]

The intuition is to recursively replace syntactic connectives with boolean versions. The boolean versions of the connectives $b\_not, b\_and, b\_or,$ and $b\_imp$ are defined below by case distinction on their arguments.

The computational versions of the connectives are defined by:

**Definition 3.4.2** Boolean versions of the connectives as defined in Coq.

\[
\text{Definition } b\_not := [x: \text{bool}] \,(\text{if } x \text{ then false else true}). \\
\text{Definition } b\_and := [x, y: \text{bool}] \,(\text{if } x \text{ then y else false}). \\
\text{Definition } b\_or := [x, y: \text{bool}] \,(\text{if } x \text{ then true else y}). \\
\text{Definition } b\_imp := [x, y: \text{bool}] \,(\text{if } x \text{ then y else true}).
\]
The universal quantifier \( \text{\texttt{f\_all}} \) is replaced by a boolean conjunction of recursive applications of \( \text{\texttt{checkVal}} \). In the conjuncts the valuation \( s \) is extended such that all occurrences of the propositional letter (represented by a natural number) bound by the quantifier are replaced by \texttt{true} in one branch and by \texttt{false} in the other branch. A universal quantifier is replaced by a conjunction where each of the two conjuncts is the body of the quantified formula but with a different valuation. The valuations differ the boolean value that is assigned to \( l \).

\[
\text{\texttt{(f\_all f)}}_l^\sigma = (\text{\texttt{f l}})_l^{\sigma_{l→\texttt{true}}} \land_b (\text{\texttt{f l}})_l^{\sigma_{l→\texttt{false}}}
\]

Just like \( \text{\texttt{extendp}} \), \( \text{\texttt{extendb}} \) is implemented using a \( \lambda \)-abstraction and a boolean valued equality test on the natural numbers \( b\_eq \).

\[
\text{\texttt{Definition extendb: (nat->bool)->nat->bool->(nat->bool) :=}}
\]
\[
\lambda[s:nat->bool][n:nat][b:bool][m:nat] \quad \text{Case (b\_eq n m) of b (s m) end.}
\]

The \( \{\cdot\} \) function gives us an effective decision procedure for classical propositional calculus. When we apply \( \text{\texttt{checkVal}} \) to the syntactical formula \( \varphi \) in Example 3.4.1 it simply returns \texttt{true}.

### 3.4.2 Proof-objects

The correctness of the \( \text{\texttt{checkVal}} \) interpretation with respect to the \( \text{\texttt{isVal}} \) interpretation can now be stated. Under the assumption \( (a:\text{\texttt{Prop}}) a\lor \neg a \) we can prove the following theorem in Coq.

\[
\text{\texttt{Theorem ok: (p:form)(s:nat->bool)(t:nat->Prop)(l:nat)}
\]
\[
((n:nat) (s n)=true <-> (t n)) ->
\]
\[
(\text{\texttt{checkVal p l s}})=true <-> (\text{\texttt{isVal p l t}}).
\]

That is, for any formula \( p \) and for all valuations \( s \) and \( t \), where \( s \) maps natural numbers to \( \text{\texttt{Prop}} \) and \( t \) maps natural numbers to \( \text{\texttt{bool}} \), if \( s \) and \( t \) agree on all propositional letters (especially the numbers of propositional variables occurring freely in \( p \)), then \( \text{\texttt{isVal}} \) and \( \text{\texttt{checkVal}} \) applied to \( p \) agree on the truth of \( p \) under these valuations. The proof is by induction on \( \text{\texttt{form}} \). The classical assumption is needed in the \( \text{\texttt{f\_all}} \) case but only in the “only if” part.

We can now find a proof-object for the statement \( A \rightarrow A \) from Example 3.4.1. Take \( l = 0 \), \( s = \lambda x: \text{\texttt{nat}}.\text{\texttt{false}} \), and \( t = \lambda x: \text{\texttt{nat}}.\text{\texttt{False}} \), then it is easy to find a proof-object \( H \) for the statement \( (s n)=true \leftrightarrow (t n) \). Since \( (\text{\texttt{checkVal}} \varphi ls) \) reduces to \texttt{true}, the expression \( (\text{\texttt{checkVal}} \varphi ls)\text{\texttt{true}} \) reduces to \texttt{true} which has a trivial proof-object, say \( (\text{\texttt{refl\_equal bool true}}) \). Now clearly \( (\text{\texttt{Proj1 (ok \varphi stl H)}) I} \) is a proof for the proposition \( (\text{\texttt{isVal}} \varphi lt) \) which is convertible to \( (a:\text{\texttt{Prop}}) a\rightarrow a \).
3.5 Primitive Recursive Arithmetic

Primitive recursive arithmetic (PRA) can be seen as a language of formulas. Formulas from this language are either basic formulas or compound formulas.

Basic formulas are built using the relations $<$, $=$, and $>$, from arithmetical terms. Arithmetical terms are either natural number constants, or number variables, or the result of applying a primitive recursive function to other arithmetical terms.

Compound formulas are built using connectives or using bounded quantifiers. Connectives are $\neg$, $\land$, $\lor$, and $\rightarrow$. Quantifiers are restricted to bounded first order quantifiers, which are $\forall_<$ and $\exists_<=$. These bind a variable of natural numbers domain. The upper bound is an arithmetical term.

The divides relation and primality predicate can be expressed in this language.

Example 3.5.1 The division and primality predicates are primitive recursive.

\[
\text{Divides}(n, m) = \exists k < m + 1 \, [k \cdot n = m]
\]
\[
\text{Prime}(n) = \forall d < n \, [\text{Divides}(d, n) \rightarrow d = 1] \land n > 1
\]

Note that the unbounded definitions of these predicates, as given in Section 2.4.18, can be proven to be equivalent to the above definitions. This has to be done manually. Note that (syntactic) primitive recursive functions are part of the term language.

The language of primitive recursive arithmetic is formalized in Coq as the inductive type $\text{form}$. Notice that the terms from which basic formulas are built are just objects of type $\text{nat}$. It is not necessary to treat these terms syntactically, since both $[\_]$ and $\langle \_ \rangle$ will translate them in a similar way. Note that the choice of not treating terms syntactically has a consequence: the formulas (the $\sigma$ of type $\text{form}$) are not really from PRA, but an extension thereof, namely where the base terms are the terms of type $\text{nat}$ in Coq (instead of the terms generated from $\mathbb{N}$ by just application of primitive recursive functions). This slight extension of PRA is more convenient to formalize, as it removes the extra syntactic level of terms. Notice also the use of higher order function types in the type of the quantifier constructors $\text{f\_all}$ and $\text{f\_ex}$. This allows binding of variables using the object level $\lambda$-abstraction.

Definition 3.5.2 The language of primitive recursive arithmetic as formalized in Coq as the inductive type $\text{form}$.

\[
\text{Inductive form: Set :=}
\]
\[
\text{f\_lt: nat -> nat -> form}
\]
\[
\text{f\_le: nat -> nat -> form}
\]
\[
\text{f\_eq: nat -> nat -> form}
\]
\[
\text{f\_ge: nat -> nat -> form}
\]
\[
\text{f\_gt: nat -> nat -> form}
\]
\[
\text{f\_not: form -> form}
\]
\[
\text{f\_and: form -> form -> form}
\]
The automatically generated induction principle \texttt{form\_ind} has the following type.

\[
\forall P : \text{form} \rightarrow \text{Prop}. \\
\forall \varphi : \text{form}. (P \varphi \rightarrow (P (\varphi \text{\_not}))) \rightarrow \\
\forall \varphi : \text{form}. (P \varphi \rightarrow \forall \psi : \text{form}. (P \psi \rightarrow (P (\varphi \text{\_and} \psi))) \rightarrow \\
\forall \varphi : \text{form}. (P \varphi \rightarrow \forall \psi : \text{form}. (P \psi \rightarrow (P (\varphi \text{\_or} \psi))) \rightarrow \\
\forall \varphi : \text{form}. (P \varphi \rightarrow \forall \psi : \text{form}. (P \psi \rightarrow (P (\varphi \text{\_imp} \psi))) \rightarrow \\
\forall \varphi : \text{form}. (P \varphi \rightarrow \forall \Phi : \text{nat} \rightarrow \text{form}. (\forall m : \text{nat}. (P (\Phi m))) \rightarrow (P (\varphi \text{\_all} \Phi))) \rightarrow \\
\forall \varphi : \text{form}. (P \varphi \rightarrow \forall \Phi : \text{nat} \rightarrow \text{form}. (\forall m : \text{nat}. (P (\Phi m))) \rightarrow (P (\varphi \text{\_ex} \Phi))) \rightarrow \\
\forall \varphi : \text{form}. (P \varphi)
\]

The predicates from Example 3.5.1 can now be expressed as functions with codomain \texttt{form}.

\textbf{Example 3.5.3} The division and primality predicates as primitive recursive Coq predicates with codomain \texttt{form}.

Definition \texttt{f\_Divides}: \texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{form} := 
\[
[n,m: \text{nat}] (f_{\text{ex}} \ (S \ m) \ [k: \text{nat}](f_{\text{eq}} \ (\text{mult} \ k \ n) \ m)).
\]

Definition \texttt{f\_Prime}: \texttt{nat} \rightarrow \texttt{form} :=
\[
[n: \text{nat}]
(f_{\text{and}} \ (f_{\text{gt}} \ n \ (1))
(f_{\text{all}} \ n \ [d: \text{nat}](f_{\text{imp}} \ (f_{\text{Divides}} \ d \ n) \ (f_{\text{eq}} \ d \ (1))))).
\]

\section{3.5. Interpretations}

Again, three interpretations are defined on the types \texttt{form}, \texttt{bool}, and \texttt{Prop}. First, \texttt{[\dash]} (isVal in Coq notation) maps terms of type \texttt{form} to terms of type \texttt{Prop}. Second, \texttt{[\dash]} (checkVal in Coq notation) maps terms of type \texttt{form} to terms of type \texttt{bool}. Third, \texttt{istrue} maps terms of type \texttt{bool} to terms of type \texttt{Prop}. The three translations are depicted in Figure 3.2.
Propositional Interpretation

The translation \([-]\) takes as input a formula \(p\) of type \(\text{form}\) and it produces a proposition of type \(\text{Prop}\). Because \(\text{form}\) is an inductive type, \([-]\) can be defined by recursion by specifying a translation for each of the \(\text{form}\)-constructors. In describing recursive functions we use the usual Coq notation introduced in Chapter 2.

Definition 3.5.4 The translation \([-]\) in pseudo Coq notation.

\[
\begin{align*}
\text{f\_lt } t_1 t_2 & = \lt t_1 t_2 \\
\text{f\_le } t_1 t_2 & = \leq t_1 t_2 \\
\text{f\_eq } t_1 t_2 & = t_1 = t_2 \\
\text{f\_ge } t_1 t_2 & = \geq t_1 t_2 \\
\text{f\_gt } t_1 t_2 & = gt t_1 t_2 \\
\text{f\_not } p & = \neg [p] \\
\text{f\_and } p q & = [p] \land [q] \\
\text{f\_or } p q & = [p] \lor [q] \\
\text{f\_imp } p q & = [p] \rightarrow [q] \\
\text{f\_all } t h & = (x: \text{nat})(\lt x t) \rightarrow [h x] \\
\text{f\_ex } t h & = (\text{EX } x: \text{nat})(\lt x t \lor [h x])
\end{align*}
\]

Boolean Interpretation

The translation \((-\)} takes as input a formula \(p\) of type \(\text{form}\) and it produces a boolean expression of type \(\text{bool}\). Because \(\text{form}\) is an inductive type, \((-\)} can be defined by specifying a translation for each of the \(\text{form}\)-constructors.
Definition 3.5.5 The translation \([-\)] in pseudo Coq notation.

\[
\begin{align*}
(f_{\text{lt}} t_1 t_2) &= b_{\text{lt}} t_1 t_2 \\
(f_{\text{le}} t_1 t_2) &= b_{\text{le}} t_1 t_2 \\
(f_{\text{eq}} t_1 t_2) &= b_{\text{eq}} t_1 t_2 \\
(f_{\text{ge}} t_1 t_2) &= b_{\text{ge}} t_1 t_2 \\
(f_{\text{gt}} t_1 t_2) &= b_{\text{gt}} t_1 t_2 \\
(f_{\text{not}} p) &= b_{\text{not}} (p) \\
(f_{\text{and}} p q) &= b_{\text{and}} (p) (q) \\
(f_{\text{or}} p q) &= b_{\text{or}} (p) (q) \\
(f_{\text{imp}} p q) &= b_{\text{imp}} (p) (q) \\
(f_{\text{all}} t h) &= b_{\text{all}} t \{ x: \text{nat} \} [h x] \\
(f_{\text{ex}} t h) &= b_{\text{ex}} t \{ x: \text{nat} \} [h x]
\end{align*}
\]

The boolean versions of the basic relations are defined recursively over \texttt{nat}. When applied to concrete numbers, these will simply reduce to \texttt{true} or \texttt{false}.

Definition 3.5.6 Boolean inequalities as formalized in Coq.

\[
\begin{align*}
\text{Fixpoint } b_{\text{le}} [n: \text{nat}]: \text{nat} -> \text{bool} := \\
&\text{Cases } n \text{ of} \\
&\quad 0 \Rightarrow [m: \text{nat}] \text{ true} \\
&\quad \mid (S x) \Rightarrow [m: \text{nat}] \text{ Cases } m \text{ of} \\
&\quad &\quad 0 \Rightarrow \text{false} \\
&\quad &\quad \mid (S y) \Rightarrow (b_{\text{le}} x y) \\
&\text{end.}
\end{align*}
\]

\[
\begin{align*}
\text{Definition } b_{\text{lt}} := [n,m: \text{nat}](b_{\text{le}} (S n) m). \\
\text{Definition } b_{\text{ge}} := [n,m: \text{nat}](b_{\text{le}} m n). \\
\text{Definition } b_{\text{gt}} := [n,m: \text{nat}](b_{\text{lt}} m n).
\end{align*}
\]

For equality we also construct a recursive function. Note that this equality \texttt{b_eq} is equivalent to the general \texttt{(eq bool)} equality as defined in Example 2.5.19 in Chapter 2. That is, we can prove in Coq that they imply each other. However, the computational behavior of \texttt{b_eq} is different. When applied to concrete terms, \texttt{b_eq} will simply reduce to \texttt{true} or \texttt{false}.

Definition 3.5.7 Decidable boolean equality on \texttt{nat} formalized in Coq.

\[
\begin{align*}
\text{Fixpoint } b_{\text{eq}} [n,m: \text{nat}]: \text{bool} := \\
&\text{Cases } n \text{ m of} \\
&\quad 0 \quad 0 \Rightarrow \text{true} \\
&\quad \mid 0 \quad (S y) \Rightarrow \text{false} \\
&\quad \mid (S x) \quad 0 \Rightarrow \text{false} \\
&\quad \mid (S x) \quad (S y) \Rightarrow (b_{\text{eq}} x y) \\
&\text{end.}
\end{align*}
\]
The computational version of the bounded universal quantifier is defined by translating it into a large conjunction. The computational version of the bounded existential quantifier is defined by translating it recursively into a large disjunction.

**Definition 3.5.8** Boolean version of the bounded universal quantifier as formalized in Coq.

```coq
Fixpoint b_all [b:nat]: (nat -> bool) -> bool :=
  [f:nat->bool]
  Cases b of
  O => true
  | (S m) => (b_and (f m) (b_all m f))
end.
```

**Definition 3.5.9** Boolean version of the bounded existential quantifier as formalized in Coq.

```coq
Fixpoint b_ex [b:nat]: (nat -> bool) -> bool :=
  [f:nat->bool]
  Cases b of
  O => false
  | (S m) => (b_or (f m) (b_ex m f))
end.
```

The translation `istrue`: `bool → Prop`

The translation `istrue` takes as input a boolean expression and lifts it to the propositional level:

**Definition 3.5.10** The translation `istrue` as formalized in Coq.

Definition `istrue := [x:bool](if x then True else False).`

### 3.5.2 Proof-objects

Given a formula `p` of type `form`, the objective is to construct a proof-object inhabiting `[p]`. This is done in two steps. First, it is shown that the diagram in Figure 3.2 commutes. Next, it is shown, using the conversion rule, that `(istrue ([p]))` is inhabited. The combination of these two steps yields the desired proof-object.

Using the induction principle generated by the inductive definition of `form`, we can construct a correctness proof `ok` of the translations.

`ok : ∀p: form.[p] ↔ (istrue ([p]))`

The proof-object `ok` shows that the diagram in Figure 3.2 commutes. In general only the implication from right to left is needed. However, in the proof of the correctness theorem the other direction is needed in some of the induction cases.
The translation \([\vdash -]\) is constructed in such a way, that for closed terms \(p\) of type \(\text{form}\) that represent a provable proposition, it is the case that

\[
([p]) \rightarrow_{\beta_i} \text{true}
\]

and therefore

\[
(\text{istrue } ([p])) \rightarrow_{\beta_i} \text{True}
\]

where \(\rightarrow_{\beta_i}\) is the Coq reduction relation. From the conversion rule, it now follows that any inhabitant of \(\text{True}\) is also an inhabitant of \((\text{istrue } ([p]))\). Clearly, \(\text{True}\) is inhabited by the unit term \(\mathbf{Id}\), and therefore \((\text{istrue } ([p]))\) is inhabited.

By combining the inhabitant of \((\text{istrue } ([p]))\) with \(\mathbf{ok}\), we get an inhabitant of \([p]\), which is what we were looking for.

Ideally, an inverse of \([\vdash -]\) would also be available. In that case the user could write down the goal as an expression \(\varphi\) of type \(\text{Prop}\) and have the system translate it to an expression \(p\) of type \(\text{form}\). This inverse translation cannot be expressed within the object language. Some programming in the implementation language of Coq would be required to implement this translation. An alternative would be to use the extensible grammar mechanism of Coq to make the syntactical level look the same as the propositional level.

### 3.6 Internal versus External Computations

This section compares proof automation based on the internal computation method (reflection style) with proof automation based on the external computation method (tacticals style).

In the tacticals style of proving, the user describes a general decision procedure for a certain class of propositions using tacticals. Tacticals combine tactics into proof procedures (new tactics). Advantages of the tactical style: It is a very general method that can save a lot of work (compared to the ad hoc style), especially when many ‘similar’ propositions have to be proven. The method yields a proof term that usually corresponds rather closely to the proof term that would have been found by using the ad hoc style. The decision algorithm is described on the meta level, which gives quite a lot of flexibility. However, this can also be a drawback, as the user will have to be able to program in the meta language (or in the tactical language if that is provided). A disadvantage is that it can be very slow: all the steps have to be executed in the proof assistant, which requires a lot of unification and type checking.

In the reflection style of proving a trivial proposition \(\varphi\) is not formalized directly as an expression of type \(\text{Prop}\). Rather, \(\varphi\) is formalized on the syntactical level as an expression \(p\) of a new type \(\text{form}\), where \(\text{form}\) characterizes the class of propositions we are trying to deal with automatically. Translations, \([\vdash -]\) from \(\text{form}\) to \(\text{Prop}\) and \(([-])\) from \(\text{form}\) to \(\text{bool}\), are used to get formalizations of \(p\) on the other two levels, such that \([p] = \varphi\).
CHAPTER 3. THE REFLECTION PRINCIPLE

Table 3.1: Comparing internal- and external computations.

<table>
<thead>
<tr>
<th></th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>proof-object</td>
<td>small</td>
<td>large</td>
</tr>
<tr>
<td>checking time</td>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>language</td>
<td>Coq</td>
<td>ML, Tactics, other</td>
</tr>
</tbody>
</table>

These translations, as well as a translation from \texttt{bool} to \texttt{Prop} are defined in subsection 3.5.1.

Eventually, what we are looking for is a proof-object inhabiting $\phi$. This proof-object is constructed by combining two proof-objects. First, the proof-object \texttt{ok} inhabits the correctness theorem, which states that for all terms $q$ of type \texttt{form}: $[[q]]$ holds, if and only if $(\texttt{istrue}([[q]]))$ holds. Second, an inhabitant of $(\texttt{istrue}([[p]]))$ is sought for. This is easy: The boolean expression $([[p]])$ reduces to \texttt{true} (and then $(\texttt{istrue true})$ is inhabited) or it reduces to \texttt{false} (and then $(\texttt{istrue false})$ is not inhabited). The construction of these proof-objects is presented in subsection 3.5.2.

Table 3.1 summarizes the properties of the internal and external computation methods. Note that, in the internal strategy, the “user program” has to be specified using recursive function in the Coq object language. This is essentially typed $\lambda$-calculus. Although the \texttt{case} and \texttt{fix} constructions make this language look like a modern functional programming language, it is still a poor programming language compared to real programming languages, especially when we want to do efficient algorithms.

Advantages of the reflection method are: The proof-object itself is trivial and has constant length; as a matter of fact almost all of the ‘proof’ is in the computation, which is hidden in the type checking algorithm. That the proof-object is trivial conforms with the idea that proofs by computation are trivial and that computations should not contribute to the proof-object. Furthermore, reflection is a very general method, it solves a class of problems instead of one problem. A disadvantage of the reflection method is that it can be very slow due to the generality of the method. The generated decision algorithms follow a general (non-optimized) pattern. For example the algorithm for checking primality is a characteristic function that is generically extracted from the definition of \texttt{Prime}. This is less efficient then, for instance the special primality algorithm used in Chapter 5. One should be aware, however, that a generic method for solving a large class of propositions will always be slow, compared to ad hoc clever tricks. Another disadvantage is that the user needs to syntactically characterize the class of propositions and provide the translations and the correctness proof.

3.7 Results

The language of primitive recursive arithmetic can be elegantly formalized in the Coq system using inductive types. As a matter of fact, the inductive type \texttt{form} contains a bit
more than the formulas of PRA, namely the ones where we take the terms of type nat in Coq as base terms. The formalization is used to automatically prove propositions of primitive recursive arithmetic.

Although primitive recursive arithmetic is a limited language, many trivial propositions that are tedious to prove by hand can be expressed in it. By having the Coq proof assistant prove these automatically, the user can concentrate on the real, important, and mathematically interesting problems. We believe that the methods discussed in this chapter contribute to the user-friendliness of systems like Coq. It is possible to extend the method to include other predicates and functions on nat (or even other logical connectives). Suppose we have a relation \( R \) typeable in Coq, so \( R : \text{nat}^n \rightarrow \text{Prop} \). Moreover suppose that \( R \) is computable in Coq, so there is a term \( f_r : \text{nat}^n \rightarrow \text{bool} \) that computes \( R \). Then we can extend our method to include \( R \) as a predicate by adding a constructor \( r : \text{nat}^n \rightarrow \text{form} \) in the definition of \( \text{form} \) and by constructing a term \( q \) such that

\[
q : \Pi \vec{x} : \text{nat}^n. R \vec{x} \leftrightarrow \text{istrue}(f_r \vec{x}).
\]

The proof term \( q \) states (in Coq) that \( R \) is computable by \( f_r \); it is used in the construction of the new proof term \( \text{ok} \) for this extension of \( \text{form} \). We can depict the situation as follows. Put differently, the end result of the described strategy is a proof generator \( G \), which

![Diagram](image)

Figure 3.3: Extension of the method with computable predicate \( R \).

has the following property for the tautologies \( A \) from a certain class of propositions:

\[
\Gamma \vdash (G' A') : \llbracket A' \rrbracket
\]

As to the efficiency of the procedure: The procedure described here is not very fast. To check \((\text{Prime} \, 61)\) takes several minutes on a fast Unix workstation. See Figure 3.4 for some benchmarks.\(^1\) There are three reasons why this method is slow. First, the addition and multiplication functions operate on the standard unary numbers (generated by the constructors \( 0 \) and \( S \)). Things would be faster had we used binary versions of these functions on the computational level [55]. However, the correctness proof will become

\(^1\)Tests were done with Coq version 6.3.1 (native code version) on a Sun Ultra 10 with 333Mhz Sparc processor and 128MB memory.
more complicated if on the propositional level the same definitions of addition and multiplication are used. The use of these inefficient definitions is desirable because a lot of theory development depends on the unary defined natural numbers. The second reason is that computations are interpreted in Coq which in turn is interpreted in a functional language. This is not the most efficient setting for large computations. Third, the procedure is very general, meaning that it cannot take into account clever tricks to avoid computations. This results in slow algorithms. For example to check \((\text{Prime}\ 61)\) all numbers between 1 and 61 are tested as divisors of 61 instead of only the numbers up to \(\sqrt{61}\). In Chapter 5 a method is presented which greatly improves on this, however this method uses a combination of external and internal computations.

The method of computational reflection is not new, [48] gives an overview and history of reflection and contrasts it with the LCF (tacticals) approach. (We have briefly contrasted the reflection method with other approaches in Section 3.6.) In the Nuprl theorem prover [31] a reflection mechanism and a library with many different applications is implemented [53]. In [19] computational reflection is applied in Coq (the “Rings” tactic) to first order theories of algebraic structures such as monoids and rings. In [41] this is extended to fields with what is called the “Rational” tactic. In [13] an application of the reflection principle to decide equational theories is studied.

In [71] a similar technique as the reflection method from this Chapter is used to generate proofs for statements of PRA; there are however differences with the internal method described in this chapter. The method in [71] uses an external computer program. This program takes as input a string containing a formula \(\varphi\) of PRA and produces output which can be read by the Lego [66] proof system. The output produced in this
way contains the formula $\varphi$ of type Prop, a characteristic term $\chi_{\varphi}$ of type bool and Lego tactics which will construct a proof-object $\text{ok}_{\varphi}$ of type $\varphi \leftrightarrow (\text{istrue } \chi_{\varphi})$. The present method uses one correctness proof $\text{ok}$, which can be instantiated with a formula $\varphi$ of PRA by applying it to $\varphi$ since $\varphi$ is of type form which is now part of the object language.

Applying the method to other theories requires modifications to the type form as well as to the translations $[-]$ and $(\langle - \rangle)$ introduced in Section 3.5.1, and to the proof-object $\text{ok}$ from Section 3.5.2.

### 3.8 Conclusions

Computing forms an important mathematical activity. Often computations and reasoning are used in the same proof, and go hand in hand. Therefore a mathematical assistant, even one geared towards proofs, should facilitate computations. In principle, computations can be mimicked by reasoning steps. This is what happens when tacticals are used to implement decision procedures. But this is not acceptable as proof-objects get very large, and no longer reflect the reasoning of the proof.

The real problem is that TPs have a closed world assumption. One can only reason about the objects that are encoded in the context, not about external computations or algorithms. An extreme solution is to implement algorithms in the object language of the theorem prover. Poincaré’s principle, implemented in CIC through the conversion rule, states that convertible types have same inhabitants. Therefore the execution of an algorithm is not reflected in the proof-object. Efficient computations are possible due to inductive types, even though everything is done in object language.

Decision procedures require computations, so the idea arises to implement those too using internal computations? Unfortunately propositions are encoded as inhabitants of Prop. Algorithms specified in object language of the TP do not have “syntactical access” to inhabitants of type Prop. The solution discussed in this chapter is the Two-level approach, or Reflection.

The resulting proof-objects are small because of the conversion rule. However, for large and complicated computations the efficiency is bad. Because algorithms need to be terminating (and the TP has to be able to check this syntactically) it is hard to specify efficient algorithms, because all algorithms have to be specified according to strict syntactical rules.

The conclusion is that this technique should be used in special cases, where computations are needed to make boring and tedious work disappear. It should not be used to do real intensive computations. One example of a situation where reflection works well is “partial reflection” in [41]. Other examples are to be found in situations where the internal and external method can be combined. I.e. do the really hard computations outside of the theorem prover, and generate a computationally easy certificate which the TP can then verify. Examples of such a combined effort are given in Chapter 5.
Chapter 4

Interactive Mathematical Documents

4.1 Introduction

Since the advent of the World Wide Web, interactive documents have become increasingly popular. Users expect content to be presented to them interactively, allowing non-linear browsing of the document and embedding of non-textual objects, which allow exploration at multiple levels of detail. Problems arise in the creation of such interactive content. It seems as though the author has to take into account all possible future behavior of the reader. This would mean an explosion in content to prepare, which means an explosion in preparation time of such documents. How can one create interactive documents without doing the exponential amount of extra work? More specifically how does one create interactive mathematical documents?

One solution might be to convert existing non-interactive documents into interactive ones by exploiting the inherent structure present in the document. For example, the document you are currently reading is not interactive. Yet, it also is not flat text, the document is structured in sections and within each section are references to the other sections. In an interactive version of this document these references would be presented as hyper-links, enabling you, the reader, to jump to the referred-to section immediately. This form of interaction, internal interaction, requires that the author makes the structure explicit in the document. Sections have to be marked as sections, and references as references. However, adding structure like this is already common practice when documents, like this one, are prepared using a document preparation system such as \LaTeX. The reasons for providing the structure are usually not the possibility for interactive presentation, but maintainability of the document text. No changes in the numbering are needed when another section is added to the document. In preparing this document, the author created an input .tex file for \LaTeX in which sections are marked using the \texttt{\section} command, and each section contains a \texttt{\label} so that it can be referred to using the \texttt{\ref} command. In general all the author has to do, is to be careful not to mix content and presentation, for instance use \texttt{\section{Introduction}} rather than \texttt{\textbf{1 Introduction}}. If content and presentation are separated, an inter-
active version of the document is readily produced by appropriate conversion tools.

Now consider, instead of conventional textual documents, formal mathematical documents, like for example those in a library of a theorem prover assistant. Because they inherently contain much more explicit structure, there are many more opportunities for interactivity. For example, if we have a definition \( a := T \) in our document, then any future use of the definiens \( a \) can be seen as a link to the definieno \( T \). This piece of internal interaction does not have to be added afterwards, but is already present in the mathematical content. Proofs form another example where internal interaction may be derived from the mathematical content. If in a proof we apply another lemma, then this can be seen as a link to the statement (plus proof) of that lemma. Moreover, proofs have a lot of structure themselves (subcases, reasoning under a binder, hypothetical reasoning, etc.), which is left implicit in a textual presentation. The structure of proofs gives a way of hiding levels of detail of a proof (to be inspected on demand), thus providing a mechanism for folding and unfolding subproofs in the actual presentation. It is the formal mathematical content that dictates this folding and unfolding structure.

As with ordinary documents, the demands are put on the author of a mathematical document are that content and presentation are separated. This means that an interactive mathematical documents should not be treated as textual documents with here and there some interactive mathematical elements. Rather we see interactive mathematical documents as formal mathematical structures which can be presented textually through views allowing interactivity.

This chapter deals with the creation of such interactive mathematical documents based on formal mathematical content. For the purposes of this thesis we use formal mathematics as produced by the Coq theorem prover [11]. The type theoretic paradigm which forms the basis of Coq makes it a good candidate, as proofs are treated as first class citizens. However, the ideas also apply to other theorem provers, it is the mathematical structure which allows the interaction.

Internal interaction is interaction based on the internal structure of objects. Instead of a "rendered" embedded object, insert the "real thing", i.e. the formal object itself, in the document. It can then be rendered in different views within the browser.

![Figure 4.1: A view on a cube.](image)

To illustrate this principle, imagine we are creating an interactive document about the cube. In a paper document the best illustration of this three dimensional body is a drawing of the cube using perspective to suggest the third dimension, like the drawing in Figure 4.1.
In electronic documents we can do a better job. For example we can embed a small movie which shows a spinning cube, this gives a much better view of what an actual cube is. Still, to really convey the concept of cube to the reader, we want the reader to be able to view it from any angle. The correct way, and probably the easiest solution, to do this is to specify formally the model, for example by putting the coordinates in the document, and providing the user with a viewer which allows viewing of the model in three dimensions.

\[
E := \{ \{(0,0,0),(0,0,1)\}, \{(0,0,0),(0,1,0)\}, \{(0,0,0),(1,0,0)\}, \{(0,0,1),(0,1,1)\}, \\
\{(0,0,1),(1,0,1)\}, \{(0,1,0),(1,1,0)\}, \{(0,1,0),(0,1,1)\}, \{(1,0,0),(1,0,1)\}, \\
\{(1,0,0),(1,1,0)\}, \{(0,1,1),(1,1,1)\}, \{(1,0,1),(1,1,1)\}, \{(1,1,0),(1,1,1)\} \}
\]

This solution is more convenient and easier to maintain than the approach in which a .gif image or an .mpeg movie is embedded. The user can interact with the 3D model through a view that is present in the browser. Such a view could generate a 3D visualization of the formal cube like in Figure 4.1 but from an arbitrary angle and distance. If the formal model is specified in a standard language for 3D visualization such as VRML, an ordinary Web browser may be used to view it. In general our approach is to put the formal content in the document and then present it using specialized views in the browser. A drawback to this approach is that the view should be part of the browser which makes the browser more complex. Yet, solutions to this problem are emerging in the form of plug-ins, applets, XML, and stylesheets.

As mentioned above, one form of internal interaction we implement is the ability of the user to change the level of detail in which proofs are presented. This can be implemented by the following idea. Certain subterms of a proof-object represent subproofs. The type of such a subterm indicates what is proved in the sub-proof, and can be used as a short description of that sub-proof. This makes it possible to create interactive documents in which the reader can dynamically change the level of detail in which proofs are presented. This form of interaction demands that for every sub-proof also the type information is present. In principle the information can be computed at presentation time, but it is easier to do this when the document is created.

So far, we have mentioned internal interaction, a form of interaction based on the internal structure of the document. A different form of interaction, external interaction, allows the reader to send formal objects present in the document to back engines external to the documents. In our case we are interested in mathematical back engines such as theorem provers and computer algebra systems. The document acts as a user interface to such a back engine. The technology supporting communication to the back engines is readily available, although often in an experimental state. Combining internal and external interaction does not pose any problems. In fact, both make similar demands on the author: the document has to be structured. Examples of external interaction can be
found in Algebra Interactive [30], an interactive book on Algebra inviting students to try to solve example problems using a computer algebra system.

The OpenMath [74] language is designed to allow universal exchange of mathematical objects. It therefore facilitates external interaction very well by abstracting from the details of the back engine. For the internal interaction based on formal mathematical developments, an extension of OpenMath, called OMDoc [60], is used. In OMDoc entire mathematical documents are encoded instead of mere objects. Both OpenMath and OMDoc are XML applications, which means that a standard Web browser may be used to view documents expressed in OMDoc. How this is done is described in more detail in Section 4.7.

To provide a more abstract view on the proof-objects, we verbalize them as natural language (NL) text. This makes the resulting presentation look more like an informal mathematical document and provides a non-trivial view on the document. Generation of natural language proofs from formal logical derivations has been studied for a long time. For example in automated theorem proving, it is often hard to read the resulting proof trace, so verbalization is necessary. See for example the PROVERB system described in [54], and the Theorema system described in [20].

It is difficult to generate natural language text which really feels natural. The combination of generating NL text and internal interaction may lead to more natural documents, as the reader can adjust certain aspects of the presentation and guide the natural language generation process interactively.

The natural language generation employed in Section 4.5 is not the most sophisticated. It is based on algorithms described in [34, 33], which directly link fragments of natural language text to lambda terms. Yet, even though the mechanism is crude, it still yields some interesting results, especially in combination with internal interaction. Improvements to make the verbalization more natural are work in progress.

The CoqViewer tool [72] described in this chapter is an attempt to interactively present formal mathematical content generated in theorem provers. To be more specific, formalizations developed in the type theoretical theorem prover Coq [11] are used as input for the tool and views are created displaying the individual elements of the development. The views present the content in a way that resembles informal mathematics. The reader can interact with the views for example by changing the level of detail of the displayed proofs. The views can access the formal mathematical content so that in principle it can be exported to computational engines and be verified or manipulated.

Since the tool is still work in progress, what is described here is the core functionality needed for the presentation. Some elementary activities for changing the presentation are possible but the tool is not an editor for mathematical content yet. The implementation language is Java [45], which makes it easy to create graphical user interfaces and to reuse the code in other systems. This chapter is based on [72], [27].

The next section introduces some of the main ideas behind theorem proving systems based on type theory. We elaborate on the differences between the three mathematical languages used in type theoretical theorem proving: The tactics language used to com-
4.2 OVERVIEW OF THE SYSTEM

municate with such systems, the language of formal objects used inside the theorem prover engine, and the informal natural language used by mathematicians. Section 4.2 presents a high level overview of the architecture of the tool we are building. Sections 4.3, 4.4, and 4.6 provide more details on the design and implementation of the tool. Section 4.8 sums up the main results and draws some conclusions from them.

4.2 Overview of the System

This section gives a high level overview of the CoqViewer presentation tool. Figure 4.2 shows the architecture of the system. A Coq context is parsed, which results in an instance of the CoqTree datatype. This datatype is described in detail in Section 4.3. Next, the tree is annotated with type information using the algorithm in Section 4.4.8. Type inference requires many more operations on CoqTree. These operations are described in Section 4.4. Each node in the tree is annotated with the type of the term beneath that node. This type information is used by the views which present the CoqTree to the end user. Some of the views try to verbalize statements and proofs in the context. This is done by first translating the CoqTree versions of the proof-object to instances of the MetaText datatype. A MetaText consists of Statements. This datatype is described in Section 4.6.2. The verbalization to natural language is defined on this intermediate level so that the methods can be reused in multiple views. Currently there are two views available. The TreeView, see Figure 4.4, is the most basic view and shows the structure of the individual lambda terms. The NLView, see Figure 4.5, renders proofs as natural language proofs resembling Fitch style natural deduction proofs [38]. In this view assumptions are indicated by displaying them inside flags with the flag pole showing the scope of the assumption. Another possibility for displaying the content is exporting it to an OMDoc document [60], which can then be viewed in an appropriate browser. More details on the views can be found in Section 4.6.

Figure 4.2: Architecture of the system.
The creation and presentation of interactive mathematical documents using the Coq-Viewer, as we envision it, consists of three phases. First, during the formalization phase, a formal context is built using Coq. Next, after loading this context into the tool, during the authoring phase, the author can add presentation information to it. Currently this is done in the TreeView. Finally, during the presentation phase, the context is presented to the reader through one of the views. The reader can then interact with the document, by browsing through it and for example changing the level of detail in some of the proofs. If the reader has opened multiple views on the same object, the results from interaction are made visible consistently across all views.

4.2.1 Formalization Phase

During the formalization phase, the author creates a type theoretical context consisting of assumptions, definitions, axioms and theorems with proofs. Instead of providing concrete λ-terms, a higher level of tactics is used as mentioned in Chapter 2. Figure 4.3 shows a screenshot of a plain Coq session. The tactics language may seem quite close to informal mathematics but it is not the medium we want to use in interactive mathematical documents. Although the tactics language can be used to generate understandable natural language texts, see for instance [49], we choose to use the formal λ-term level as our input format for presentation. These formal objects, especially the ones representing proofs, may look less similar to the natural language like presentation we aim at, but it is easier to create internal interaction based on this formal content. This is mostly due to the fact that the high-level tactic language demands a complicated program, like Coq, to interpret it.

![Figure 4.3: Screenshot of Coq.](image)
4.2.2 Authoring Phase

During the authoring phase, the formal objects stored in the CoqTree are altered. However, the author is not allowed to change the mathematical structure of the formal objects, only the way they will be presented initially to the reader. By authoring we mean refining the presentation and not the mathematical content.

Adaptation of presentation is implemented by extending the CoqTree datatype (explained in Section 4.3) with attributes containing presentation information. Currently only the TreeView can be used to add presentation information to CoqTree objects. This view presents to the author a tree-like representation of the document, comparable to the folder tree a file browser provides. Figure 4.4 presents the TreeView. The panel on the left shows the context. The panel on the right shows the context item that is selected. This view is described in more detail in Section 4.6.1. In this view, the author is allowed to change the following information in each node:

- If the node is a variable (variables are the leaf nodes), the author can change its name. The variable name will be changed consistently throughout the document. All variables are bound, since the input is a complete theory development.

- Each node which is not a variable can be either collapsed or expanded. This will affect the way this node is presented in the views. It depends on the view how this is done, but in general collapsed nodes are displayed concisely, for instance the translation will not recursively translate the subtrees of a collapsed node, expanded nodes are translated verbosely.
• Nodes representing certain mathematical objects can have a preferred view. For example the author might assign the natural language view as default view for proof-objects.

More attributes might be added in the future. Perhaps even an extension mechanism so the author can add annotations to suit new view specific properties of nodes.

One might envision an editing environment based on the natural language view described in the next subsection. Currently the TreeView is the only view which allows this kind of authoring.

4.2.3 Presentation Phase

The reader is not allowed to change any attribute, formal or presentation, of the CoqTree objects in the context. He or she is only allowed to interact with the document. By interacting we mean changing the details of the current presentation only. Figure 4.5 presents the NLView. This view presents proof-objects as Fitch style natural language proofs. The panel on the left shows the context. The panel on the right shows the context item that is selected. This view is described in more detail in Section 4.6.2. Statements are presented in natural language, and the reader can change the level of detail of certain parts of the proof text. The datastructure allowing this form of interaction is described in Section 4.6.2. In order to generate this view, some involved computations are necessary, for example the type inference algorithm defined in Section 4.4.8 and the verbalization algorithms defined in Section 4.5.
4.3. TERMS

The views on mathematical content that the tool generates can also be exported as an OMDoc document and then presented to the reader using an OMDoc browser. For more details, see Section 4.6.

4.3 Terms

This section describes the datatype CoqTree and its implementation in Java. All Coq terms are encoded as CoqTree nodes. The parser generates a CoqTree containing as root a context node.

On the first level under the context node are the context item nodes. There are two possibilities for a context item node: An assumption node or a definition node. Both of these are described in Section 4.3.1. A special kind of definition node is the inductive definition node, which is described in Section 4.3.4. The context item nodes have CoqTree terms as subtrees describing mathematical objects, statements about mathematical objects, or proofs of those statements.

Even though, by the De Bruijn criterion, the set of nodes for terms can be limited to basic lambda calculus, the parser recognizes primitive nodes for many more notions, such as the logical connectives and the natural numbers. All of these notions can be defined in terms of lambda calculus, and this is how they are implemented in Coq. However, from a presentation point of view, since we want a presentation close to informal mathematics, it is a good idea to treat them as primitives.

A CoqTree consists of nodes connected by pointers. There are a number of different kinds of pointers. The foremost one, subtree, connects a parent node to the root nodes of its subtrees. Then there are the bindvar and bindsym pointers, which are used to indicate formal binding of variables. Furthermore, some temporary pointers copylink, alphalink, and typelink are needed during some of the operations described in Section 4.4.

Since the tool is implemented in the object oriented language Java, there are two distinct options for the representation of trees. The first option, in true object oriented style, is to define an abstract class CoqTree and define subclasses for every different kind of node. These classes are then organized in a hierarchy based on the inheritance relation, such that operations on similar nodes need only be specified once. For example, λ- and Π-nodes are both abstraction nodes and behave the same with respect to alpha-conversion, substitution, etc. The drawback of this option is that methods to manipulate the terms, such as the operations described in Section 4.4, are scattered throughout the different classes.

The second option, which is actually implemented, is to have one class for CoqTree of trees which has a field treekind indicating what kind of tree is represented. Most of the algorithms from Section 4.4 can now be specified using case distinction on this field. This style of coding is closer to the functional programming style.
4.3.1 Context and Context Item Nodes

What the parser gets from Coq is a context with definitions, assumptions, and theorems with proofs. One context node is created, which has as subtrees all definition and assumption nodes. In Figure 4.6 a context node with three context items is shown. The nodes labeled with BV are abstraction variables which are discussed in Section 4.3.3.

A definition node has three subtrees. The first one is its name (the definiendum). The second subtree is the actual value of the definition (the definiens). And the third subtree is its type. Assumption nodes are just like definition nodes except they do not have a definiens subtree. They only declare a name with a type. Inductive definition nodes are described in Section 4.3.4.

Once a definition or an assumption has been declared, its name may be used in the rest of the context. These names are treated like variables, and therefore the definition and assumption nodes are in fact abstraction nodes, described in Section 4.3.3. A definition node differs from ordinary abstraction nodes, such as lambda nodes, in that the scope of the bound name is not a subtree of the node, but all sibling trees in the context to the right of this definition node.

4.3.2 Basic Nodes

The simplest of all nodes is the variable node. It has no subtrees. Variable nodes do not have a name. Since all variables are bound, the name can be stored in the formal abstraction variable node, described in Section 4.3.3. The only relevant attribute of a variable is a bindvar pointer to this formal abstraction variable node.

Other basic nodes correspond to connectives like negation, conjunction, and application etc. These do have subtrees, but they introduce no other structure, see Figure 4.7.

4.3.3 Abstraction Nodes

An abstraction node is used to introduce a formal name that binds occurrences of this name in the subtrees of the node. Usually there is only one subtree of an abstraction
node where a bound variable can occur called the body, but there are exceptions. The variable nodes occurring in the body in the tree need to be able to indicate that they are bound by this node. For example in the body (the third subtree) of the $\lambda$-abstraction node in Figure 4.8 a variable node marked with $v$ is bound by the $\lambda$-node. The binding is implemented using a bindvar pointer, indicated by a dashed arrow in Figure 4.8.

To be more accurate, the variable occurring in the body of the lambda term is bound by the formal abstraction variable, indicated by $BV$. It was decided to not bind variables to the abstraction node itself because we encounter examples of abstraction nodes where multiple variables are bound at once in Section 4.3.5. This decision forces us to add yet another link: The bindsym pointer, indicated by a dotted arrow in the figure, provides a reference from the abstraction variable to the abstraction node. Certain operations on the tree use this bindsym reference to make decisions depending on the kind of abstraction used for some variable.

The second subtree contains the type of the abstraction variable and is called the domain.

Examples of abstraction nodes are $\lambda$ and $\Pi$-nodes, but also definition and inductive definition nodes and the match nodes occurring in the cases construct.
4.3.4 Inductive Definition Nodes

Inductively defined sets are often used in mathematics. For example the set of natural numbers can be defined as the smallest set which contains $0$ and which is closed under the successor operation $S$. In Coq this type is introduced with:

$$\text{Inductive nat: Set := O: nat | S: nat->nat}$$

Although all inductive definitions can be encoded in the calculus of Coq as impredicative types, inductive definitions were explicitly added to the calculus both for convenience and for efficiency reasons. See the discussion in Section 2.5. The extension with inductive types is done by introducing a new sort of definition called \textit{inductive definition}. Both inductive sets and inductive propositions may be defined.

![Figure 4.9: An inductive definition node with two constructors.](image)

An inductive definition introduces a new name and states a type for the object it defines. It also introduces a number of constructors. Each constructor introduces a new name with a type. The type of a constructor may contain a reference to the type we are defining. There are some restrictions on the position where this name may occur, ensuring that the type defined is well founded. A description of these restrictions is in Chapter 2, but the \textit{CoqViewer} does not check them since the context will be fully checked by Coq.

4.3.5 Cases and Match Nodes

The \textit{cases} construct was added to the language to allow case distinction on values of an inductive type. Every concrete object of an inductive type is constructed by repeated application of the type constructors. Cases can be used to determine which constructor was applied last. See Section 4.3.6 for some examples of expressions involving \textit{cases}. 

4.3. TERMS

A *cases* node, Figure 4.10, has as subtree the term on which the case distinction is applied and the type of the expression itself. In addition there are several *match nodes*, marked with ‘⇒’.

Each match node, Figure 4.11, contains a pattern in the left subtree and a body in the right subtree. The body contains variables which are bound by pattern variables in the pattern. The binding symbol for these pattern variables is the ‘⇒’ in the match node. During reduction, the term on which case distinction is applied is compared to each of the patterns using the matching algorithm described in Section 4.4.6. The matching algorithm returns a substitution for variables occurring in the corresponding body. The result of the reduction is this body after applying the substitution.

Internally Coq does not use the *cases* but rather the *case* primitive. See Chapter 2 for a discussion. However, for presentation purposes, we choose to treat *cases* as primitive because it explains the case distinction on a slightly higher level.

### 4.3.6 Fixpoint Nodes

Fixpoints are used to define recursive functions. In Coq a recursive function can only be specified over inductively defined types. The syntactic restrictions on formation of
inductive types and fixpoint terms guarantee termination of all recursive functions. A fixpoint introduces a temporary name, the \textit{fixpoint variable}, which may be used again in the body of the fixpoint construct.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fixpoint_node.png}
\caption{A fixpoint node with one recursion parameter.}
\end{figure}

A general recursion scheme like this would allow non-terminating functions, therefore some restriction is necessary. In Coq this is solved by demanding that a \textit{recursion parameter} (in Figure 4.12 one recursion parameter appears which is marked with RP) is mentioned explicitly. This is a variable whose value gets structurally smaller with every recursive application of the function. Early versions of Coq used a positive integer to indicate which variable plays the role of the recursion parameter. For example the definition of the addition function would be printed as:

\begin{verbatim}
plus = fix f/1: nat->nat->nat :=
{ [n,m:nat]
  <nat>Cases n of
  0    => m
  | (Sk) => (S (f k m))
  end
}
\end{verbatim}

The number 1 indicates that the first \(\lambda\)-abstracted variable in the body, \(n\), is the recursion variable. This is the same notation used in Chapter 2. In newer versions of Coq a slightly friendlier but equally powerful notation is used:

\begin{verbatim}
plus = fix f [n:nat]: nat->nat :=
{ [m:nat]
  <nat>Cases n of
  0    => m
  | (Sk) => (S (f k m))
  end
}
\end{verbatim}
The name of the recursion parameter \( n \) is mentioned explicitly in the type of the fixpoint variable. The square brackets in the type act as a \( \lambda \)-abstraction. If the recursion parameter is not the first variable, more variables need to be abstracted in this way. In the “display” following the fixpoint variable \( f \), always the last abstracted variable indicates the recursion variable.

The combination of fix and cases allows the Coq user to specify recursive functions using an intuitive-looking syntax. The recursion check (to ensure recursive calls are applied to a smaller term than the value of the recursion parameter) and the positive occurrence check (on the definition of the inductive type) ensure that we get a terminating function.

### 4.4 Operations

Several operations can be defined on the CoqTree datatype. Ultimately what is needed for the NLView, described in Section 4.6.2, is a type inference algorithm. Type inference requires operations such as reduction and copying of terms. Because of the representation of the terms described in Section 4.3 these operations may not be very standard anymore. This section describes some of the problems encountered in implementing them.

#### 4.4.1 Copy

For various purposes, one of which is type inference, it is useful to be able to make a copy of a term. The obvious way to copy a tree is to go top-down from the root to the leaves, copying all information in this node to a newly created one, and applying the copy method recursively to all subtrees. A problem arises here because we do not want to copy the binding links of bound variables literally, as this would bind the variables to abstraction nodes in the source tree. We need to bind those variables to the copy of this abstraction node in the destination tree.

![Figure 4.13: The copy method in action.](image-url)
The solution is easy. While copying top-down, copylinks are made, which are indicated by the dash-dotted arrow in Figure 4.13. These temporary links connect abstraction variable nodes, marked with BV in Figure 4.13, in the source tree to the corresponding abstraction variable nodes in the target tree. When a variable is encountered the copylink of the abstraction node in the source tree now points to the new abstraction node in the target tree.

4.4.2 Syntactical Equivalence

To test for syntactical equivalence of terms, the trees in question should be isomorphic. This operation suffers from the same problem as the copy operation. To determine whether two variables have the same abstraction node, it is not enough to compare the binding links. If the pointers are exactly the same, then the variables are alpha equivalent, but the two variables are also alpha equivalent if they have alpha equivalent abstraction variables.

![Figure 4.14: Testing for syntactical equivalence.](image)

The solution is similar to the one used in the copy situation: we introduce temporary alphalinks, indicated by the dash-dotted arrow in the figure, connecting abstraction nodes, marked with BV in Figure 4.14, in the first tree to abstraction nodes in the second tree. Now we can compare the abstraction node of a variable in the second tree with the alphalink of the abstraction node of a variable in the first tree.

4.4.3 Currying

Application in lambda calculus takes two arguments: a function and its argument. Functions of higher arity are specified through currying. Currying of a function is to define it in such a way that the result after one application is again a function which can be applied to another argument, for example instead of defining a function of type \( A \times B \rightarrow C \), we define a function of type \( A \rightarrow (B \rightarrow C) \). See Chapter 2.

For some of the operations, notably reduction of fix definitions, described in this section it is useful to be able to detect whether such a curried function is used, especially
since most of the operations in this section traverse the tree recursively and can only see the kind of tree for the current node. We have operations that detect application spines such as in Figure 4.15 and yield pointers to the function $F$ (which itself is not an application) and its arguments $\langle A_1, A_2, A_3 \rangle$, giving the intended meaning of this tree: $(F A_1 A_2 A_3)$. This is for instance used in the reduction method to collect all the recursion parameters of a fix application, to figure out whether it is a redex or not, see Section 4.4.5.

A related problem is detecting repeated $\lambda$- and $\Pi$-abstractions. We also have operations for this. These are primarily used for presentation purposes such as pretty-printing of expressions in the views described in Section 4.6.

### 4.4.4 Substitution

Since bound variables cannot be identified by their name (remember that the name of a variable is stored in the abstraction variable which binds the variable) we use a pointer to their binding abstraction variable to identify them. This node is passed to the substitution method as a parameter.

The substitution method just traverses the tree top-down from root to leaves, replacing every variable bound to this parameter by a copy of the tree we want to substitute in place for it. This is done by the copy method. This means that there is no sharing of subtrees.

### 4.4.5 Conversion

There are two reduction methods. One reduces the term to normal form and one reduces the term only to weak head normal form (WHNF). See Definition 2.5.6 for the definition of WHNF. Both methods traverse the term leftmost outermost, reducing reducible expressions, or redexes. The three different kinds of CIC redexes from Chapter 2 are considered: $\beta$, cases, and fix. Furthermore, if a variable node is encountered which
is bound by a definition node, a so-called \( \delta \)-redex, it is replaced by the corresponding definition.

\[
(\beta) \quad (\lambda x : A.M) N \rightarrow M[N/x]
\]

\[
\text{(cases)} \quad \begin{cases}
\text{cases (constr}\bar{A}\text{) of} \\
| P_1 \Rightarrow B_1 \\
\vdots \\
| P_n \Rightarrow B_n \\
\text{end}
\end{cases} \rightarrow B_i[\sigma]
\]

\[
\text{(fix)} \quad (\text{fix} \, f \, x \, T \, B) \, \bar{C} \,(\text{constr}\bar{A}) \rightarrow B[f := (\text{fix} \, f \, x \, T \, B)] \, \bar{C} \,(\text{constr}\bar{A})
\]

In the cases case, \( \sigma \) is the substitution produced by the match method and \( P_i \) is the pattern which matches with (constr\( \bar{A} \)). Section 4.4.6 explains matching of trees with patterns. In the fix case, \( f \) is the fixpoint variable and \( x \) is the recursion variable.

Weak head reduction only reduces the top level redex and does not continue to reduce the subterms. This is preferable if only the root symbol of a term is needed. Weak head reduction is also useful in the natural language generation algorithms in Section 4.6.2. By only reducing a statement to weak head normal form, it remains as abstract as possible.

### 4.4.6 Matching

The matching method is only used in the reduction of a cases redex. It matches terms against patterns that occur in the left hand side of the \( \Rightarrow \) nodes. It takes as input a pattern and a tree and returns whether the tree matches the pattern. If they match, then a substitution \( \sigma \) is returned.

A pattern is a tree that is either a new abstraction variable or a constructor (previously defined in an inductive type) or an application of patterns. Of course, only well-typed patterns are allowed. Here \( P \hat{=} M \) denotes the term \( M \) matches the pattern \( P \). The arrow on the left indicates that the rules form a priority rewrite system (notation from [5]). The rules should be tried from top to bottom, and the first rule that fits is used.

\[
\begin{align*}
\text{constr} \, c_1 & \hat{=} \text{constr} \, c_2 \quad & \text{if } c_1 \text{ and } c_2 \text{ are the same} \\
\text{bvar} \, x & \hat{=} T \quad & \text{for all trees } T, \, \sigma := \sigma \cup \{x \mapsto T\} \\
T_1 & \hat{=} T_2 \quad & \text{if root symbols are equal} \\
& & \text{and all subtrees match}
\end{align*}
\]

Applying the matching method to a pattern \( P \) and a term \( M \) yields a boolean, indicating whether \( M \) matches \( P \), and a substitution \( \sigma \) such that \( \sigma(P) \equiv M \).
4.4.7 Typing

Most typing algorithms for systems of typed lambda calculus are presented as inference rules, see for instance [7]. The concept of local context plays an important role in this style of presentation. Local contexts are used to keep track of free variables and their types. When the typing algorithm encounters an abstractor such as a \( \lambda \), it stores the variable with its type in the local context and continues with the body of the term.

In our system, however, all variables are bound through binding links to abstraction variables in the global context. Our typing algorithm does not make use of local contexts. Whenever the algorithm encounters a variable, it retrieves its type by following the binding link to the abstraction node and copying the associated type. For example, if the abstraction node is a \( \lambda \)-abstraction node, then the second subtree of this node represents the type of the variable.

The type of a term is built recursively. Based on the symbol of the current node, new nodes are created and the method continues recursively with the subtrees. The type algorithm below will, given a typeable term, reconstruct its type.

\[
\begin{align*}
\text{type}(\text{Set}) &= \text{type}(\text{Prop}) = \text{Type} \\
\text{type}(\lambda x : A.B) &= (\Pi x : A.\text{type}(B)) \\
\text{type}(FM) &= \text{if (\text{whnf}(\text{type}(F)) = (\Pi x : A.B))}
\text{then } B[x := M] \\
&\quad \text{else failure} \\
\text{type}(\text{var } x) &= \text{find the abstraction var of } x \\
&\quad \text{and determine its type} \\
\text{type}(A \rightarrow B) &= \text{type}(B) \\
\text{type}(\Pi x : A.B) &= \text{type}(B) \\
\text{type}(\text{ind } x T \bar{C}) &= T \\
\text{type}(\text{fix } f x T B) &= T
\end{align*}
\]

These rules are sufficient for well-typed CIC terms, but we also give rules for some of the frequently used “defined concepts” that are recognized by the parser and treated as “primitive concepts”. These concepts may occur in the parse trees. For instance, we have:

\[
\begin{align*}
\text{type}(A \times B) &= \text{Set} \\
\text{type}(A + B) &= \text{Set} \\
\text{type}(\neg A) &= \text{Prop} \\
\text{type}(A \land B) &= \text{Prop} \\
\text{type}(A \lor B) &= \text{Prop}
\end{align*}
\]

In the implementation we also have rules for the introduction constants (the constructors of inductively defined types) and elimination predicates.

The type inference algorithm makes some assumptions about the term to be typed. For example, in the application case it is assumed that \( M \) has type \( A \). The algorithm can be made more robust, so that it will always yield failure for terms that are not typeable.
As the terms are part of a context that is checked by Coq it can safely be assumed that the terms are typeable.

During type inference, some copying of subtrees is required. For example when typing a tree with a λ as root symbol, a new Π-node is created which also abstracts a variable. The domain of this new binding variable is an exact copy of the domain of the binding variable of the λ-node. A problem that arises is that the domain tree to be copied might contain variables with binding links which point to places outside of the destination tree.

![Figure 4.16: The type inference method in action.](image)

For example to infer the type of the double λ-abstraction in Figure 4.16, a double Π-abstraction is created. In the tree on the left, the variable occurring in the domain of the second λ-abstraction is bound by the first λ-abstraction. In the tree on the right, the corresponding variable needs to be bound by the first Π-abstraction. Therefore, the copy operation has to have knowledge about which λ-nodes in the source tree correspond to which Π-nodes in the destination tree.

To solve this problem, we introduce typelinks which connect the binding variables in the term to binding variables in the type. The copy algorithm uses these typelinks just like it uses copylinks. Note that it is necessary, in order to make a shallow copy of the domain of abstracted variables, that the copy method is aware of the typelinks.

### 4.4.8 Expected Type

The above type inference algorithm derives a type for a term based on the derived types of its subterms. A type for a term is derived top-down.

Types for terms are not unique. A term can be assigned different types, although they will always be equivalent with respect to the reduction relation described in Section 4.4.5. Especially δ-redexes, i.e. definitions, can be used to make types look more...
abstract. For example the proof-object proving reflexivity of Leibniz’ equality has derived type:

\[(x:A; P:A\rightarrow Prop; H: (P \ x)) (P \ x)\]

but the same term can also be assigned the more descriptive type:

\[(\text{isReflexive leibniz)}\]

A CIC term can have many different (albeit equivalent) types. The typing algorithm in Section 4.4.7 computes the derived type for a given term. In order to ensure the most abstract type for the subterms of a proof-object, the above algorithm is augmented to compute both the derived type and an expected type, similar to the algorithm described in [33]. It takes as input a term and an expected type for this term, and annotates the term and all of its subterms with expected type information. At the same time the node is also annotated with its derived type.

\[
\text{exptype}(\lambda x: A.M, \tau) = \text{Annotate current term with } \tau.
\]

\[
\text{Reduce } \tau \text{ to WHNF: } \tau = (\Pi x: A.B).
\]

\[
\text{Call exptype}(M, B).
\]

\[
\text{exptype}(FM, \tau) = \text{Annotate current term with } \tau.
\]

\[
\text{Let } \tau_F = \text{type}(F), \tau_M = \text{type}(M).
\]

\[
\text{Reduce } \tau_F \text{ to WHNF: } \tau_F = (\Pi x: A.B).
\]

\[
\text{Call exptype}(F, \tau_F).
\]

\[
\text{Call exptype}(M, \tau_M).
\]

\[
\text{exptype}(\text{var } x, \tau) = \text{Annotate current term with } \tau.
\]

After parsing, each subtree of the context tree is annotated with derived and expected type attributes. The expected type of a subterm is computed from the expected type of the parent node. This means that the algorithm needs to be initialized with an expected type of the context item. Fortunately, since all definitions and proofs occur in the context with a preferred type, such an initial expected type is always available.

### 4.5 Generation of Documents

This section describes how OMDoc documents are generated from Coq contexts. Our goal is to generate interactive mathematical documents based on Coq contexts. The documents should be readable as natural language mathematical documents, so some verbalization of the formal objects is needed. This demands that this informal description is present in the document.

Essentially, this means that content and presentation are stored separately in the document as suggested in [27]. The OMDoc language allows us to do this. Since OMDoc is an XML application, XSL stylesheets can be used to do the actual presentation. Below
we give an overview of the tool by which we generate the OMDoc document. In Figure 4.17 a detailed view of this process is given. Boxes with rounded corners represent Java classes in our implementation.

We start with a formal Coq context. Inside Coq this context is stored in some internal data structure; in Figure 4.17 this is represented by the box labeled with ‘Coq ADT’. We do not really have to know what the internal data structure is like. This context is parsed

![Diagram](image)

Figure 4.17: Detailed overview of the system.

and stored in our tool as an instance of the CoqTree class described in [72]. All terms themselves are also stored as objects of class CoqTree. The class has methods to do reduction, and type inference.

There are two ways to arrive at a CoqTree object starting from the Coq ADT. One way is printing the context with the Coq Print command, and parsing the ‘printed context’ with our parser. The other way is to use the Coq module made by the HELM project group which exports the Coq terms as XML documents. To this end they have developed a DTD for terms of the Calculus of Inductive Constructions in [4]. Section 4.7 describes the XML format.

The HELM method is obviously cleaner, and parsing the resulting XML document is probably easier than parsing the pretty-printed context we get from Coq. But it also involves more initial work to implement; luckily the HELM project group did this for us. For our purposes, all that matters is that there is a way to get from the abstract datatype in Coq to our own CoqTree datatype. After parsing, the CoqTree representing the context is annotated with type information, in every subterm the type of that subterm is stored. The algorithm for verbalization of proofs relies on this type information.
4.5. GENERATION OF DOCUMENTS

4.5.1 Mathematical Meta Texts

The treecview mentioned above presents proof-objects directly as trees in the main window of the CoqViewer. For other views it is profitable to first generate an intermediate datatype. Such a datatype can be shared by many views, and analysis and transformations can be performed on an intermediate level. In the views that verbalize proofs as natural language text, we make use of two such datatypes for verbalization: The MathStatement and the MathMetaText.

The intermediate datatype MathStatement can represent a sentence in a mathematical document. A MathStatement consists of MathUnits containing either informal text or formal OMOObjects.

**Notation 4.5.1** MathStatement.

1. The construction of a MathStatement out of MathUnits $u_1, \ldots, u_n$ is denoted as:
   \[ [u_1, \ldots, u_n] \]

2. Concatenation of MathStatements is denoted by the $+$ symbol.

The embedding of text units is a bit crude. It would be more flexible and sophisticated to choose the level of units a little more abstract, for example units like punctuation, adjectives, nouns, verbs. This would make the presentation more flexible, and also allows some fancy natural language transformations. It might be interesting to compare this to work such as [68], which attempts to provide a semi-formal level of description somewhere between natural language and type theory.

For proofs yet another intermediate datatype, named MathMetaText, is used. This datatype facilitates the internal interaction by which proofs can be inspected on multiple levels of detail. A MathMetaText consists of a list of MathStatements but may also contain recursive MathMetaTexts. Furthermore, every MathMetaText has a conclusion, which is just a MathStatement.

A MetaText contains a number of MathStatements which either consist of Objects or contain a recursive MetaText. Recursive MetaText statements also contain a conclusion statement. An Object is either some concrete text string or it is a pointer to a CoqTree representing a mathematical object term.

**Notation 4.5.2** The construction of a MathMetaText with conclusion $s_0$ out of MathStatements $s_1, \ldots, s_n$ is denoted as:

\[
\left[ \begin{array}{c}
s_1 \\
\vdots \\
s_n 
\end{array} \right]_{s_0}
\]

Figure 4.18 shows a schematic presentation MathMetaText with some components.

The recursive nature of a MathMetaText provides the multiple levels of detail of a mathematical proof and also indicates the points in the representation where a proof
can be folded or unfolded. If we look at the example MathMetaText in Notation 4.5.2, this object can either be folded, in which case we just display $s_0$ or unfolded, in which case we display $s_1, \ldots, s_n$. If $s_1$ itself is again a MathMetaText $[t_1, \ldots, t_m]_{t_0}$, then $s_1$ can also be folded (displaying $t_0$) or unfolded (displaying $t_1, \ldots, t_m$). Whether specific objects are folded or unfolded is determined by the reader via the interface, but the author might provide a preferred setting along with the document, determining the statuses of all the folding/unfolding points at start-up.

To summarize, for the statement of an axiom or theorem (encoded by the type of a definition) we use MathStatement, and for the proof of a theorem we use MathMetaText. For non-propositional objects we just use OMObjects by employing the Coq-to-OpenMath Codec [24].

Here is how we verbalize propositional statements to a MathStatement. Propositional statements are generated from objects of type Prop. Note that this verbalization treats the propositional connectives as though they were primitives, even though these connectives have a definition in Coq, given in Example 2.5.17.

**Definition 4.5.3** Verbalization $s(\_)$ of a propositional statement as a MathStatement.

\[
\begin{align*}
    s(\text{not } A) &= ["\text{not}"] + s(A) \\
    s(\text{and } A B) &= s(A) + ["\text{and}"] + s(B) \\
    s(\text{or } A B) &= s(A) + ["\text{or}"] + s(B) \\
    s(A \rightarrow B) &= ["\text{if}"] + s(A) + ["\text{then}"] + s(B) \\
    s(\forall x: A.B) &= ["\text{for all} ", x, "\text{in} ", A] + s(B) \\
    s(\exists x (\lambda x: A.B)) &= ["\text{there exists an} ", x, "\text{in} ", A, "\text{such that}"] + s(B)
\end{align*}
\]

In the above definition all connectives are lifted to natural language. What remains formal are the atomic propositions, which are applications of predicates to mathematical objects. The atomic propositions are eventually encoded as OpenMath objects within
4.5. GENERATION OF DOCUMENTS

the OMDoc document. Alternatively the entire propositional statement could have been encoded as one big OpenMath object. To which degree propositional statements belong to the object language or to the meta language seems to be a matter of taste. Perhaps this choice should be left to the reader of the document, this option is easily implemented as an internal interaction feature, for example a button which switches the document to formula or natural language presentation of propositional statements.

Repetitions of implication result in nested if-then-else verbalizations which may look awkward. This is the result of curried function applications which are quite common in mathematics formalized in type theory. To counter this, de-currying is applied. This means that a statement of the form \( A \rightarrow (B \rightarrow C) \) is replaced by \( (A \land B) \rightarrow C \) prior to verbalization. Something similar can be done to verbalize repeating universal and existential quantifiers where the abstracted variables come from the same domain. For example \( \forall x: A. \forall y: A.B \) is verbalized as “[“for all”, x,” and”, y,” “in”, A] + s(B). Note that this is only presentation. The formal objects do not change.

4.5.2 Verbalization of Proof-objects

Proof-objects are verbalized via the MathMetaText datatype. An adaptation of Coscoy’s translation from [34, 33] is used. In the definition below, where appropriate, types of terms are indicated as superscripts.

Definition 4.5.4 Verbalization \( t(\cdot) \) of a proof-object as a MathMetaText. We use the verbalization of propositional statements \( s(\cdot) \) as given in Definition 4.5.3. The type of the proof-object is assumed to be \( \tau \) in each case.

\[
\begin{align*}
t(h) & = \left[ \text{“by”, h, “we have”} \right] + s(\tau) \\
t(M^{\forall x:A.B} N) & = \left[ t(M) \right] \left[ \text{“by taking”, N, “for”, x,” we get”} \right] + s(\tau) \\
t(M^{A \rightarrow B} N) & = \left[ t(N) \right] \left[ \text{“we deduce”} \right] + s(\tau) \\
t(\lambda h: A^{\text{Prop}.} M) & = \left[ \text{“assume”} \right] + s(A) + \left[ \text{“(”, h“)”} \right] \\
t(\lambda x: A^{\text{Prop}.} M) & = \left[ \text{“consider an arbitrary”, x,” in”, A} \right] \end{align*}
\]

Proof-objects containing repeated \( \lambda \)-abstracted variables from the same domain can be presented in a more compact way by de-currying, similar to the de-currying used in verbalization of propositional statements containing repeated universal or existential quantifiers, see the remarks following Definition 4.5.3.
The verbalization of proof-objects involving cases is below. However, in practice cases rarely occurs as a head symbol in a proof-object. As destruction of inductive values is done using elimination predicates within proof-objects (for proof-objects that are created in interactive proof-mode in the Coq system, the system uses the elimination predicates rather than the concrete fix and cases constructions). The names of some of the more frequently used elimination predicates are recognized and intercepted by our parser and treated as primitives of the calculus.

**Definition 4.5.5** Verbalization of a proof-object as a MathMetaText, cases-case.

\[
\begin{align*}
  t & \left( \langle \tau \rangle \text{cases } M . \left\{ \begin{array}{l}
  P_1 \Rightarrow B_1, \\
  \vdots \\
  P_n \Rightarrow B_n
  \end{array} \right\} \right) \\
  &= \left[ \begin{array}{l}
  \text{["we reason by cases on", } M] \\
  \text{["if", } M, \text{ "is", } P_1] \\
  t(B_1) \\
  \vdots \\
  \text{["if", } M, \text{ "is", } P_n] \\
  t(B_n) \\
  \text{["in all cases"]} + s(\tau)
  \end{array} \right] \! s(\tau)
\end{align*}
\]

There is no instance for the fix construction. The combination of fix and cases occurring in a proof-object suggests that the proof is by induction over an inductive type. At this moment we do not have a good solution for verbalization of such proofs.

For some of the frequently used inductive types in the Coq standard library we have special verbalization rules. We can detect such situations, since the generated elimination predicate will be used. Our verbalization rules are in Definition 4.5.7. We also give special verbalization rules for introduction of the logical connectives.

**Definition 4.5.6** Verbalization of proof-objects introducing connectives (using the constructors described in Examples 2.5.17 and 2.5.19).

\[
\begin{align*}
  t(\text{andintro } A B a b) &= \left[ \begin{array}{l}
  t(a) \\
  t(b) \\
  \text{["therefore"]} + s(\tau)
  \end{array} \right] s(\tau) \\
  t(\text{orintro } AB a) &= \left[ \begin{array}{l}
  t(a) \\
  \text{["therefore"]} + s(\tau)
  \end{array} \right] s(\tau) \\
  t(\text{orintror } AB b) &= \left[ \begin{array}{l}
  t(b) \\
  \text{["therefore"]} + s(\tau)
  \end{array} \right] s(\tau) \\
  t(\text{exintro } AP a H) &= \left[ \begin{array}{l}
  t(H) \\
  \text{["therefore"]} + s(\tau)
  \end{array} \right] s(\tau) \\
  t(\text{eqintro } A a) &= \left[ \begin{array}{l}
  \text{["by reflexivity"]} + s(\tau)
  \end{array} \right] s(\tau)
\end{align*}
\]
**Definition 4.5.7** Verbalization of proof-objects eliminating connectives (using the elimination predicates described in Examples 2.5.2, 2.5.17, and 2.5.19).

\[
\begin{align*}
\text{t(falseelim } P f) & = \begin{bmatrix} t(f) \\
["ex falso sequitur quodlibet", "so we have"] + s(\tau) \end{bmatrix} s(\tau) \\
\text{t(andelim } AB P f c) & = \begin{bmatrix} t(c) \\
t(d) + s(\tau) \end{bmatrix} s(\tau) \\
\text{t(orelim } AB P f_a f_b d) & = \begin{bmatrix} t(d) \\
t(f_a) + ["in any case we have"] + s(\tau) \end{bmatrix} s(\tau) \\
\text{t(exelim } A P P_0 f a) & = \begin{bmatrix} t(f) \\
t(a) + ["so we have"] + s(\tau) \end{bmatrix} s(\tau) \\
\text{t(eqelim } AX P HY q) & = \begin{bmatrix} t(H) \\
t(q) + ["replacing", X, "with", Y, "in", H, "we get"] + s(\tau) \end{bmatrix} s(\tau) \\
\text{t(natelim } P H_b H_s n) & = \begin{bmatrix} ["induction on", P] \\
t(H_b) + ["so we have"] + s(\tau) \end{bmatrix} s(\tau)
\end{align*}
\]

### 4.6 Presentation of Documents

As outlined in Section 4.2, presentation of the document takes place through *views*. This section describes the view mechanism and the two implemented views. Additionally, documents can be exported to OMDoc, which can be viewed in a standard browser with the help of appropriate stylesheets. Section 4.7 describes OMDoc. The idea is to have multiple views which present the underlying formal structure consistently. For example, when a user changes the preferred level of detail at which a CoqTree object is viewed, this information is stored in the CoqTree itself and distributed to all views. Currently only two views are available, the TreeView and the NLView. The context can also be exported as an OMDoc XML document, so that it can be viewed in a standard browser.

To facilitate the translation to OMDoc, we implemented Java classes corresponding to the different elements that can occur in an OMDoc document. Each context item
in the CoqTree context is translated to an OMDoc context item object. Here the type information present in the CoqTree is used to make the distinction between ordinary assumptions and definitions on the one hand, and axioms and proofs of theorems on the other.

### 4.6.1 Tree View

The most basic view is called **TreeView** and just presents the context as a large tree, showing the nodes described in Section 4.3. Each node can be either collapsed or expanded. If a node is collapsed, the type of the tree starting beneath that node is displayed. If a node is expanded, the complete tree is displayed. See Figure 4.4 for a screenshot of this view. As described in Section 4.2, this view allows some simple editing of the presentation information. This view does not make use of MathMetaText or MathStatement.

### 4.6.2 Natural Language View

The natural language view **NLView** is based on the standard translation algorithm presented in [34] and [33]. However, instead of generating flat text, objects of a new class called MetaText are generated, see Figure 4.18.

The MetaText datastructure is intended to facilitate both the accessibility of the underlying formal proof-object, a readable natural language text representing the proof, as well as the folding and unfolding mechanism of proofs in the natural language view. Moreover, assumptions and their scope can be marked which makes Fitch style presentation of proofs possible, see [88, 87]. See Figure 4.5 for a screenshot of this view.

A CoqTree object is translated to a MetaText using the algorithm above. A typing algorithm is necessary for this translation, see Section 4.4.7.

### 4.7 XML, OpenMath and OMDoc

This section describes the XML language and two XML applications for specification of mathematical content: OpenMath and its extension OMDoc. Both the design goals and a large part of the syntax are described here. However, a complete description of these languages is outside the scope of this thesis; only the features we actually use are described. A good introduction to XML is [47].

#### 4.7.1 XML

The eXtensible Markup Language (XML) [85], is an emerging standard for storing and communicating structured content. XML is a meta markup language, in the sense that it describes other languages. XML is derived from SGML, but XML is a simpler standard. Although applications of SGML such as the HTML language have become quite
popular, the SGML language itself turned out to be too complicated, and a simpler version, XML, was needed.

In an XML document first a grammar, the Document Type Definition (DTD), is given prescribing the “dialect” for the rest of the document. A language specified in XML is called an XML application. Next, after (a link to) the DTD, the document contains a number of elements consisting of opening and closing tags. For example, an element named thesis would have opening tag <thesis> and closing tag </thesis>. Each tag can have several attributes, which are name and value pairs specified in the opening tag of the element. For example, perhaps our thesis element has an author attribute instantiated as <thesis author="M. Oostdijk">. Note that the value of an attribute is always surrounded by quotation marks. Which elements are allowed and what attributes they have is specified in the DTD.

The HTML language is an example of an SGML application. It is not an XML application as many HTML elements cannot be well-formed XML elements. For example all XML elements have to have a closing tag. In HTML the <hr> tag does not have to be closed with an </hr> tag. Furthermore, XML demands that values of attributes are enclosed in quotation marks. In HTML this need not be the case. However, there is an XML application called XHTML which is very close to HTML. XHTML is a real XML application, i.e. there is a DTD for it. In a well-formed XHTML document each tag has to be closed so one has to write <hr></hr>. The <hr/> notation may be used as a shortcut for <hr></hr>. This hold for all XML elements, of course.

An XML document written according to one DTD can be transformed into another XML document of a different DTD using a stylesheet. Stylesheets are formulated in XSL [86], the XML stylesheet language. XSL is a pattern matching language, which happens to be an XML application itself. When transforming for example to XHTML, the stylesheet can be seen as a specification of how documents should be presented. This allows a rigorous distinction of presentation and content. The content is specified in the XML document according to a previously defined DTD, the presentation is specified in a stylesheet.

There are many advantages to using the XML metalanguage. Open standards which exceed the proprietary formats of individual systems are good and XML makes designing such open standards a little bit easier. Parsing is potentially easier as technology from others can be used, which means parsing can be done on a higher level. Other tools such as viewers and editors can also be reused. Since the distinction between content and presentation is made explicit, more attention is given to what really is the content which leads to easier sharing of content.

A drawback to XSL is that it is not as flexible to program in as a full programming language. Although XSL is Turing complete according to [47], it remains a simple pattern rewriting language. The XML itself does not guarantee that content and presentation are separated, or that content can be shared among different applications. The DTD only prescribes how the XML elements are to be nested. The actual content of the elements can be arbitrary text, which may contain its own encoding, not specified through XML. The following fragment shows that flat binary formats are still very possible.
The designer of the DTD and XSL stylesheets still has the responsibility to design a meaningful and easy to understand grammar. From a language theoretic point of view, the ideas behind XML are not very shocking. In order to do any complex computations, the XML document still needs to be parsed into some internal datastructure.

Yet, for our purposes XML is ideal, as it allows us to draw the distinction between content and presentation explicitly in the document. It is not surprising that the Computer Algebra community embraces XML for encoding the OpenMath and OMDoc standards described below.

### 4.7.2 OpenMath

The OpenMath standard [74, 75] is a language intended to share expressions between computer mathematics systems. It provides primitives for variables, abstraction and application. In order to be universally applicable, OpenMath theories can be parameterized with a so-called content dictionary (CD) containing names and types for new symbols. If two CA systems want to communicate objects, they first have to agree on a CD before they share any mathematical objects. For example, the \( \sin \) and \( \cos \) symbols are most likely in the trigonometry CD. If two systems want to share those symbols, they have to use the same CD. Before the communication starts both systems have to know which CD is being used. In the context of an OMDoc document, the CD is ‘defined’ on the fly by the document.

The following elements of OpenMath are relevant for our implementation. Each of the elements is described using an example. For the full description of the DTD, see [75].

- `<OMOBJ>`: The top level element containing an OpenMath object.
- `<OMV>`: Variables that are locally bound. Variables have a name attribute and usually do not contain other elements. For example, the variable \( x \) is denoted by `<OMV name="x"/>`. The name attribute is used to bind the variable, which will be explained below.
- `<OMI>, <OMF>`: These represent constants of various types. For example, the integer 42 is denoted by `<OMI>42</OMI>` and the floating point number 3.1415 is denoted by `<OMF dec="3.1415"/>`.
- `<OMS>`: Symbols are defined in a Content Dictionary (CD). The CD describes a group of symbols. For example, the arithmetical + operation is denoted by `<OMS cd="arith" name="plus"/>`. 

4.7. XML, OPENMATH AND OMDOC

- **<OMA>**. Functions are applied to zero or more elements using this element. It should contain the function and each of the arguments. For example, the following denotes the expression $x + 42$:

  ```xml
  <OMA>
    <OMS cd="arith" name="plus"/>
    <OMV name="x"/>
    <OMI>42</OMI>
  </OMA>
  ```

- **<OMBIND>, <OMBVAR>**. Variables can be bound by an <OMBIND> element. The element contains a symbol indicating the kind of binding, the binding variable which is an <OMBVAR> element, and an object in which the abstracted variable may be used. Binding of variables is used for example for $\lambda$-abstraction. The expression $\lambda x.x$ is denoted as follows:

  ```xml
  <OMBIND>
    <OMS cd="lc" name="Lambda"/>
    <OMBVAR><OMATTR>
      <OMATP>
        <OMS cd="icc" name="type"/>
        <OMV name="A"/>
      </OMATP>
      <OMV name="x"/>
    </OMATTR>
  </OMBVAR>
  </OMBIND>
  ```

- **<OMATTR>, <OMATP>**. Attributes can be assigned to OpenMath elements. Note that these OpenMath attributes have nothing to do with the XML attributes that may occur in elements. An OpenMath attribute simply links one OpenMath object to another. We use attributes for assigning types to elements. The <OMATTR> element contains the attributed property, which is an <OMATP> element, and the OpenMath element which is to be attributed. The <OMATP> element contains a symbol indicating the kind of attribution and the attribute element. For example, the typed $\lambda$-abstraction $\lambda x:A.x$ is denoted by the following. A property is added to the binding variable $x$, the property states that its type is $A$.

  ```xml
  <OMBIND>
    <OMS cd="lc" name="Lambda"/>
    <OMBVAR>
      <OMATTR>
        <OMATP>
          <OMS cd="icc" name="type"/>
          <OMV name="A"/>
        </OMATP>
        <OMV name="x"/>
      </OMATTR>
    </OMBVAR>
  </OMBIND>
  ```
The OpenMath language is designed to be flexible, because mathematics is constantly changing. The Symbol/CD mechanism makes sure that the language does not restrict itself to only a fixed set of CA systems. CA systems are under developments with respect to mathematical content. New mathematical theories with new symbols can be dealt with by defining a new CD. The attribution mechanism also contributes to the flexibility of the language.

A CD for type theory is available, see [24]. Symbols for $\lambda$- and $\Pi$-binding as well as inductive types, case analysis, and fixpoints are defined in this CD. Also we have Java technology to generate OpenMath XML output from the CoqTree datatype described in Section 4.3. This is a relatively straightforward encoder.

However, for some mathematical purposes the OpenMath language is not flexible enough. For example for encoding contexts. OpenMath can only specify objects. Because CASs have a large collection of common predefined objects it is sufficient to have the ability to exchange objects only. The problem OpenMath solves is indicating which symbol stands for which mathematical concept.

OpenMath is useful for communicating mathematical objects between computer algebra systems. OpenMath only describes objects not theories or documents or contexts. The ability to encode contexts is essential if one wants to communicate with theorem prover systems. The CD mechanism of OpenMath is not flexible enough for theorem provers: Contexts in which one can define new concepts are needed. Another feature that is missing is a query language or protocol for dynamically figuring out the capabilities of other systems. Although in theory contexts can be encoded as ordinary objects, see for instance [18], the need is felt for a language which treats contexts as first class citizens. The solution for the context problem lies in an extension of OpenMath called OMDoc.

### 4.7.3 OMDoc

The OMDoc language [60, 61, 70] is an extension of OpenMath which may be used to describe mathematical documents rather than mathematical terms. OpenMath is embedded in OMDoc: OMDoc uses OpenMath to express the individual terms. An important property of OMDoc documents is that they may also contain informal textual parts. The following elements of OMDoc are relevant for our implementation. Each element is described using an example. For the full description of the DTD, see [70].

- **<symbol>**. This element declares a new mathematical symbol. In the remainder of the document the symbol may be used in OpenMath objects as if it was declared in a CD. Example:

  ```xml
  <symbol id="i" type="object" scope="global">
  ```
• **<definition>**. This element defines properties of symbols declared earlier in the document. It is not required to give a definition in "dictionary style" where a symbol assigned an expression which completely defines it. However, the definitions generated from Coq definitions will always completely define an object. Example:

```xml
<definition id="isReflexive"
  type="inductive"
  item="leibniz.context">
  <CMP> ... </CMP>
  <FMP><OMOBJ> ... </OMOBJ></FMP>
</definition>
```

• **<axiom>**. This element declares a new axiom. Example:

```xml
<axiom id="assoc">
  <CMP><OMS cd="dummy" name="R"/> is associative</CMP>
  <FMP>
    <OMOBJ>
      ... <OMS cd="dummy" name="R"/> ...
    </OMOBJ>
  </FMP>
</axiom>
```

• **<assertion>**. This element postulates a new assertion which may be proved later in the document. The type of assertion can be refined to "lemma" or "theorem" using the `type` attribute. Example:

```xml
<assertion type="theorem" id="uu">
  <CMP> ... </CMP>
  <FMP><OMOBJ> ... </OMOBJ></FMP>
</assertion>
```

• **<proof>**. This element states a proof of an assertion. An assertion can have any number of proofs. Proofs consist of derivation steps and have a conclusion. Furthermore, proofs may recursively contain proofs. Example:
CHAPTER 4. INTERACTIVE MATHEMATICAL DOCUMENTS

• <derive>. This element denotes a derivation step inside a proof. Example:

  <derive id="step1">
    <CMP> Trivial </CMP>
  </derive>

• <conclude>. This element denotes the conclusion of a proof. Proofs end in a conclusion indicating what statement was proved. Example:

  <conclude id="concl">
    <CMP> ... </CMP>
  </conclude>

• <CMP> and <FMP>. These elements stand for commented and formal mathematical property, respectively. These are used inside the other elements.

All these OMDoc context items have a formal <FMP> and a commented <CMP> part. The <FMP> part of definitions and assumptions is easy. We translate the CoqTree to an object of class OMObject using the Coq-to-OpenMath Codec described in [24].

The MathMetaText objects representing proof-objects are easily translated to the <proof> elements in the OMDoc document. The MathStatements are translated to <CMP> parts of those <proof> elements and the other OMDoc context item elements. Formal units are replaced by OMObjects, while string units remain text. Therefore even though the <CMP> elements are commented content, they do contain formal parts. Finally the OMDoc XML can be transformed to Dynamic HTML using an XSL stylesheet. Dynamic HTML supports folding and unfolding of trees which we use for the proofs.

In more detail: An assumption of propositional sort, \( x : A : \text{Prop} \), becomes <axiom>, and an assumption \( x : A \) of non-propositional sort becomes <symbol>. A definition \( x := M : A \) becomes an <assertion type="theorem"> element with a <proof> element if \( A : \text{Prop} \), and a <symbol> element with a <definition> element otherwise.

The OMDoc document can be exported in XML format. Consider for example a step from the proof of reflexivity of Leibniz’ equality:

  <derive id="leibniz_refl-prf.2p.2p.2p.1">
    <CMP>
      By
      <OMOBJ>
        <OMV name="H"/>
      </OMOBJ>
    </CMP>
  </derive>
This step contains a commented mathematical property, between `<CMP>` tags, which consists of natural language text, for example “by” and “we have”, with embedded OpenMath objects, for example $H$ and $P(x)$.

The generated XML can be transformed into for example HTML using an appropriate XSL style-sheet. The OMDoc view is not really a view in the sense described above. The presentation of the generated XML is done in a different tool, XSLT is an example of such a tool. Depending on which XSL stylesheet is used, XSLT generates a different output format. For example the `<OMA>` part in the above OMDoc fragment is presented as $P(x)$, where the stylesheet provides the parentheses, but using a different stylesheet perhaps it is presented as $(P \ x)$. Stylesheets can in principle take care of some advanced presentation issues such as currying of applications. Using a different tool for presentation has a drawback that there is no direct connection back to our internal CoqTree datastructure the document was generated from. However, all the formal MathObjects which occur in a MetaText are translated to OpenMath objects. Therefore, the formal content is still present in the OMDoc document.

The use of standard export formats for mathematics like OpenMath and OMDoc potentially allows communication of objects to symbolic computation engines such as theorem provers and computer algebra systems. This may lead to true interactive mathematical documents in the sense of [27, 26], and is subject of future work.

To summarize, the use of XML enables a rigorous distinction between content and presentation. The XML applications OpenMath and OMDoc are good candidates for representing mathematical content as both the expressions and the meta-structure of a mathematical document can be dealt with.

### 4.8 Conclusions

We developed a method to create interactive mathematical documents based on formal mathematical content developed inside the Coq theorem prover. The resulting documents can be viewed in our prototype CoqViewer tool, and can be exported to an XML based presentation language called OMDoc. The formal structure of the mathematics is still present in the resulting document, yet the document can be presented as informal
mathematics to the end user. The presence of formal structure in the document allows internal interaction, a form of interaction based on structured content. For example, using the fact that proof-objects are treated as first class citizens in the calculus of Coq, we can provide the reader with a view on the document in which proofs can be viewed at multiple levels of detail. The reader can interact with the proofs by adjusting this level of detail.

In order to have a prototype for experimentation, we created a tool called CoqViewer. The tool can be used for authoring and presenting mathematical content based on type theory. The Coq system is used to create an initial mathematical theory. Using the tool, an author can add presentation information to the definitions and proof-objects that form a formally derived mathematical context.

In the implementation many new technologies are used. Java is used as the implementation language. The Coq-to-OpenMath encoder in [24] is used for the formal mathematical terms. For the proofs we first generate MathMetaText objects. The natural language generation is based on Coscoy's translation [34, 33] of proof-objects. It is not a very sophisticated verbalization algorithm, but the results look very promising. The OMDoc Java package we implemented follows closely the structure of the OMDoc DTDs, it consists of about 20 classes describing all the OMDoc elements. The use of Java and the various XML application makes it easy to share code with other authors. For examples we are using the OpenMath library from the PolyMath group [80], we use the XSLT transformation program and other XML4J software from IBM, we use stylesheets developed for OMDoc by the Omega project group (although the stylesheets need more work to enable folding and unfolding of proofs, but this should be easy), in the future we may start using the HELM module, this should also be easy and makes us less dependent on Coq.

Although we use Java as an implementation language, we do not really exploit the object oriented features when implementing the datatype representing CIC expressions. Rather, we take a straightforward approach and implement terms as pointer trees. Abstraction of variables is also internally represented using pointers. Operations on such trees consist of a large case analysis on the root symbol of the tree, and such a case analysis shows more resemblance to a functional style of programming than to an object oriented one. This is convenient as most of these algorithms are specified in the literature in a functional style. However, the use of pointers is alien to functional programming. This combination of features makes implementing the usual algorithms such as copying, checking syntactical equivalence, and type inference non-trivial.

Type inference is needed in order to implement the natural language view. Because it can be assumed that the input to the tool is a checked mathematical context, some side conditions are not checked during reduction and type inference.

One problem our tool cannot deal with is caused by user-defined Coq syntax rules. Coq allows the specification of special syntax for mathematics formalized in Coq. We are presently not able to parse such syntax. If we were to use the HELM module which directly operates on the Coq internal datatypes, these syntax rules would no longer pose a problem since the problem is caused by the fact that we parse the pretty-printed
output of Coq. We could also try to parse the special syntax rules. As presentation is very important, we should be careful not to treat it as just a user interface issue. CtCoq [17] and its successor PCoq [64] allow user specified syntax rules for presentation. Our pretty-printing is hardwired into the CoqViewer or OMDoc stylesheet. In case of an OMDoc stylesheet, the user can extend such a stylesheet and put pretty-printing information in there.

We applied this technology to a case study found in Chapter 5. In future work we plan to apply it to some bigger examples. More future work can be found in making the intermediate level of MathStatements more subtle, maybe draw some inspiration from Nederpelt’s Pseudo Type Theory [68] or the proof-planning provers such as Omega [15], which seem to provide a level between formal mathematics and informal natural language, although they are more interested in the opposite direction, i.e. going from informal to formal. For longer term future work we would like to investigate the reverse translation from (a subset of) OMDoc to Coq, but it is unknown how difficult this is.
Chapter 5

Theorem Proving and Computer Algebra

5.1 Introduction

As we have seen in Chapter 3, theorem provers do not excel in checking mathematics that involves computational tasks. For example verifying primality of numbers proves to be difficult due to inefficient handling of datastructures and computations in the theorem prover. Of course, any computer mathematics system has limitations on its computational power, but one wonders why the difference in computational capabilities between theorem provers and for example computer algebra systems exists. What is holding theorem provers back?

The answer lies in the way mathematical notions are implemented in these different computer mathematics systems. Mathematics used in computer algebra systems and theorem provers is implemented differently, motivated by the different uses of these systems. In computer algebra systems a large set of notions is already defined in the implementation language. All mathematical notions in theorem prover systems are defined in the object language. Furthermore, the implementation of the datastructures in a CAS is geared towards efficiency instead of correctness, whereas correctness is the key issue in the theorem proving world.

Both systems have useful properties which originate from key architectural design decisions. The main question is: Can computer algebra systems be used in combination with theorem provers to get efficient yet trustworthy computations? We want to know whether it is in principle possible to combine these systems, but we are also interested in the technical details of combining computer mathematics systems in general. Already many attempts from both sides have been made to answer this question, for example [50] and [57].

In answering the question whether we can combine these systems without losing trustworthiness or efficiency, we have a preference for trustworthiness as we are viewing this problem from a TP standpoint. There are three strategies for combination
that more or less maintain trustworthiness and efficiency. The first strategy, “TP as
side condition checker”, considers the computer algebra system as the most impor-
tant component and the theorem prover is called upon to assist when correctness be-
comes an issue. The second strategy, “CA as oracle”, considers the theorem prover as
the most important component and the computer algebra system is used to assist the
theorem prover whenever efficient computations are needed. The third strategy, “CA
and TP in framework”, is to consider CA and TP as equally important; they incorpo-
rated in a larger framework in which both components can be asked to solve prob-
lems. The larger framework would ideally be the mathematical workspace discussed
in Chapter 1. There is a wide amount of literature both on the subject of combining
systems for problem solving and on specific case studies, among many see for instance
[16, 20, 22, 23, 24, 29, 36, 43].

The first strategy takes the CA as the most prominent component. It can be imple-
mented as a computer algebra system which has limited theorem proving capabilities
to check for example side conditions in symbolic computations, for example described
in [1] and in [14]. Other incarnations of this strategy implement a propositional or pred-
icate logic prover within the computer algebra system. See, for example, Buchberger’s
Theorema project [20].

The second strategy can be implemented for example as a theorem prover which can
consult a symbolic computation engine oracle, for example to find witnesses that require
heavy computation or to verify an equation. In order to make proofs that depend on
computations trustworthy, either the computations have to be repeated inside the TP or
there must be a way to verify that the witnesses are correct. In [12] the Lego theorem
prover is extended with oracle types, which allow efficient rewriting using a link to
Reduce.

The third strategy combines both systems using an external program. This requires
much more work on a higher level. But it has the advantage that the computer math-
ematics systems involved can be considered as black boxes. Moreover, if a standard
language can be used, like OpenMath, we might save a lot of work in the future since
we can use standard interchangeable components. However, the current state of the art
requires us to adapt current systems to comply to the standard framework.

In this chapter we develop an example implementation of an interactive document
which uses the “CA as oracle” approach. In spirit, this is the second strategy. In reality,
however, the “CA and TP equally important in large framework” approach, the third
strategy, is used. The end result is a formal proof in the TP world while using interme-
diate results from the CA oracle. To the user the interface is an interactive mathematical
document that makes use of external interaction as mentioned in the introduction of
Chapter 4.

The example implementation enables the user to create a document proving the pri-
mality of a number. This is a good example since it requires heavy computations on
concrete numbers, something a TP is not good at. If we can get this working, we will
have shown that combining a CA and a TP is feasible. We show that the technology
exists to combine existing packages.
In [8] two distinct ways are presented of how to interpret the results of a computational oracle. First, in the **believing** way, theorems stating a computational result are assumed as axioms. By choosing good names for such axioms, the proof-object reflects which statements were believed during the proving. For example, we could introduce the following axiom in Coq.

\[
\text{Axiom maple_add_1_1: (plus (1) (1))=(2).}
\]

When we need to prove that \(1+1=2\) we simply apply this axiom. The proof-object \(\text{maple_add_1_1}\) then corresponds to “Maple says that \(1+1=2\)”. This means that the places in the proof where the author refers to this belief in a CAS are clearly marked as such. The believing way of interpreting oracle results allows us to easily incorporate results of computational tasks where we fully trust the CAS.

Second, in the **skeptical** way, when an oracle is consulted to do the computation, it is asked to also generate a **trace** or **certificate**. By inspecting the trace or certificate the TP can check the results of the computation. For example, suppose we want to prove \((x+1)^2=x^2+2x+1\) using lemmas stating the properties of the operators such as associativity, commutativity, distributivity, etc. We could ask a CAS to do this and give us a trace such as \(\langle D_R, D_L, D_L, (C_M x1), \ldots \rangle\). The TP still needs to do the work constructing a proof-object by applying the lemmas, but the oracle finds out which lemmas to apply and in what order.

The fully skeptical constructive approach, in which no axioms are assumed, is perhaps a cultural thing in the Coq community, quite contrary to the more pragmatic approach of say the PVS community, which is willing to believe. Examples from mathematical practice of both approaches are given in [9].

This chapter presents how Pocklington’s criterion can be employed to produce efficient formal proof-objects that show primality of large positive numbers in a proof assistant such as Coq [11]. This entails a formal development in Coq of Pocklington’s criterion, and a study of how computer algebra software can assist as an oracle to Coq. Finally we give details of the implementation of these ideas. The part about Pocklington’s criterion is based on [26] and [28].

The structure of this chapter is as follows. Pocklington’s theorem, a criterion for testing primality for large numbers, is introduced. The theorem was formalized in Coq and this formalization is described in Section 5.2. Section 5.3 shows how the criterion can be used to produce formal proofs of primality for concrete numbers with the aid of computer algebra oracles. Our implementation of the algorithm results in a prototype interactive mathematical document showing a formal Coq proof of primality. Timings for some benchmarks are summarized in Section 5.4.

### 5.2 A Formal Proof of Pocklington’s Criterion

The problem of showing whether a positive number is prime or composite is historically recognized to be an important and useful problem in arithmetic. Since Euclid’s times,
the interest in prime numbers has only been growing. For today’s applications, primality testing is central to public key cryptography and for this reason it is still heavily investigated in number theory [81].

The problem is clearly decidable, the trivial algorithm, that checks for every number $q$ such that $q \leq \sqrt{n}$ whether $q|n$, is far too inefficient for practical purposes. There exist several alternative methods to check primality and in this chapter we deal with a classical criterion due to Pocklington in 1914 [78]. Our interest is motivated by the fact that in order to produce a proof of primality the criterion needs to find numbers that satisfy certain algebraic equalities. These numbers are easily generated using a computer algebra package, for instance Gap [39].

Notice that the cooperation of theorem provers with computer algebra is essential for being able to solve this task. Theorem provers are very limited in the amount and type of computations they can perform [8], however, they are very well suited for organizing the logical steps of a proof. On the other hand, although computer algebra systems have algorithms for deciding whether a number is prime or not, they cannot produce a proof of primality. Thus, the winning strategy is to combine both kinds of systems.

The Pocklington criterion is one of many number theoretical results that are useful for verifying primality of a positive number $n$. The work we present in this section stems from the study that Elbers did in his PhD thesis [37]. It is a continuation of [24, 25], where it is shown how an informal textual proof of primality generated by GAP can be turned into an interactive mathematical document. The complete development consists of a number of Coq vernacular files available online at [65].

### 5.2.1 Pocklington’s Criterion

The presentation of the criterion is roughly the informal proof as it appears in [37], however, towards a full formalization, more details are given here.

**Theorem 5.2.1 (Pocklington’s Criterion)** Let $n > 1$ with $n - 1 = qm$ such that $q = q_1 \cdots q_t$ for certain primes $q_1, \ldots, q_t$. Suppose that $a \in \mathbb{Z}$ satisfies $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{n-1}/q_i - 1, n) = 1$ for all $i = 1, \ldots, t$. If $q \geq \sqrt{n}$, then $n$ is prime.

**Proof:**

Let $p|n$ and Prime$(p)$, put $b = a^n$.

Then $b^q = a^{mq} = a^{n-1} \equiv 1 \pmod{n}$.

So $b^q \equiv 1 \pmod{p}$.

Now $q$ is the order of $b$ in $\mathbb{Z}_p^*$ because:

Suppose $b^{\frac{p}{q}} \equiv 1 \pmod{p}$, then $a^{\frac{mq}{n}} = a^{n-1} \equiv 1 \pmod{p}$.

There exist $\alpha, \beta \in \mathbb{Z}$ such that $\alpha(a^{n-1}/q - 1) + \beta n \equiv 1 \pmod{p}$.

So, $\alpha(1 - 1) + \beta n \equiv 1 \pmod{p}$. Contradiction.

By Fermat’s little theorem: $b^{p-1} \equiv 1 \pmod{p}$,

therefore $q \leq p - 1$, so $\sqrt{n} \leq q < p$. 


Hence for every prime divisor $p$ of $n$: $p > \sqrt{n}$.
Therefore $\text{Prime}(n)$.

The $m$ is there to avoid a full factorization of $n - 1$. Although in our approach, the CA does a full factorization of $n - 1$ to determine optimal $q$ and $m$. With optimal we mean optimal from the point of view of the theorem prover. This is because we are not interested in avoiding work for the CA, but in avoiding work for the TP, as the speed of the CA is much higher than the speed of the TP. We want to avoid work for the TP at the expense of the CAS. Although Pocklington’s criterion makes it possible to guess a good $q$, and not do the full factorization of $n - 1$, we make the CA do a factorization of $n - 1$ anyway, in order to choose a $q$ which provides the TP with the least amount of work. See Section 5.3.3 for the exact choice of $q$ and $m$.

This is what the formalized version looks like.

Theorem pocklington:
$(n,q,m:\text{nat})(a:Z)(\text{qlist:natlist})$
$(\gt n (1)) \rightarrow$
$n=(\text{S (mult q m)}) \rightarrow$
$q=(\text{product qlist}) \rightarrow$
$(\text{allPrime qlist}) \rightarrow$
$(\text{Mod (Exp a (pred n)) '1' n}) \rightarrow$
$(\text{allLinCombMo danm qlist}) \rightarrow$
$(\le n (\text{mult q q})) \rightarrow$
$(\text{Prime n})$.

Note that the $\gcd$ requirement in the original statements of the theorem is replaced by an application of $\text{allLinCombMod}$, which says that $1$ is a linear combination of $(a^{n_i} - 1)$ and $n$ for each prime factor $q_i$ of $q$. Obviously this is equivalent to the original requirement.

Although the proof is easy from a mathematical viewpoint, full formalization in Coq, i.e. finding a proof-object inhabiting the above statement, is not straightforward at all. Our development of the criterion is a full formalization, meaning that no lemma is assumed as axiom. Building the formal proof from the basic primitives requires a lot of work. However, the Coq proof assistant is equipped with a library of defined concepts concerning standard mathematical theories including natural numbers, integers, relations, and lists. Furthermore, as we have seen in Chapter 2, it has a powerful language of tactics which allows the user to specify abstract tactic scripts instead of concrete lambda terms.

The proof proceeds using several technical lemmata which mimic the high level reasoning of the informal proof. These technical lemmata are necessary for two reasons. First of all, the informal proof uses forward reasoning whereas the Coq system, being a goal directed theorem prover, uses backward reasoning. In the backward reasoning style of theorem proving, the user is presented with a goal to prove and works his way back
to the assumptions on which this goal depends by means of tactics. The informal proof, however, proceeds by contradiction introducing an arbitrary prime divisor $p$ of $n$. It shows that $p > \sqrt{n}$, and from this it concludes that $n$ must be prime. This forward style of reasoning can be simulated in Coq by first proving some lemmata.

The second reason for dividing the proof up in technical lemmata is the size of the steps in the informal proof. These are simply too big for the theorem prover. The user has to help the prover interactively and in order to get a grasp of the complexity it is necessary to recognize some high level lemmata.

However, before we can start with formalizing the informal reasoning of Pocklington’s proof, we are forced to construct the underlying mathematical theory. When Coq is started, it loads the standard library which contains concepts and results from logic and arithmetic, but we have to build a lot more before we can even state Pocklington’s criterion.

### 5.2.2 Modules

Formalization starts with identifying those mathematical concepts used in the proof that are not yet in the standard library. Many notions that are part of the repertoire of any mathematician are not (yet) in the standard library. The most prominent of the concepts found in the proof of Pocklington’s lemma are division and primality on the naturals, equality modulo $n$, greatest common divisor, exponentiation, and the order of an element $b$ in the multiplicative group $\mathbb{Z}_p^*$. They are formalized in the natural way by the following definitions.

\[
\begin{align*}
\text{Definition} \text{ Divides} & := [n,m:nat](EX q:nat \mid m=(\text{mult } n \ q)). \\
\text{Definition} \text{ Prime} & := [n:nat](gt n (1)) \land (q:nat)(\text{Divides } q \ n) \Rightarrow q=(1) \lor q=n. \\
\text{Definition} \text{ Mod} & := [a,b:Z;n:nat](EX q:Z \mid 'a = b + (\text{inject}_n \ n) \times q'). \\
\text{Definition} \text{ common\_div} & := [x,y:Z; d:nat] \\
& \quad (\text{Divides } d \ (\text{absolu } x)) \land \text{Divides } d \ (\text{absolu } y). \\
\text{Definition} \text{ gcd} & := [x,y:Z; d:nat](\text{common\_div } x \ y \ d) \land \\
& \quad ((e:nat)(\text{common\_div } x \ y \ e) \Rightarrow (\text{le } e \ d)). \\
\text{Fixpoint} \text{ Exp} \ [a:Z;n:nat]: Z := \\
& \quad \text{Cases } n \text{ of} \\
& \quad \quad 0 \Rightarrow '1' \\
& \quad \quad (S \ m) \Rightarrow 'a \times (\text{Exp } a \ m)' \\
\text{end}. \\
\text{Definition} \text{ Order} & := [b:Z][q,p:nat](\text{lt } 0 \ q) \land \\
\end{align*}
\]
The definition of Exp is a so-called fixpoint definition, which means that Exp is defined by well-founded recursion. In practice this means that the Coq term representing \( \text{Exp}(a, n) \) is convertible to \( a^n \) for concrete values \( a \) and \( n \), and for example finding an inhabitant for the statement \( \text{Exp}(2,3) = 8 \) is as easy as finding an inhabitant for \( 8 = 8 \). We study this phenomenon in Chapter 3. The definitions of Divides and Mod do not have this computational behavior, in order to prove for example \( \text{Divides}(2,8) \), one has to provide the witness \( q = 4 \). So, proving concrete instances of Exp statements can be handled using internal computation, while Divides relies on oracle computations. See Chapter 3 for definitions of these models of computation.

The definition of concepts alone is not enough. Many trivial (and less trivial) lemmata about the concepts have to be proved so that they can be used in the course of proving Pocklington’s criterion. The definitions together with the lemmata are grouped together in Coq modules representing mathematical theories. In this way, the theories can be reused when formalizing other parts of mathematics. Figure 5.1 gives an overview of the different modules developed to prove Pocklington’s criterion. For the modules that are not part of the standard library, the size is given. The modules in the figure are Coq vernacular files.

The Arith and ZArith modules are provided by Coq to support basic arithmetic on the natural and integer numbers respectively. The natural numbers in Arith are
implemented inductively with constructors for zero element and successor function, as shown in Example 2.5.2. This unary representation makes these natural numbers very inefficient for computation on concrete instances. The integer numbers in \( \mathbb{Z}_{\text{Arith}} \) are also implemented inductively but with a binary representation which is much more suited for concrete computations. However, when reasoning about abstract numbers, the binary representation can become a hindrance. To overcome these difficulties, Coq results are available in the \( \mathbb{Z}_{\text{Arith}} \) library that allow to convert between the slow naturals and the fast integers. We developed some more conversion related results in the \( \text{natZ} \) module, thus allowing switching between the two representations in the proof of the criterion. When the criterion is applied in Section 5.3 to generate concrete primality proofs, the binary representation is used exclusively.

The \( \text{natZ} \) module is part of a layer of modules built on top of the arithmetic modules. This layer develops some mathematical tools, it consists of datastructures and lemmas to allow a slightly higher level of reasoning later on. The module \( \text{lemmas} \) collects the additional lemmas on elementary arithmetic which were needed during the development. So, \( \text{natZ} \) and \( \text{lemmas} \) can be seen as the collection of results we would have liked to be in the \( \text{Arith} \) and \( \mathbb{Z}_{\text{Arith}} \) standard libraries.

The theory of finite lists is in the \( \text{list} \) module; it is heavily used in reasoning about prime factorizations and in the proof of Fermat's little theorem. The higher order type quantification of Coq allows defining of some abstract logical tools, for example \( \text{exlist} \) checks whether there exists a member in a list which has the property \( P \).

\[
\text{Fixpoint exlist \[A:\text{Set}; P:A\to\text{Prop}; \text{qlist}:\text{(list A)}\]: \text{Prop} := \\
\text{Cases qlist of} \\
\text{Nil} \quad \Rightarrow \text{False} \\
\text{| (Cons m l)} \Rightarrow ((P m) \lor (\text{exlist A P l})) \\
\text{end.}
\]

Using \( \text{exlist} \) we can define a membership predicate called \( \text{inlist} \).

\[
\text{Definition inlist := \[A:\text{Set}; a:A\](exlist A [b:A]a=b).}
\]

The “correctness” of \( \text{exlist} \), with respect to \( \text{inlist} \) and the real existential quantifier, can then be proved.

\[
\text{Theorem exlist_ok:} \\
\quad (A:\text{Set}; P:A\to\text{Prop}; \text{qlist}:\text{(list A)}) \\
\quad (\text{exlist A P qlist}) \leftrightarrow \\
\quad (\exists q:A \mid (\text{inlist A q qlist}) \lor (P q)).
\]

Many more properties of lists are in this module.

The \( \text{dec} \) theory contains lemmata useful for proving decidability of predicates in general. Decidability of a predicate \( P \) in the context of constructive theorem provers like Coq means that the principle of excluded middle, \( P(n) \lor \neg P(n) \), holds for \( P \). One may carry out a formalization in Coq in classical logic by assuming the principle of the
5.2. A FORMAL PROOF OF POCKLINGTON’S CRITERION

excluded middle as an axiom holding for any proposition. Instead, our formalization of
Pocklington’s criterion is done fully constructively. The dec module starts with some
decidability proofs for simple predicates.

\begin{align*}
\text{Lemma eqdec: } & (n,m:\text{nat}) \ n=m \ \lor \ \neg n=m. \\
\text{Lemma ledec: } & (n,m:\text{nat}) \ (le \ n \ m) \ \lor \ \neg (le \ n \ m). \\
\text{Lemma ltdec: } & (n,m:\text{nat}) \ (lt \ n \ m) \ \lor \ \neg (lt \ n \ m). \\
\text{Lemma gedec: } & (n,m:\text{nat}) \ (ge \ n \ m) \ \lor \ \neg (ge \ n \ m). \\
\text{Lemma gtdec: } & (n,m:\text{nat}) \ (gt \ n \ m) \ \lor \ \neg (gt \ n \ m). \\
\end{align*}

Other lemmata in dec serve as tools for proving decidability for compound predicates. Decidability is preserved by the propositional connectives.

\begin{align*}
\text{Lemma notdec: } & (P:\text{Prop}) \ (P \lor \neg P) \Rightarrow (\neg P \lor \neg \neg P). \\
\text{Lemma anddec: } & (P,Q:\text{Prop}) \ (P \lor \neg P) \Rightarrow (P \lor Q) \lor \neg (P \land Q). \\
\text{Lemma ordec: } & (P,Q:\text{Prop}) \ (P \lor \neg P) \Rightarrow (Q \lor \neg Q) \Rightarrow (P \lor Q) \lor \neg (P \lor Q). \\
\text{Lemma impdec: } & (P,Q:\text{Prop}) \ (P \lor \neg P) \Rightarrow (Q \lor \neg Q) \Rightarrow (P \Rightarrow Q) \lor \neg (P \Rightarrow Q). \\
\end{align*}

It is also preserved by bounded versions of the quantifiers. This means that proving de-
cidability of predicates like Divides and Prime is reduced to proving that the quantifiers
in the defining terms can be bounded.

\begin{align*}
\text{Theorem alldec:} \\
(P:\text{nat} \Rightarrow \text{Prop}) (N:\text{nat}) \\
((n:\text{nat}) (P n) \lor \neg (P n)) \Rightarrow \\
((x:\text{nat}) (lt x N) \Rightarrow (P x)) \lor \\
\neg ((x:\text{nat}) (lt x N) \Rightarrow (P x)). \\
\end{align*}

\begin{align*}
\text{Theorem exdec:} \\
(P:\text{nat} \Rightarrow \text{Prop}) (N:\text{nat}) \\
((n:\text{nat}) (P n) \lor \neg (P n)) \Rightarrow \\
(E X x:\text{nat} \mid (lt x N) \lor (P x)) \lor \\
\neg (E X x:\text{nat} \mid (lt x N) \lor (P x)). \\
\end{align*}

Note the similarity with the reflection method as used in the PRA reflection example
in Chapter 3. These lemmata make proving decidability for compound predicates, like
Prime and Divides, as easy as proving that the predicate belong to the class of primitive
recursive predicates. However, here the proving is done externally: The user has to
apply the right lemmata, although this can in principle be done mechanically.

Having developed the basic theory so far, it is possible to build the real mathematical
type needed for Pocklington’s criterion. The divides, prime, mod, gcd, exp, and
order modules define the mathematical notions introduced in the definitions given
above and contain many useful lemmata with proofs about these concepts.

Finally, after building all the modules, we can concentrate on the informal proof of
the criterion itself in the pock module.
5.2.3 Technical Lemmata

The informal proof of Pocklington’s criterion in Theorem 5.2.1 is already spelled out at a level of detail, normally not found in mathematical textbooks. A much more brief account of the same proof could be given, and would still be understandable by human mathematicians. The proof depends on a number of technical lemmas, among which Fermat’s Little Theorem, but this is the only one explicitly mentioned in the proof.

The ‘primepropdiv’ lemma (Lemma 5.2.2 below) corresponds to lines 1, 10, and 11 of the proof of Pocklington’s criterion. This lemma can be applied initially, applying it to the current goal, \( \text{Prime}(n) \). Doing so, the goal is replaced by new obligations to prove that \( n > 1 \) and \( p > \sqrt{n} \) for all prime divisors \( p \) of \( n \). This is the main result in the \texttt{prime} module, which is concerned with prime numbers. This theorem states that in order to prove primality of a natural number \( n \) it is enough to check divisibility by all primes up to \( \sqrt{n} \).

**Lemma 5.2.2 (‘primepropdiv’)** Let \( n \in \mathbb{N} \). If for every prime divisor \( p \) of \( n \), \( p > \sqrt{n} \), then \( n \) is prime.

**Proof:**

Assume \( p > \sqrt{n} \) for every prime divisor \( p \) of \( n \). Let \( d \mid n \), say \( n = dx \).

Suppose \( d \leq \sqrt{n} \),

Unless \( d = 1 \), there is a prime factor \( p \) of \( d \) with \( p \mid d \mid n \) and \( p \leq \sqrt{n} \)

which contradicts our first assumption.

Suppose \( d > \sqrt{n} \), then \( x \leq \sqrt{n} \),

Unless \( d = n \), there is a prime factor \( p \) of \( x \), with \( p \mid x \mid n \) and \( p \leq \sqrt{n} \)

which contradicts our first assumption.

Therefore, every divisor \( d \) of \( n \) is either 1 or \( n \) and so, \( n \) is prime.

The proof itself relies again on some hidden lemmas. Here is the formalized statement.

**Theorem primepropdiv:***

\[
(n: \texttt{nat})(\text{gt } n (1)) \Rightarrow
((q: \texttt{nat})(\text{Prime } q) \Rightarrow (\text{Divides } q n) \Rightarrow
(\text{gt } (\text{mult } q q) n) \Rightarrow
(\text{Prime } n)).
\]

The \texttt{modprime} module contains some results about modulo arithmetic where the modulus is prime. The combination of the \texttt{modprime} and \texttt{order} modules could be replaced with additional effort by more abstract group theory modules. The \texttt{fermat} module contains Fermat’s little theorem.
5.2. A FORMAL PROOF OF POCKLINGTON’S CRITERION

In informal mathematics the lemma known as ‘Technical Lemma 3’ or ‘tlemma3’ (Lemma 5.2.3) would not need a proof. It follows trivially from the fundamental theorem of arithmetic (every number has a unique prime factorization). However, a formalized proof of this theorem is not available, and formalizing it would be more work than Pocklington’s criterion itself.

Lemma 5.2.3 (‘tlemma3’) If \(0 < a < b\) and \(a | b\), then there is a prime factor \(q_i\) of \(b\) such that \(a \mid (b/q_i)\).

Proof:

Suppose \(0 < a < b\) and \(a | b\), say \(b = ax\).

Induction on the number of prime factors in \(b\).

(Base case)

Suppose \(b\) itself is prime. Then \(a = 1\), choose \(q = b\).

(Step case)

Suppose the statement holds for \(b\)'s with \(k\) prime factors, suppose \(b\) has \(k + 1\) prime factors.

Take a prime factor \(q_i\) of \(b\).

If \(q_i | a\), then \(a = q_i a', b = q_i b'\), by induction there exists a \(q\) such that \(a' | (b'/q_i)\).

If \(q_i \nmid a\), then \(q_i | x\), so \(a | (ax/q_i)\), so choose \(q = q_i\).

\[\square\]

The formal statement corresponding to Lemma 5.2.3 is:

Lemma tlemma3:

\[(\text{qlist} : \text{natlist}) (a, b : \text{nat}) (\text{lt O a}) \rightarrow (\text{lt a b}) \rightarrow\]

\[(\text{Divides a b}) \rightarrow \text{b} = (\text{product qlist}) \rightarrow (\text{allPrime qlist}) \rightarrow\]

\[(\text{EX q} : \text{nat} | (\text{inlist nat q} \text{ qlist}) /\)

\[(\text{Divides a} (\text{multDrop q} \text{ qlist})).\]

Each element in \(\mathbb{Z}_p^*\) has an order. In the informal version of the proof, order is a function. However, since we formalized \(\text{Order}\) as a relation, we have to prove this as a lemma.

Lemma order_ex: \((b : \mathbb{Z}) (p : \text{nat}) (\text{Prime p}) \rightarrow \neg(\text{Mod b } \text{’0’} \text{ p}) \rightarrow\]

\[(\text{EX x : nat} | (\text{lt x p}) /\ (\text{Order b x p})).\]

Consider line 4 of the informal proof of Pocklington’s criterion. In order to show that \(q\) is the order of \(b\) in \(\mathbb{Z}_p^*\), a contradiction is derived from the assumption that \(b^{\frac{x}{q_i}} \equiv 1 \pmod{p}\) for some prime factor \(q_i\) of \(q\). This follows from a number of non-trivial technical lemmata such as:

Lemma 5.2.4 (‘orderdiv’) Let \(p\) be a prime number, let \(x, y \in \mathbb{Z}_p^*\). If \(b\) is the order of \(x\) in \(\mathbb{Z}_p\) and \(b^y \equiv 1 \pmod{p}\), then \(x | y\).
Proof:
Suppose \( b \) is the order of \( x \) in \( \mathbb{Z}_p \), and suppose \( b^y \equiv 1 \pmod{p} \).
Say \( y = qx + r \) with \( 0 \leq r < x \), then
\[
\begin{align*}
  b^y &= b^{qx+r} = b^{qx}b^r \\
  \quad &\equiv b^r \pmod{p}
\end{align*}
\]
So \( b^r \equiv 1 \pmod{p} \), so \( r = 0 \) because \( x \) is the smallest element in \( \mathbb{Z}_p^* \) with \( b^x \equiv 1 \pmod{p} \). So \( x|y \).

The formal version of this lemma:

\[
\text{Lemma order\_div:} \\
(b:Z) (x,p:nat) (Order b x p) \rightarrow \\
(y:nat) (lt O y) \rightarrow (Mod (Exp b y) '1' p) \rightarrow (Divides x y).
\]

Finally, a technical lemma that is explicitly mentioned in the informal proof is Fermat’s little theorem. The common proof of Fermat’s little theorem found in any introductory Algebra textbook uses binomial coefficients. Formalization of this proof requires building a theory of summation of sequences and of binomial coefficients. We use a different proof which, although also requiring building of some mathematical theories, seems easier to formalize in Coq than the binomial coefficient style proof.

Theorem 5.2.5 (‘flt’) For all \( a \in \mathbb{Z} \) we have \( a^{p-1} \equiv 1 \pmod{p} \), when \( p \) prime.

Proof:
Let \( a \in \mathbb{Z} \) and \( p \) prime.
Let \( x = 1 \cdot 2 \cdots (p-1) \).
Then \( (a \cdot 1) \cdot (a \cdot 2) \cdots (a \cdot (p-1)) = a^{p-1}x \).
Since multiplying with \( a \) in \( \mathbb{Z}_p \) is injective,
the factors \( a \cdot 1, \ldots, a \cdot (p-1) \) are a permutation
of \( 1, \ldots, p-1 \), so \( x \equiv a^{p-1}x \pmod{p} \).
Therefore, \( a^{p-1} \equiv 1 \pmod{p} \).

The main reason for using technical lemmata in the construction of the proof of Pocklington’s criterion is to capture high level reasoning. In fact, the steps in the informal proof are large steps. The full formalization of Pocklington’s result gives us a way to generate proofs of primality that are formal and acceptable by the most skeptical approach.

5.3 Generating Proofs of Primality

This section describes how Pocklington’s Criterion can be used to produce a formal and efficient primality proof for a relatively big prime number. In a skeptical approach one
invokes an outside oracle to supply the theorem prover with the necessary witnesses for applying Pocklington’s criterion. For instance, computer algebra systems can act as oracles when algebraic equalities have to be verified. For example, when \( a \equiv b \pmod{n} \) needs to be proved the computer algebra system can provide \( q \in \mathbb{Z} \) such that \( a = b + qn \).

For the skeptical approach to work, a computer algebra system must be able to supply both a fast decision for the primality of a positive number \( n \) and in the affirmative case the ability to provide additional extra information for building a proof-object.

To realize the automatic generation of primality proofs exploiting computer algebra systems to provide the necessary witnesses, we implement a small Java applet which can communicate both with a computer algebra oracle and with Coq. Figure 5.2 gives an overview of the overall architecture. Note that it is not essential that the control application runs on a client machine different from the servers. However, implementing the control in a Java applet accentuates the possibility of remote computing.

The applet allows the user to input a number \( n \) and generates (if \( n \) is prime) Coq tactics that show the primality of \( n \). The proof described by these tactics applies Pocklington’s criterion to the correct instantiation of the parameters. The parameters are witnesses retrieved from the computer algebra system. Communication with the computer algebra system takes place using OpenMath, therefore any OpenMath compliant computer algebra system can be used. The generated tactics are sent to Coq and a formal proof-object is returned.

Communication is implemented by building a ‘shell’ around GAP and Coq, which connects the standard input and output of these programs to a TCP socket. This means that these programs may run on different machines somewhere on the Internet, and our applet can access them remotely. This is not the most elegant way to achieve interoperability between systems. Essentially we use an interface that was meant for communication with a human user to connect the systems. It would have been better if both systems had some sort of OpenMath interface for the purpose of machine-machine com-
5.3.1 Computer Algebra Oracles

The CAS is expected to support the following functionality:

- A fast primality check algorithm. If the number $n$ is not prime the applet refuses to generate a tactic script.
- An algorithm for computing the primitive root modulo $n$.
- An extended Euclidean GCD algorithm. Given $x$ and $y$ this computes $d \in \mathbb{N}$ and $\alpha, \beta \in \mathbb{Z}$ such that $d = \alpha x + \beta y$.
- An integer factorization algorithm.
- An algorithm for computing powers of numbers modulo $n$.

Most general purpose CA systems have these algorithms natively. Note that we only use the integer datatype of the CAS. Although this is a rather simple type in the CA world, our applet has to take into account that these are integers of arbitrary length, therefore the standard `int` Java type does not suffice.

5.3.2 Pocklington as Interactive Document

The applet can be seen as a prototype interactive mathematical document. Both the connection to the CAS and to Coq are examples of external interaction, see Chapter 4. As to internal interaction: The user can influence the presentation of the document in several ways. The prime number $n$ can be set by the user, there are several computer algebra systems to choose from, and the user can specify to which extent the theorem prover should trust the witnesses provided by the computer algebra oracle. The latter choice is made by selecting a belief level ranging from 1 to 5.

1. Believe nothing. All generated subgoals require proofs.
2. Believe simple modular equations. As above, but modular equations of the form $a \equiv b \pmod{n}$, that are proved directly by finding a witness, are assumed as axioms instead.
3. Believe modular equations. As above, but also modular equations of the form $a^m \equiv b \pmod{n}$, that are proved using the divide and conquer method outlined below, are assumed as axioms instead.
4. Believe modular equations and linear combinations. As above, but also subgoals of the form $\exists \alpha, \beta. (\alpha x + \beta x \equiv c \pmod{n})$ are assumed as axioms.
5.3. GENERATING PROOFS OF PRIMALITY

5. Believe everything. Any subgoal may be assumed as an axiom.

In principle the proof-object returned by Coq could be parsed and presented using the CoqViewer tool developed in Chapter 4 leading to a real interactive mathematical document in the sense of [27]. However, the proof uses large parts of the ZArith library, which uses special syntax rules which cannot currently be parsed by the CoqViewer tool.

5.3.3 The Algorithm

Now we describe in detail the algorithm based on Pocklington’s criterion which reduces the proof of primality of \( n \) to the proof of a number of algebraic equalities. The PocklingtonC algorithm summarized in Figure 5.3 takes as input a candidate prime number \( n \) and produces a Coq tactic script for constructing a proof-object for \( \text{Prime}(n) \). Interaction with computer algebra oracles takes place mostly in Step (1), where the witnesses are found, and (4), where the algebraic identities are shown.

When the computer algebra software is given a positive number \( n \), it first tests whether the number is indeed prime. If not, it returns the number. If the number \( n \) is prime, then the CAS can easily compute the numbers \( a, q, \) and \( m \) as follows. For \( a \) take the primitive root \((\text{mod } n)\), namely an element \( a \) such that \( a^{n-1} = 1, (\text{mod } n) \) and \( a^i \neq 1, (\text{mod } n) \) for \( i = 1, \ldots, n - 2 \). For \( q \), consider the prime factorization \( n - 1 = q_1 \ldots q_t \), where \( q_1 \geq \ldots \geq q_t \), and take \( q = q_1 \ldots q_t \) for the smallest \( t \) such that \( q \geq \sqrt{n} \). Finally, for \( m \) take \( m = (n - 1)/q \). All the operations to compute the appropriate \( a, q = q_1 \ldots q_t, \) and \( m \) are carried out by the computer algebra package upon receiving the prime number \( n \). Notice that these witnesses, computed as described, satisfy the hypotheses of Pocklington’s criterion. Subgoals (4)(a) and (4)(c) are clearly true. Condition (4)(b) is true because \( n \) is prime and \( \gcd(a^{n-1} - 1, n) \) cannot be \( n \). If it was \( n \), then \( a^{\frac{n-1}{q}} \equiv 1 \pmod{n} \) for an exponent \( \frac{n-1}{q} < n - 1 \). However, this is not possible because \( a \) is the primitive root \((\text{mod } n)\). With this choice for \( a, q, \) and \( m \) Pocklington’s criterion is “complete”, i.e. it works for any prime number \( n \).

The computer algebra oracle is also called in Step (4)(b). It computes the coefficients for the linear combinations generated by the gcd proof obligations using a straightforward extension of the Euclidean gcd algorithm. Most computer algebra systems provide this algorithm as primitive. The oracle also computes the result of the exponentiations for \( a^{n-1} \pmod{n} \) and \( a^{n-1} \pmod{n} \) and for all the intermediate steps in the divide and conquer procedure. Intermediate obligations are of the form \( x^m \equiv y \pmod{n} \) where all variables are concrete instances such that \( x, y \leq n \). Although finding the witness \( z \in \mathbb{Z} \) such that \( x^m = y + z \cdot n \) is easy for the computer algebra system, the computations involved in proving the equality directly are too expensive for Coq as \( x^m \) gets large. Instead, the goal is changed by replacing the exponent \( m \) as follows.

\[
x^m \equiv y \pmod{n} \iff \begin{cases} 
x^{\frac{m}{2}} \equiv z \pmod{n}, \quad zz \equiv y \pmod{n} & \text{if } m \text{ even} \\
x^{\frac{m-1}{2}} \equiv z \pmod{n}, \quad xzz \equiv y \pmod{n} & \text{if } m \text{ odd}
\end{cases}
\]
PocklingtonC \((n; T)\)

**Input:** \(n\) a prime number.

**Output:** \(T\) a tactic script for proving primality of \(n\) by Pocklington’s criterion.

1. **[Find witnesses.]**
   - Let \(a\) be the primitive root mod \(n\). Choose \(q\) and \(m\) such that \(n = qm + 1, q \geq 0, m \geq 0\). Compute the prime factorization of \(q\) in \(q_1 \cdot \ldots \cdot q_t\).

2. **[Recursion Step]**
   - Apply recursively PocklingtonC \((q_i; S_i)\) for \(i = 1, \ldots, t\) to every prime factor \(q_i\) in the factorization of \(q\), thus obtaining tactic scripts \(S_1, \ldots, S_t\).

3. **[Apply Pocklington’s]**
   - Apply Pocklington’s criterion using the parameters \(a, q, q_1, \ldots, q_t,\) and \(m\) in order to prove \(\text{Prime}(n)\).

4. **[Prove the subgoals]**
   - Provide the tactic scripts \(S_a, S_b,\) and \(S_c\) for proving the subgoals corresponding to the hypotheses of Pocklington’s criterion.
     - \((a)\) \(a^{n-1} = 1 \pmod{n}\) is shown by a divide and conquer strategy in which the exponent gets smaller until the computation is trivial.
     - \((b)\) \(\gcd(a^{\frac{n-1}{q_i}} - 1, n) = 1, i = 1, \ldots, t\) is shown by proving that 1 is a linear combination of \(a^{\frac{n-1}{q_i}} - 1\) and \(n \pmod{n}\).
     - \((c)\) \(n \leq q^2\) is shown trivially.

3. **[Output]** Assemble the tactic scripts \(S_1, \ldots, S_t, S_a, S_b,\) and \(S_c\) in the tactic script \(T\) for proving \(\text{Prime}(n)\).

---

Figure 5.3: Pocklington’s Criterion Algorithm
5.3. GENERATING PROOFS OF PRIMALITY

The computer algebra oracle is used to compute $z$ such that $0 \leq z < n$. The resulting goal involving $x$ is solved by recursively applying this procedure, the other goal can be proved directly as all numbers are small. Note that this solution again relies on the computer algebra oracle to find witnesses.

Figure 5.4 shows a run of the algorithm for $n = 1999$. The applet communicates $n$ to the CAS which returns the witnesses $a = 3, q = 111, m = 18$, as well as the prime factorization of $q$: $\langle 3, 37 \rangle$. The applet knows which goals the TP will consecutively need to prove and provides tactics to solve them. When the goals $\gcd(a^{n-1}/n - 1, n) = 1$ are encountered, the CA is consulted again to provide the necessary $\alpha$ and $\beta$ ($\alpha = 270, \beta = 109$ for $q_i = 3$). At some point the TP will need to prove that the prime factors of $q$ are really prime. This is when the applet applies Pocklington recursively.

To summarize the overall picture, the algorithm PocklingtonC can be used to produce a Coq tactic script that generates a proof-object for the primality of a positive number. The only requirement on the computer algebra systems used as oracles is the ability to perform integer computations like prime testing, factorization, $\gcd$ computation and some modular arithmetic. Since the communication uses the OpenMath standard, the architecture allows for multiple computer algebra oracles, see Figure 5.2. The tactics view of the applet presents the generated tactic script to the user, see Figure 5.5.

All responses of Coq can be predicted, so the tactic script can be composed without
consulting Coq. Once the script is generated, the user can send it to Coq which returns a proof-object. The proof-object is presented in the proof-object view, see Figure 5.6.

5.4 Results

Our implementation of the architecture described above consists of a Java application in which the user enters a positive integer \( n \) and selects a computer algebra package running on a remote server. If the number is prime, the computer algebra package is repeatedly invoked for a concrete value of \( n \) and for the subsequent recursive calls of the factors. The application then generates a Coq tactic script that can be sent automatically to Coq when proving the goal \( \text{Prime}(n) \).

In practice, the algorithm outlined in Section 5.3 has to take into account limitations on the size of the prime number \( n \). Computer algebra software, like GAP, is able to test primality for integers up to 13 digits long. For bigger integers, the primality test are probabilistic and return a probable prime. For instance, testing numbers with several hundreds digits is quite feasible in GAP4 using \( \text{IsPrimeInt} \) or \( \text{IsProbablyPrimeInt} \). Concerning factorization, \( \text{FactorsInt} \) is guaranteed to find all factors less than \( 10^6 \) and will find most factors less than \( 10^{10} \).

Computing on the Coq unary \texttt{nat} type is very slow. So, we use the binary represented integers from the \texttt{Zarith} library. This requires a reformulation of Pocklington’s theorem, and of course the two formulations have to be proven equivalent. This results in the following formalization of the criterion:

\[
\text{Theorem Zpocklington:} \\
(n, q, m : \mathbb{Z})(a : \mathbb{Z})(\text{qlist} : \text{Zlist})
\]
5.4. RESULTS

Figure 5.6: Proof-object view of applet.

\[ n > 1 \rightarrow 0 < q \rightarrow 0 \leq m \rightarrow n = q \cdot m + 1 \rightarrow q = (\text{zproduct } \text{qlist}) \rightarrow (\text{allZPrime } \text{qlist}) \rightarrow (\text{ZMod } (\text{ZExp } a \ 'n-1') \ '1' \ n) \rightarrow (\text{ZallLinCombMod } a \ n \ m \ n \ \text{qlist}) \rightarrow n \leq q \cdot q \rightarrow (\text{ZPrime } n) . \]

We tested the generated tactic scripts for all primes between 2 and 7927 (the first 1000 primes) and measured the run time needed by Coq to produce and check the proof-object on a Unix workstation.\(^1\) Obviously the general trend is that larger primes need more time. However, some numbers are much harder due to an unfortunate prime factorization of \(q\). Some examples of extremely easy and extremely hard primes are given in Table 5.1.

For example when proving \(\text{Prime}(2039)\), the algorithm is forced to choose \(q = 1019\), since the prime factorization of 2038 is \(2 \cdot 1019\). Now, 1018 in its turn has as prime factorization \(2 \cdot 509\). In the end recursive calls for 3, 7, 127, 509, and 1019 are needed to prove \(\text{Prime}(2039)\), and it takes about 17 seconds to verify the proof. In contrast to verify the proof generated for \(n = 2939\) only needs recursive calls for 3, 7, and 113 and only takes about 9 seconds. The number of primes required by the algorithm in the recursion step for the first 1000 primes are plotted in Figure 5.7. Note that numbers of the form \(2^n + 1\) require no recursive call (they are 3, 5, 17, 257, \ldots). Next best are primes

\(^1\)The machine used for all tests in this chapter is a Sun Ultra 10 with 333Mhz Sparc processor and 128MB memory. The Coq version used is 6.3.1 (native code version).
Table 5.1: Easy and hard primes with Coq timings (in seconds).

<table>
<thead>
<tr>
<th>n</th>
<th>Recursive calls</th>
<th>Believing</th>
<th>Skeptical</th>
</tr>
</thead>
<tbody>
<tr>
<td>2939</td>
<td>3, 7, 113</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>4111</td>
<td>17, 137</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>7829</td>
<td>17, 103</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>2039</td>
<td>3, 7, 127, 509, 1019</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>4079</td>
<td>3, 7, 127, 509, 1019, 2039</td>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>7727</td>
<td>3, 193, 1931, 3863</td>
<td>3</td>
<td>19</td>
</tr>
</tbody>
</table>

of the form $p + 1$ where the factorization of $p$ involves primes of the form $2^n + 1$ above (they are 7, 11, 13, 19, 37, 41, ...), and so on. Based on the observations in Figure 5.7 we conjecture the following lemma about the complexity of the algorithm.

**Lemma 5.4.1** Given a number $n$, there are at most $\lceil \log_2 \frac{n+1}{3} \rceil$ primes to be checked recursively.

**Proof:**

In the worst case we are forced to choose $q$ large, i.e. $n - 1 = 2p$ with $p$ prime, and $p$ has the same problem.

Let $n_i$ be the value of $n$ in the $i$th recursive call, then

\[
\begin{align*}
  n_0 &= n \\
  n_{i+1} &= \frac{n_i - 1}{2}
\end{align*}
\]

The algorithm stops when $n_k = 2$ after $k$ recursive calls.
If we rename \( h_i = n_{k-i} \) then for \( i = 0, \ldots, k \):
\[
\begin{align*}
  h_0 &= 2 \\
  h_{i+1} &= 2h_i + 1
\end{align*}
\]
This recursive system is easily solved: \( h_i = 3 \cdot 2^i - 1 \).

We know \( h_k = n \), so \( n = 3 \cdot 2^k - 1 \), and therefore \( k = \log_2 \left( \frac{n+1}{3} \right) \).

The logarithmic trend also shows in the plot of the size of the context used in the primality proof (number of lemmata about modular equations, linear combinations, and primes) as given in Figure 5.8. The complete run time results for the first 1000 primes are plotted in Figure 5.9. Note that this the time needed by Coq to check the already generated script. The time needed by the applet (using the Gap CA oracle) is negligible compared to the Coq run time. The timings (in seconds) for the believing approach (belief level 4 on page 114, where subgoals to prove modular equations and linear combinations are not proved) and for the skeptical approach (belief level 1 on page 114, where all subgoals are proved) are given. The results in Figure 5.9 show that the current implementation significantly improves the implementation discussed in [26] and [28]. This is due to the fact that the current implementation avoids the unary represented numbers completely. The generated tactic script in [26] still used unary represented numbers in some places, and so the time complexity of the algorithm seemed to be worse than linear, while Figure 5.9 shows a logarithmic time complexity. The improvement over [28] is less dramatic and is achieved by using a version of Coq compiled to native machine code, whereas the version used in [28] was compiled to bytecode.
Some large primes that were tested are in Table 5.2. Actually, due to the size of these numbers, we ran into some practical problems. For example, we were forced to replace the Java integers in the applet by objects of the `BigInteger` class as the applet was suffering from integer overflows when talking to the CAS. Another overflow problem was encountered in the way Coq deals with lemmas whose name ends in a number $i$. The applet generates lemmas “prime$i$” which for large $i$ are not accepted by Coq (version 6.3.1). Changing the name to “prime$i$a” solves this problem. For primes with more than 13 digits GAP issues a warning that a probabilistic primality test is used. Some GAP functions used to generate the witnesses are non-probabilistic. In any case, the generated tactic scripts are accepted by Coq, therefore the numbers in Table 5.2 are really prime. The table lists timings for some “large” primes found in [51]. The times in the table are in seconds. This is the time needed for Coq to verify the tactics script that was generated by the applet with the help of the CAS.

In an earlier implementation, where `nat` was uses in some of the verifications, Coq would run out of stack space for primes in the 4-digit range. Once the implementation exclusively used the `ZArith` numbers, this memory constraint was lifted. We tested the current implementation with primes up to 44 digits.

### 5.5 Conclusions

We have shown how by combining computer algebra oracles and theorem provers it is possible to automatically produce proofs of primality that are efficiently and formally
Table 5.2: Some large primes with the time (in seconds) needed for Coq to verify the skeptical proof.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Digits</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>188888881</td>
<td>9</td>
<td>72</td>
</tr>
<tr>
<td>1234567891</td>
<td>10</td>
<td>87</td>
</tr>
<tr>
<td>1666666666661</td>
<td>13</td>
<td>153</td>
</tr>
<tr>
<td>88088088088080803</td>
<td>16</td>
<td>372</td>
</tr>
<tr>
<td>74747474747474747</td>
<td>17</td>
<td>275</td>
</tr>
<tr>
<td>111111111111111111</td>
<td>19</td>
<td>492</td>
</tr>
<tr>
<td>455666777888888999999</td>
<td>21</td>
<td>885</td>
</tr>
<tr>
<td>1234567891010987654321</td>
<td>22</td>
<td>519</td>
</tr>
<tr>
<td>122333444455555444433221</td>
<td>25</td>
<td>746</td>
</tr>
<tr>
<td>11777310696798643575324671</td>
<td>26</td>
<td>1250</td>
</tr>
<tr>
<td>15496731425178936435099327796097</td>
<td>32</td>
<td>2333</td>
</tr>
<tr>
<td>9026258083384996860449366072142307801963</td>
<td>40</td>
<td>4154</td>
</tr>
<tr>
<td>20988936657440586486151264256610222593863921</td>
<td>44</td>
<td>9750</td>
</tr>
</tbody>
</table>

verifiable. The primality proofs are obtained according to Pocklington’s criterion.

To formally prove Pocklington’s criterion in Coq, we needed to first prove about 260 smaller lemmata taking approximately 5000 lines of code. No matter how enjoyable, this is still a lot of hard work. A less skeptical approach that assumes many of the needed lemmata as axioms requires less effort and might still produce proofs that are acceptable in other communities.

The architecture for using Pocklington’s criterion relies on computer algebra oracles. We interpret these oracles as mathematical servers providing computational capabilities on the network and the Java application that produces the tactic script as a client to these servers. In this general view, our experiments are an example of how to use computer algebra in theorem proving and an investigation on the tools that are required to effectively carry out the integration. We profited greatly from our work in using the standard communication language OpenMath to interface to a variety of symbolic computation systems.

We tested the current implementation with primes up to 44 digits. It is not known what is the largest prime that can be tackled by our approach.
Chapter 6

Conclusion

6.1 Summary

This final chapter reflects on the results described in the preceding chapters. By finding similarities and differences we come to a conclusion about the viability and feasibility of the ideas put forward in this thesis. The main ideas can be summarized as:

- Combining Computer Mathematics systems, especially when TP systems are involved, requires a language which formally grounds the mathematical symbols that are exchanged.

- Type Theory, and more specifically the Calculus of Inductive Constructions, is a good candidate for such a language, as proofs are first class citizens.

- Interaction based on the structure of formal mathematical contexts makes proofs in interactive mathematical documents possible. This requires that content and presentation are separately stored in the document. Again, explicit proof-objects, such as present in type theoretical theorem provers, are very helpful.

- Proof automation is possible in type theoretical theorem provers through the reflection principle. However, the internal computations involved make this approach rather inefficient for statements involving many computations.

- In some cases efficient computations are possible in type theoretical theorem provers using a combination of CA oracles and internal computations.

Chapter 1 discussed the ideal mathematical workspace, which is the ultimate goal of the efforts undertaken in the new field of Computer Mathematics. In the work described in this thesis, we have certainly not implemented the ideal mathematical workspace. However we have some observations about the nature of mathematical reality and the activities of theorem proving and symbolic computing. Moreover we have performed and described some concrete formalizations of mathematical theories in type theory. Also some prototype tools were implemented which present mathematics encoded in
type theory and combine existing Computer Mathematics systems. The results of all this work sheds some light on what are the important issues that have to be solved before we can hope to see the ideal mathematical workspace become a reality.

In many places in this thesis, research was done through concrete example case studies and implementation of prototype tools. Often, when exploring a new field, such an empirical approach is the only possible way to find out what the real issues and problems are. There is however a danger that the results are not generally applicable and depend too much on implementation details and “today’s technology”. However, it is safe to say that despite the implementation details, the conclusions in this chapter are general. This is for a large part due to the type theoretical nature of the tools that have been used. Type theory is the formal vehicle for both mathematics and logic in our formalizations and prototype tools. The formalization case studies are done internally in the Coq object language, as Chapter 3 and Chapter 5 show. The prototype tools are external to the theorem prover. The tools are implemented in Java, see Chapters 4 and 5. The implementation work relies on intimate details of type theory, more specifically the calculus of inductive constructions. But we do not rely on the actual implementation of type theory in Coq. Type theoretical theorem proving is “open”: the De Bruijn criterion ensures that it is in principle easy to construct one’s own independent type checker. The De Bruijn criterion applies also to presentation tools, and to tools for connecting type theory based TPs to other systems. This means that creating such tools can be done independently from the implementation of the TP. For instance the CoqViewer tool described in Chapter 4, with some minor modifications, applies to other type theoretical theorem provers, not just the Coq system. The algorithms do not depend on the internal datatypes of Coq. Such implementations are relatively easy due to type theory’s universality, its conciseness and the fact that proof-objects are first class citizens in this language. We conclude that it is thanks to TT that this works, and not despite TT.

In Chapter 1 we noted that the quest for the ideal mathematical workspace essentially boils down to finding a good formal language on which we can base mathematics. Not only should this formal language be expressive and powerful in a logical sense, but it should also allow efficient computations and be presentable in a notation resembling informal mathematics. A necessary low level condition for such a language is that mathematical expressions can be transported from one application to another in a standard way. Attempts at such a standard have been undertaken, for example in the form of the OpenMath language. However, we argue that a real mathematical formal language based on logic, with a well understood semantics, is necessary to successfully combine all forms of Computer Mathematics. Type theory, and more specifically the calculus of inductive constructions, which is discussed in Chapter 2, is a good candidate for this formal language. The inductive types in CIC make efficient computations feasible. Since proof-objects are first class citizens in CIC, we can present proofs elegantly and interactively, as shown in Chapter 4. The results in Chapter 4 can be summarized in the following “slogan”:

Interactive Mathematical Documents = Formal Mathematics + Presentation
6.1. SUMMARY

In general, using a formal language to describe mathematical content provides an extra level. Presentation and sharing of the content in a more symbolic language such as OMDoc is still possible. A striking overall similarity between the chapters in this thesis is that in each of the chapters we seem to need such an extra level in order to connect other levels:

- In Chapter 1 we need a **formal mathematics level** in addition to **Plato’s Heaven** and the **document level**. The document level forms the user interface to the ideal mathematical workspace. A direct connection to Plato’s heaven is not possible, as computers need syntax.

- In Chapter 2 we need a **typing level** to make clear, in a concise way, which terms on the **pseudo expressions level** are well formed. A two-level definition of the calculus makes for a very concise definition, which naturally adheres to the De Bruijn criterion.

- In Chapter 3 we need a **syntactical level** form in addition to the **boolean level** bool where we can compute, and the **propositional level** Prop for representing statements. The user should specify propositions on this syntactical level so that it can be translated to an internal computation as well as to a proposition. The extra level allows to use internal decision procedures which are proven correct.

- In Chapter 4 we need a **content level** in addition to the **presentation level**. The distinction between content and presentation is essential for our notion of interactive mathematical document.

- In Chapter 5 we need a **control level**, in the form of an external applet, in addition to the CA and TP levels. By using an extra level, both the CAS and the TP can be approached as back engines for our services. Even though the Pocklington criterion example yields a result in the TP world, and the CA is only used as an oracle, the extra level grants a more general architecture in which CA and TP systems can be combined.

Three different kinds of computer mathematics systems are considered: theorem provers, computer algebra systems and systems that take care of the presentation of mathematics. These different systems have distinct ways of formalizing mathematical content. Yet, if these systems are to be combined into the ideal mathematical workspace, they have to speak the same language. In the ideal mathematical workspace our formal language \( \mathcal{L} \) could be used. However, in order to combine the different CM systems of today such a formal language is too restrictive. In Chapter 4 and Chapter 5 we look at languages specifically designed for combining computational systems: OpenMath and OMDoc. These languages are more abstract than for instance the type theoretical languages CC and CIC. There are fewer restrictions on the well-formedness of expressions which makes OpenMath and OMDoc suited for combining systems.
The formalized world in the TP is implemented through the object language of the logical system underlying the TP. Potentially all of mathematics can be reduced to this object language. Unfortunately, a drawback of the TP formalized world implementation is that everything must be expressed inside this object language. Many assumptions about symbols are made explicit. The general problem is that mathematicians are forced to make implementation choices as they formalize parts of mathematics in the formal language of a theorem prover. A second drawback is that computational algorithms have to be encoded in this object language, unless the computations are done on the meta level using tactics. The resulting computations are not particularly efficient as the encoding gets in the way.

CASs perform computations on the objects in the world of formalized mathematics. Theorem provers verify statements about the objects in the formalized world. We have already noted that CA and TP work on very different implementations of the formalized world. The CA formalized world is implemented by the programmer of the CAS through a number of datastructures. The datastructures are specifically geared towards efficient computations, and in general CASs there is no formal way inside the system to reason about properties of the datatypes or the algorithms defined on them. This way of implementing mathematical notions has a serious disadvantage in that the system can only deal with the concepts that the designer of the CAS has put in.

The results of the preceding chapters are arranged into the following three categories, and discussed in more detail.

1. Formalizing mathematics in type theory.
2. Computations and proofs.
3. Presentation of formal mathematics.

6.2 Formalizing Mathematics

This activity of formalizing mathematics in type theory is primarily discussed in Chapter 2, where technical details are rendered of representing mathematics in the calculus of constructions and the calculus of inductive constructions, and in Chapter 5, where we formalize a non-trivial informal mathematical proof and the theory underlying the proof. Our concrete formalization in Chapter 5 leads to some observations on how to perform such a formalization, and what is missing in Coq.

In order to be able to reason about concrete formalizations of parts of mathematics, Chapter 1 introduces a candidate universal language $L$ which is based on typed $\lambda$-calculus.

We discuss *types* in programming languages and theorem provers in Chapter 1. Type theory has proven to be quite useful in programming languages as it can be used to detect mistakes the programmer makes. In type systems for programming languages, the programming language can be seen apart from the type assignment. In type theoretic
6.3. COMPUTATIONS AND PROOFS

Theorem provers, the types are an integral part of the language. The type system is used
to define the language. We have shown that type theory has some of the properties
we need in a universal language for mathematics: it is powerful, it conforms to the De
Bruijn criterion and is thus trustworthy, and computations are possible.

The Calculus of Constructions is the first concrete system we present. Since it is a
Pure Type System, we know that it has a number of meta-theoretical properties. The
CC is extremely powerful yet very simple, and it adheres to the De Bruijn criterion in
a natural way. Complex datastructures are possible through impredicative encodings.
However, this is inefficient. So, although CC is probably the best system with respect to
the De Bruijn criterion, we should look for an alternative.

When inductive types are added to the Calculus of Constructions, the Calculus of
Inductive Constructions is obtained. The new primitives have their own reduction and
typing rules. The new rules allow the user to define recursive functions in a natural
way, and these recursive functions can be evaluated in an efficient manner. However,
the side conditions in the type system get very complex. Still, representing mathematics
can be done efficiently and elegantly in the CIC. We show how to do logic, predicates,
equalities and computations in the CIC.

Although we do not provide many detailed examples of how to formalize mathe-
matics in CIC in Chapter 2, Chapter 5 develops a non-trivial case study by formalizing
Pocklington’s criterion. To formally prove Pocklington’s criterion in Coq, we needed
to first prove about 260 smaller lemmata taking approximately 5000 lines of code. No
matter how enjoyable, this is still a lot of hard work. A less skeptical approach that
assumes many of the needed lemmata as axioms requires less effort and might still pro-
duce proofs that are acceptable in other communities.

Also we conclude in Chapter 5 that not many results are in the currently available
standard libraries, and using previously formalized results is difficult due to many in-
terface problem. Of course this is not just a formal mathematics problem, but a general
one of the Software Engineering world at large. It is not clear whether the problems
in formal mathematics are more complex than Software Engineering problems (for ex-
ample because mathematics has been around longer, the specifications in the form of
informal proofs, suffer more from “legacy” problems) or that they are easier (in formal
mathematics much more powerful mathematical tools are available). It seems that in
informal mathematics, mathematical theories depend heavily on other mathematical
theories, which makes full formalization time-consuming.

6.3 Computations and Proofs

The act of computing forms an important mathematical activity. Often computations
and reasoning are used in the same proof, and go hand in hand. Think, for example, of
algebraic proofs containing equational reasoning. Therefore a mathematical assistant,
even one geared towards logical proofs, should facilitate computations. In principle,
computations can be mimicked by reasoning steps, which is what happens when tacti-
cals, or other external computations, are used to implement decision procedures. In our type theoretical setting, this is often not acceptable as proof-objects get rather large, and no longer reflect the reasoning of the proof.

One solution to this problem would be to keep using external computations, but prove the algorithms correct instead of mimicking the computation with proof steps. The real problem is that the TP can only say something about objects formalized in the object language of the theorem prover, not about external algorithms in for instance a CAS.

Another solution, which is applied in TT, is to use Poincaré’s principle, which enables internal computations inside the TP. Poincaré’s principle states that computations do not require proofs, which is implemented in type theory through the conversion rule which states that convertible types have the same inhabitants. Efficient computations are possible due to inductive types in the CIC. Therefore, even though everything is done in the object language, proof automation leading to small proof-objects is still possible. Based on this principle we propose to implement proof automation. This is done via the reflection principle which we investigate in Chapter 3. The resulting proof-objects are small because of the conversion rule. However, for large and complicated computations the efficiency leaves much to be desired. Because algorithms need to be terminating (and the TP has to be able to check this syntactically) it is hard to specify efficient algorithms because all algorithms have to be specified according to strict syntactical rules.

In Chapter 5 we take a different approach to computations in proofs. We show how by combining computer algebra oracles and theorem provers, one can have ‘impeccable’ results in an efficient way, as suggested in [9]. This is demonstrated in Chapter 5 by a generator which automatically produces proofs of primality that are efficiently and formally verifiable. The primality proofs are obtained according to Pocklington’s criterion.

The architecture for using Pocklington’s criterion relies on computer algebra oracles. We interpret these oracles as mathematical servers providing computational capabilities on the network and the Java application that produces the tactic script as a client to these servers. In this general view, the experiments in Chapter 5 are examples of how to use computer algebra in theorem proving and an investigation on the tools that are required to effectively carry out the integration. We profited greatly from our work in using the standard communication language OpenMath to interface to a variety of symbolic computation systems.

We conclude that the reflection method should be used in special cases, where computations are needed to make boring and tedious work disappear. Real intensive computations should be done outside of the theorem prover, if possible. In some special cases it is possible to generate a computationally easy certificate which the TP can then verify internally. This is essentially what is done in Chapter 5.
6.4 Presentation of Mathematics

In Chapter 4 we developed a method to create interactive mathematical documents based on formal mathematics developed in the Coq theorem prover. The resulting documents are specified in the OMDoc language, or can be viewed in our prototype CoqViewer tool. The formal structure of the mathematics is still present in the OMDoc document, but it can be presented as informal mathematics to the end user. The presence of formal structure in the document allows internal interaction, a form of interaction based on structured content. For example, using the fact that proof-objects are treated as first class citizens in the type theory of Coq, the reader can interact with the proofs in the document by adjusting the level of detail.

To this end Chapter 4 introduces a prototype tool, called CoqViewer. The tool can be used for authoring and presenting mathematical content based on type theory. The Coq system is used to create an initial mathematical theory. Using the tool, an author can enhance the definitions and proof-objects that form a formally derived mathematical context with presentation information.

In this implementation many new technologies are used; Java, for instance, is used as the implementation language. The Coq-to-OpenMath encoder in [24] is used for the formal mathematical terms. For the proofs we first generate MathMetaText objects. The natural language generation is based on Coscoy’s translation [34, 33] of proof-objects. It is not a very sophisticated verbalization algorithm, but the results look promising. The OMDoc Java package we implemented follows closely the structure of the OMDoc DTDs, it consists of about 20 classes describing all the OMDoc elements. The use of Java and the various XML applications makes it easy to share code with other authors. We use, for example, the OpenMath library from the PolyMath group [80], we use the XSLT transformation program and other XML4J software from IBM, we use stylesheets developed for OMDoc by the Omega project group (although the stylesheets need more work to enable folding and unfolding of proofs, but this should be easy), we are investigating using the technology developed in the HELM project [4] which may make us less dependent on Coq specific details.

The terms representing the mathematical content are implemented inside the tool as pointer trees. References are used to indicate bound variables. This causes some non-trivialities in implementing the usual algorithms such as copying, checking syntactical equivalence and type inference.

Type inference is needed in order to implement the natural language view. Because it can be assumed that the input to the tool is a checked mathematical context, some side conditions are not checked during reduction and type inference.

We applied the technology from Chapter 4 to some small examples found in Chapters 4 and 5. In future work we may apply it to some larger examples. Further study could make the intermediate level of MathStatements more subtle, maybe by drawing some inspiration from Nederpelt’s Pseudo Type Theory [68] or the proof-planning provers such as Omega [15]. Both of these seem to provide a level between formal logic based mathematics and informal natural language, although they are more interested...
in the opposite direction, i.e. going from informal to formal. which seems to provide a level between formal mathematics and informal natural language. For longer term future work we would like to investigate the reverse translation from OMDoc to Coq. It is unknown how difficult this is.
Bibliography


[22] J. Calmet and K. Homann, *Classification of communication and cooperation mechanisms for logical and symbolic computation systems*, in Proceedings of Frontiers of


Samenvatting

In dit proefschrift wordt gekeken naar *wiskundige stellingbewijzers* (theorem provers). Dit zijn computerprogramma’s die een wiskundig document kunnen controleren op fouten en ongerijmdheden. Vaak bieden ze een auteur ook mogelijkheden om zo’n document tot stand te laten komen.

Alle wiskundige stellingbewijzers zijn gebaseerd op een of andere logische taal. Het is noodzakelijk dat de te controleren wiskunde is *geformaliseerd* in zo’n logische taal. Omdat de computer de inventiviteit van een menselijk wiskundige mist, moeten de documenten uitgespeld worden tot in de fijnste details. De computer beschikt echter over een aantal eigenschappen die hem uiterst geschikt maken voor zijn verificatietaak: Hij is precies, onvermoeibaar, snel, betrouwbaar en klaagt niet.

In dit proefschrift wordt voor de logische taal de zogenaamde *typentheorie* gebruikt. Informele wiskundedocumenten worden direct gecodeerd in typentheorie, inclusief de redeneringen in de bewijzen. Wiskunde, gecodeerd in typentheorie, die gecontroleerd is door de computer is volledig correct, mits het verificatieprogramma correct is. Gelukkig kan de kern van zo’n programma relatief klein blijven, zodat handmatige controle van de correctheid mogelijk is. Men zegt wel dat typentheoretische stellingbewijzers voldoen aan het De Bruijn criterium.

Het formaliseren van informele wiskunde blijkt nog niet zo makkelijk te zijn. Er moeten veel details gegeven worden, waardoor vooral formele bewijzen erg groot kunnen worden. Vaak gaat het om grote aantallen relatief triviale stappen. Bovendien moeten er tijdens het formaliseren allerlei implementatiekeuzes gemaakt worden die grote gevolgen kunnen hebben voor de structuur van het document. Verder is er een groot aantal onopgeloste problemen op het gebied van het hergebruik van en communicatie met geformaliseerde wiskunde tussen stellingbewijzers en andere computerwiskundesystemen.

Het proefschrift richt zich op de volgende drie problemen: het automatisch bewijzen van tautologieën door middel van berekeningen binnen bewijzen, de interactieve presentatie van wiskundige documenten nadat ze geformaliseerd zijn, en de communicatie tussen stellingbewijzers en andere computerwiskundesystemen zoals bijvoorbeeld computeralgebrasystemen. Daartoe zijn de volgende hoofdstukken opgenomen in het proefschrift.

Hoofdstuk 2 behandelt de theorie achter stellingbewijzers. Specifieker wordt de *Calculus of Inductive Constructions* behandeld. Dit is de typentheorie die in de populaire
Coq stellingbewijzer gebruikt wordt. Dit hoofdstuk toont aan dat veel wiskundige begrippen elegant gecodeerd kunnen worden in de Calculus of Inductive Constructions. Het hoofdstuk maakt het proefschrift "self-contained" en legt een basis voor de andere hoofdstukken.

Hoofdstuk 3 richt zich op berekeningen binnen bewijzen. Hoewel het doen van (symbolische) berekeningen tot de hoofdactiviteiten van een wiskundige behoort, maken berekeningen geen deel uit van een in typentheorie gecodeerde logische redenatie. In dit hoofdstuk wordt dit gegeven gebruikt om triviale redeneerstappen te automatiseren door middel van een principe dat reflectie genoemd wordt. Via een aantal case studies wordt gedemonstreerd hoe dit principe werkt en wat de grenzen zijn.

Hoofdstuk 4 gaat over interactieve presentatie van formele wiskundige documenten. Het gaat hierbij om interactieve presentatie gebaseerd op inhoudelijke typentheoretische structuur, waarbij de wiskunde gepresenteerd wordt in de wiskundige omgangstaal. Het wordt de lezer bijvoorbeeld mogelijk gemaakt om het niveau van detail waarop bewijzen gepresenteerd worden, dynamisch te veranderen. Een door ons geïmplementeerd prototype tool laat zien hoe wiskundige inhoud gecodeerd in typentheorie in principe op een natuurlijke en interactieve wijze gepresenteerd kan worden.

Hoofdstuk 5 behandelt de communicatie van stellingbewijzers met computeralgebrasystemen. Het gaat hierbij om communicatie gebaseerd op open standaarden zoals de OpenMath taal. Computeralgebrasystemen excelleren in het doen van symbolische berekeningen, maar missen de garantie voor correctheid die stellingbewijzers bezitten. In een concreet voorbeeld worden formele bewijzen van primaliteit van grote getallen gegenereerd, hetgeen meer rekenkracht vereist dan aanwezig bij de stellingbewijzer. Daarom wordt de hulp ingeroepen van een computeralgebrasysteem. Dit voorbeeld toont aan dat communicatie tussen computerwiskundesystemen nuttig en mogelijk is.

Hoofdstukken 1 en 6 bevatten de inleiding respectievelijk de conclusies van het proefschrift. Deze hoofdstukken plaatsen de bevindingen uit de andere hoofdstukken in de context van een algemene zoektocht naar het ideale computerwiskundesysteem dat ondersteuning biedt bij alle aspecten van het wiskundige werk.
Curriculum Vitae

Titles in the IPA Dissertation Series


P.M. Achten. Interactive Functional Programs: Models, Methods, and Implementation. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. Parallel Local Search. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. The Implementation of Functional Languages on Parallel Machines with Distributed Memory. Faculty of Mathematics and Computer Science, KUN. 1996-5


H. Doornbos. Reductivity Arguments and Program Construction. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. Functorial Operational Semantics and its Declarative Dual. Faculty of Mathematics and Computer Science, VUA. 1996-9


N.W.A. Arends. A Systems Engineering Specification Formalism. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. Normalisation in Lambda Calculus and its Relation to Type Inference. Faculty of Mathematics and Computing Science, TUE. 1996-12


M.M. Bonsangue. Topological Dualities in Semantics. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. Algorithms for Graphs of Small Treewidth. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. Process-algebraic Transformations in Context. Faculty of Computer Science, UT. 1997-02


C.J. Bloo. Preservation of Termination for Explicit Substitution. Faculty of Mathematics and Computing Science, TUE. 1997-05


F.A.M. van den Beuken. A Functional Approach to Syntax and Typing. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. Ins and Outs in Refusal Testing. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. A Discrete-Event Simulator for Systems Engineering. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. Scheduling with Communication for Multiprocessor Computation. Faculty of Mathematics and Computer Science, UU. 1998-03


E. Voermans. Inductive Datatypes with Laws and Subtyping – A Relational Model. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. Towards Probabilistic Unification-based Parsing. Faculty of Computer Science, UT. 1999-02

C.H.M. van Kemenade. **Recombinative Evolutionary Search.** Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. **Learning Reliability: a Study on Indecisiveness in Sample Selection.** Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. **Scheduler Optimization in Real-Time Distributed Databases.** Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. **Message Sequence Chart: Syntax and Semantics.** Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. **Nonlinear approaches to satisfiability problems.** Faculty of Mathematics and Computing Science, TUE. 1999-08

M.T. Romijn. **Analysing Industrial Protocols with Formal Methods.** Faculty of Computer Science, UT. 1999-09

P.R. D’Argenio. **Algebras and Automata for Timed and Stochastic Systems.** Faculty of Computer Science, UT. 1999-10

G. Fábián. **A Language and Simulator for Hybrid Systems.** Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. **Object-Oriented Concepts and Proof Rules.** Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. **Aspects of an Integrated Neural Prediction System.** Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. **A Purely Functional Implementation of Attribute Grammars.** Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. **Viper, A Visualisation Tool for Parallel Program Construction.** Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. **Cryptography and Statecraft in the Dutch Republic.** Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. **UNITY in Diversity. A stratified approach to the verification of distributed algorithms.** Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. **Theories and Tools for the Design of Delay-Insensitive Communicating Processes.** Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. **Studies in Computer Aided Verification of Protocols.** Faculty of Science, KUN. 2000-04

P.H.E.M. Verhoeven. **The Design of the MathSpad Editor.** Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. **Design of a Fruit Juice Blending and Packaging Plant.** Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. **Cocktail: A Tool for Deriving Correct Programs.** Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. **A Framework for Debugging Heterogeneous Applications.** Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. **Another Formal Specification Language.** Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. **The Shape of Evolutionary Search Discovering and Representing Search Space Structure.** Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. **Agents, Objects and Events a computational approach to knowledge, observation and communication.** Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. **Reasoning about Java programs in higher order logic using PVS and Isabelle.** Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. **Improving Design Processes through Structured Reflection.** Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. **Term Graph Rewriting: syntax and semantics.** Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. **Studies in Interactive Visualization.** Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. **Languages for Analysis and Testing of Event Sequences.** Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. **Structural Aspects of Switching Classes.** Faculty of Mathematics and Natural Sciences, UL. 2001-08
M.H. Lamers. Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. Towards Effective Model Checking. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. Mechanical verification of concurrency control and recovery protocols. Faculty of Mathematics and Computing Science, TU/e. 2001-11