Constructive Real Analysis:
a Type-Theoretical Formalization and Applications

Luís Cruz-Filipe

Constructive Real Analysis:
a Type-Theoretical Formalization and Applications

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

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen
op gezag van de Rector Magnificus
Prof. dr. C.W.P.M. Blom
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
dinsdag 15 juni 2004
des namiddags om 3.30 uur precies
door

Luís Calhorda Cruz Filipe

geboren op 10 juli 1978 te Lissabon

Promotor:
Prof. dr. H.P. Barendregt

Copromotor:
Dr. H. Geuvers

Manuscriptcommissie:
Prof. dr. B. Jacobs
Prof. dr. C. Paulin-Mohring (Université Paris-Sud)
Prof. dr. H. Schwichtenberg (Ludwig-Maximilians-Universität München)

AMS Subject Classification (2000): 03B15, 03B35, 03F60, 68T15

Cover design: Catarina Moreno
ISBN 90-9018055-9

IPA Dissertation Series 2004-07
Typeset with LaTeX2$_\varepsilon$
Printed by Print Partners Ipskamp, Enschede

Funding for this work was provided by Fundação para a Ciência e Tecnologia under grant SFRH / BD / 4926 / 2001 (FSE, 3º Quadro Comunitário).

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
This work was partially supported by FCT and FEDER under POCTI.

Acknowledgements

There are many people without whose help this thesis would never have been written.

In the first place, I wish to thank my promotor Henk Barendregt and my copromotor Herman Geuvers. Not only did they guide me through all my work, helping me establish goals and progress towards them, but they encouraged me continuously and were always available. Out of the many discussions that we had there arose many fundamental ideas that were vital for the accomplishment of this work.

As regards the thesis itself, I would like to express my gratitude to all those who helped it evolve: my "proefkonijnen" Bas Spitters, Freek Wiedijk and Iris Loeb, who read the first drafts of several chapters and gave me valuable suggestions regarding form and content; my copromotor Herman Geuvers and my promotor Henk Barendregt, who read the whole manuscript and helped it reach its mature form; and the members of the reading committee, Bart Jacobs, Christine Paulin and Helmut Schwichtenberg, who also made worthy comments which helped me improve its final version. My old friend Catarina Moreno contributed with her designing skills, supplying the cover.

I also feel indebted to all the people who made working in Nijmegen during these years such a pleasant experience. In particular, I would like to thank Bas, Dan, Freek, Georgi, Iris, Jasper, Mariusz, Milad and Sébastien for all the informal (lunch) discussions where so many interesting ideas originated. And I would never have survived without the precious help of Nicole and Marianne.

Throughout this period I also had the opportunity of collaborating with people from the Coq team; in particular, I wish to mention Jean-Christophe Filliâtre, Hugo Herbelin and Pierre Letouzey, whose help was inestimable and in whose cooperation many interesting results were attained. Also worthy of mention are the several people from the TYPES Working Group whom I met on several occasions and with whom I had many interesting conversations.

I should also mention the constant support and motivation provided by the people at the Center for Logic and Computation in Lisbon, who invited me there several times to give talks and with whom many significant e-mails were exchanged.

Besides my work acquaintances, there are several other people I wish to thank. My experience in Nijmegen was highly enriched by the musical presence of PANiek, and I would like to mention Ingrid, Harriët, Wilmy, Irene, Tamara, Mieke, Dik, Pino, Dorus, Nico, Josef, Ivo—and Maria. Throughout these years they were my companions, always cheerful and supportive; and thanks to them I learned the ins and outs of the Dutch way of life.

As always the most important are mentioned last. My friends Paulo and Catarina were the ones who helped me take the big step of moving to Nijmegen in the first place, for which I am grateful; and they have always kept in touch regardless of the distance. And words are not enough to express my gratefulness to my mother, my father and Gonçalo, whose frequent phone calls, letters, e-mails and visits make me feel were here with me all the time.

Contents

List of Figures

Chapter 1

Introduction

For many centuries (even millenia), mathematics has been concerned with providing algorithmic solutions to numerical problems, not only for its useful (and needed) applications but also for the art of finding them. Thus, the ancient Egyptians, Greeks and Babylonians had methods to solve at least particular cases of linear and quadratic equations, while formulas to solve cubics and quartics were published in the 16^{th} century by Cardano and Tartaglia.

In the 19^{th} century, with the proof by Abel of the *inexistence* of a general formula to solve polynomial equations of degree 5 or higher, the spectrum of interest of mathematicians began to broaden: even if general methods to solve equations do not exist, can one prove that there is a solution? The question of solvability became thus a very important one in 20^{th} century mathematics.

With the growing power of computers during the last century, another interesting aspect arose: even if no analytic method exists to solve a given equation exactly, can approximate solutions be numerically computed with arbitrary precision?

The relationship between provability and computability became more clear when notions such as realizability were introduced in the early 20^{th} century. It was then noticed by Brouwer (quoted in [10, Chapter 1]) that the use of the classical axiom $A \vee \neg A$ prevents one from mechanically deriving the algorithm for constructing x from a proof of $\exists x.P(x)$; and this observation eventually gave rise to the constructive way of doing mathematics.

Still most mathematicians think constructive mathematics is not powerful enough for their needs. In answer to this, Bishop published in 1967 his book "Foundations of Constructive Analysis" [10], where (as he says in its introduction) he shows that, contrarily to what many claimed, it was possible to develop almost all of the "relevant" mathematics in a constructive way.

This thesis discusses how this objective can be pursued even further, showing that it is even possible to completely formalize his development of Real Analysis [10, Chapter 2] in the Coq proof assistant, building on top of a previously existing formalization of the Fundamental Theorem of Algebra.

But why formalize it at all? And why constructively?

Formalizing mathematics can be inspiring. In the process of formalizing, one discovers the fine structure of the field one is working with and one gains more confidence in the correctness of the definitions and the proofs. But in addition to this, formalizing mathematics can also be useful; among some of its possible uses are the following.

Correctness guaranteed: the formalized mathematics is checked and thus the proofs are guaranteed to be correct. This can be vital in the realm of software or system correctness, where one wants to be as sure as possible that the mathematical models and the results proved about them are correct.

Exchange of "meaningful" mathematics: that the mathematics is formalized means that it has a structure and a semantics within the Proof Assistant. So a mathematical formula or proof is not just a string of symbols, but it has a structure that represents the mathematical meaning and its building blocks have a definition (within the Proof Assistant). These can in principle be exploited to generate meaningful documents or to exchange mathematics with other applications.

Finding mathematical results: based on the semantics and the structure of the formalized mathematics, it should be possible to query the library and find results easily. Querying based on the (meaningful) structure is already possible, but more semantical querying would be welcome.

An important point of this formalization is that it is a constructive one. There are several reasons for this.

The main one is the intended use of the library. Apart from studying how it conducts as a repository, one wants to study the connections between "conceptual" (abstract) mathematics and "computational" (concrete) mathematics. The first (conceptual math) deals with e.g. the proof of (and theory development leading to) the Fundamental Theorem of Algebra, while the second (computational math) deals with an actual representation of the reals and complex numbers and the actual root finding algorithm that this proof exhibits. This was also an important motivation for choosing to work with Coq; in particular, the work in program extraction relies heavily on the fact that the underlying library is constructive.

The other issue is that of generality. As described in [5], there are many different schools of mathematicians, each claiming that "their" way is the right way to do mathematics. In the intersection of the major schools lies the style proposed by Bishop: the principle of the excluded middle has been repeatedly criticized over the 20^{th} century; so do not assume it. But do not assume any of the more controversial axioms of intuitionistic or recursive mathematics either. The result is a theory which will, in principle, be accepted by most mathematicians.

This thesis describes how Bishop's constructive development of Real Analysis was for the first time formalized within the Coq system. In order to do this, serious thought had to be given to several important issues which had not previously been addressed in this context, namely the representation of concepts and the construction of auxiliary tools to help developing the proofs.

The formalization is the departure point to more general considerations on how a large library should be developed and organized so that its contents can be easily accessed and used by others.

The work described in this thesis can be summarized in three points:

- construction of the C-CoRN library (formalization of Real Analysis and development of tactics);
- development of a working methodology;
- applications to program extraction (case study: extracting and optimizing a program from the formalized library).

For presentation, the first two points are addressed in the inverse order.

Chapter 2 presents a more detailed introduction on the whole work, discussing other formalizations of Real Analysis (in particular in Coq) as well as the pre-existing library of Algebra on top of which this work was developed: the FTA-library. This is followed by some general considerations about the methodology used throughout the whole formalization; and the chapter concludes with a description of the mathematics to be formalized, Chapter 2 of [10], and the identification of some problems which will have to be solved.

Dealing with partial functions is one of these problems, addressed in Chapter 3. This chapter discusses possible ways to formalize them, both in Coq and in other proof assistants, and focuses on the two most natural ways to do it in the context of this work. The advantages and disadvantages of each of these two options are then discussed before a final choice is made. Part of the material in this chapter has been published in [20].

The next crucial issue is automation. Chapter 4 deals with this at length, presenting different ways to define tactics in Coq as well as examples and advantages of each mechanism. Special attention is paid to the new tactics developed to automate proofs in Real Analysis. Some of the content of this chapter has been published in [20]; the extension of `Rational` to partial functions is joint work with Freek Wiedijk, and the new version of the `Step` tactic was designed with Hugo Herbelin.

Chapter 5 then discusses extensively how the mathematics of [10, Chapter 2] were formalized, presenting the major problems encountered and examining the choices that had to be made at each stage. This chapter is a (very) extended version of [21].

The last chapter explores one of the applications of formalized constructive mathematics: program extraction. In Chapter 6, the ideas behind program extraction are presented together with the description of work previously done in the area before examining the problem of getting first a program and later a working program from the repository of formalized mathematics developed at the University of Nijmegen. Most of this chapter contains joint work with Bas Spitters, which was partially published in [22]; the first part of Section 6.3 is joint work with Pierre Letouzey and Bas Spitters.

The thesis ends with a general overview of what was achieved and some concluding remarks.

A short introduction to Coq is provided as Appendix A.

The whole formalization, together with the documentation, can be accessed via the C-CoRN home page, `http://c-corn.cs.kun.nl/`.

Chapter 2

Some Metamathematical Considerations

Formalizing a significant piece of mathematics is a highly non-trivial task which requires some previous analysis and careful planning.

The field of Real Analysis had already been formalized in other theorem provers when this work was begun. This chapter begins with an overview of these different formalizations, together with a brief analysis of their pros and cons and a justification of why it was still sensible to do this work in a new context.

The environment where this formalization was to be developed was the FTA-library. As its name suggests, this library contained a formalization of the Fundamental Theorem of Algebra (the FTA) carefully developed so as to be usable in future work. In Section 2.2, the notation and conventions used in the FTA-library are introduced and explained, as well as the underlying philosophy and the more abstract goals at the heart of the FTA-project.

Since Real Analysis can hardly be classified as a part of the Fundamental Theorem of Algebra, the library was renamed during the development of this work, and is presently known as C-CoRN (the Constructive Coq Repository at Nijmegen).

According to the goals of the FTA-project, which are also the goals of C-CoRN, the formalization should be as general and widely applicable as possible. Thus, instead of aiming at a specific theorem, it was decided to try to formalize a chapter of a reference book as closely as possible, with the advantages and disadvantages of such an option. Since C-CoRN is a formalization of *constructive* mathematics, the natural bibliographic reference was Bishop's "Foundations of Constructive Analysis" [10], and in particular Chapter 2. In Section 2.4 a more detailed overview of the mathematical content of this work is presented, with the double purpose of acquainting the reader with it and *a priori* identifying potential problematic issues which should be solved before starting the formalization.

At the end of the chapter a brief discussion of the problems which can already be seen is presented. A relevant aspect of this discussion is the fact that C-CoRN is on the one hand meant as an extension of the FTA-library, and hence should be coherent with it, but on the other hand expected to follow [10] as closely as possible.

2.1 Related Work

Several formalizations of real numbers, Real Analysis and properties of elementary transcendental functions have been previously completed in different systems. Most of these differ from the one described in this thesis simply because they are classical formalizations.

The first successful attempt to formalize a reference book on mathematics was made by van Benthem Jutting [6] (partially reprinted as [7]). He followed Landau's construction of real numbers [46] as closely as possible, translating it into Automath. His work bears many similarities to the one described in this thesis, not only in motivation but also in his treatment of partial functions (discussed in Section 3.2); but the differences inherent to thirty years' development of technology are marked. Among these, the readability of the development stands out as foremost: whereas the Coq formalization is readable without too much explanation, as Chapter 5 clearly shows, understanding van Benthem Jutting's coding in Automath requires a previous study of that system's language. Also, Automath was not meant to do proofs automatically, therefore Chapter 4 of this thesis has no parallel in his work; this is also reflected in the difference between the conclusions of [6] and those in Section 5.6: whereas the ratio between the original text and its formalized counterpart is, in Automath, always roughly the same, in the Coq formalization of [10] this ratio will be shown to decrease as the work progresses.

Mizar [51] presently includes a classical formalization of Real Analysis. Differential calculus was developed by Kotowicz, Raczkowski and Sadowski, whereas Endou, Wasaki, and Shidama have formalized integral calculus. These formalizations include classical counterparts to all the results presented below in Section 2.4, in particular Rolle's Theorem (Theorem 2.4.11), Taylor's Theorem (Theorem 2.4.15) and the Fundamental Theorem of Calculus (Theorem 2.4.20). It is also interesting to note that it is the only other formalization of those here mentioned that explicitly attempts to deal with partial functions in their full generality.

Harrison's HOL-light system (described in [38]) is another proof assistant that comes with a library of Real Analysis. Once again, this library is built classically. Among the results therein included figure the usual results on preservation of continuity through algebraic operations, derivation rules, Rolle's Theorem (Theorem 2.4.11) and the Law of the Mean (Equation 2.2). Also included in this system is a library of transcendental functions, where exponential and trigonometric functions are defined as power series and their inverses as inverse functions. Finally, integration is defined and the Fundamental Theorem of Calculus (Theorem 2.4.20) is proved. This work, described in [38], has been used together with his formalization of floating point arithmetic in [39] to prove correctness of floating point algorithms.

Mayero has also formalized differential calculus and transcendental functions in Coq, starting with an axiomatic characterization of the reals, and showed in [50] how this formalization can be used to prove correctness of programs in Numerical Analysis. Her formalization of differential calculus in Coq includes notions of (point-wise) continuity and differentiability, derivation rules, and some work on transcendental functions. However, it does not either deal with integral calculus nor state more general results like Rolle's theorem. This is because her motivation is not formalizing Real Analysis in itself, but showing how such a formalization can be used for Numerical Analysis, whence she develops just the theory that she needs for that purpose. For the same reason, she argues that it makes more sense to work classically—which makes her work totally distinct from the one in this thesis.

Mayero's treatment of partial functions is discussed in more detail in Chapter 3, where in particular it is shown that it cannot be adapted to a constructive setting.

In the PVS system, Dutertre has also developed a classical theory of Real Analysis which most of differential calculus. This formalization is described in [27]. Building upon this work, Gottliebsen built a library of transcendental functions described in [37], where she defines exponential, logarithmic and trigonometric functions, proving similar results to those in Section 5.5. Besides these, she defines an automatic procedure to prove continuity of a large class of functions, which works in a similar way to the `Contin` tactic described in Section 4.4, and shows how it can be used interactively with Computer Algebra systems to guarantee the correctness of applications of the Fundamental Theorem of Calculus (Theorem 2.4.20).

More specific aspects of related work are discussed at the beginning of Chapter 3 (how the different formalizations above described deal with partial functions) and Chapter 6 (on the more specific topic of program extraction from constructive proofs). Those chapters also include more detailed comparisons with the work described in this thesis.

Throughout Chapter 5, several comments are made on specific aspects which turned out to be more difficult to formalize than the rest. Besides the afore-mentioned references, many of the remarks there made about similar difficulties encountered by other people when formalizing related topics in other proof assistants are motivated by a discussion during the Workshop in Continuous Mathematics which was held as part of TPHOLs 2002 and attended by many of the people mentioned above.

2.2 Background

The work described here was initially thought of as a direct continuation of the FTA-project, which had been going on at the University of Nijmegen during the period 1999–2000. In this section, the aims and results of that project will be briefly presented, in order to give a clearer picture of the setting in which C-CoRN came to exist and to introduce the preexisting library.

The FTA-project: goals and philosophy

One of the main criticisms to formalizations of mathematics as presently done is their little reusability. Most of the work is geared to proving a given theorem within a specific theorem prover, and little or no attention is paid during that process to the development of a library which will be usable by others later on.

Challenging this approach was one of the early motivations fueling the FTA-project. The goal of this project was to formalize in Coq a constructive proof of the Fundamental Theorem of Algebra (FTA).

Theorem 2.2.1 Let f be a non-constant polynomial over the complex numbers. Then f has a root, i.e., there is a complex number z such that $f(z) = 0$.

Instead of simply proving this theorem as quickly as possible, though, care was taken to develop in this process a generic library of elementary constructive algebra.

In particular, this meant that all the auxiliary lemmas needed for the proof of the FTA should be formalized in as general a way as possible. As a consequence, real numbers were axiomatized (as a Cauchy-complete ordered field satisfying the Archimedean property) rather than built as a concrete structure; similarly, an Algebraic Hierarchy (described in detail in [32]) was constructed, where properties of the algebraic operations were proved at the right level of abstraction.

Two nice properties of this approach are worth pointing out.

First, having generic types of algebraic structures allows reuse of notation for the algebraic operations, which is a very relevant point: in this way one can simulate overloading in a system such as Coq where it is not available. As an example, any ring has an operation (multiplication) that can be written down as $\times$, leaving the ring implicit[1]—thus allowing one to write both $\mathsf{x} \times \mathsf{y}$ for real numbers x and y, $\mathsf{p} \times \mathsf{q}$ for polynomials p and q, or $\mathsf{z} \times \mathsf{z}'$ for complex numbers z and z'. Furthermore, once the axiomatic properties of $\times$ (e.g. associativity, commutativity) have been proved for each structure, all lemmas about rings can be used for all of them.

The Algebraic Hierarchy also leaves open the possibility of reusing the library for other domains than that of the complex numbers. The existence of results about groups makes it possible to formalize group theory on top of it, even though the FTA has nothing to do with groups. A convincing example of this is the extension of the FTA-library with a formalization of Real Analysis described in Chapter 5. The new and the old parts share as much as possible—namely, all the Algebraic Hierarchy and the results about real numbers. The integration between the old and new parts of the extended Algebraic Hierarchy can be seen from the formalization of the complex exponential function done by Hinderer [41].

The main disadvantage of this approach is that the terms that occur in the formalization quickly become large, since they have to refer to the specific structure used. Also, because of the hierarchy of the types of algebraic structures, one will needed to insert in the terms to get to the right kind of algebraic structure. For example, to multiply two elements of a field one first has to access the underlying ring in order to use the multiplication. This is not a problem for the user, since all these operations are implicit: the specific structure is generally an implicit argument, while the functions that map algebraic structures are implicit coercions[2]. But internally these terms are still big, so it slows down the processing of the formalization by Coq.

Another slight disadvantage of this approach is that sometimes proofs can be less direct than in the case that all functions are concretely defined. This will be especially relevant when working with program extraction, which is the subject of Chapter 6. For example, if one knows that one is dealing with the rational numbers, a proof of a specific lemma might be possible that gives a much better extracted program. In the case that one has to give a proof from axioms, this optimization might not be available. Concrete examples

[1]That is, the ring is an implicit argument which the system figures out for itself; see Appendix A.5 for a more precise description of how implicit arguments work.

[2]A description of how the coercion mechanism works can be found in Appendix A.5

of such situations can be found in Chapter 6.

A different feature of the FTA-project was the focus on documentation. If a library is meant to be reused, then it is necessary that a good description of its content be available.

It should be stressed that *documentation* is here understood as different from *presentation*. A presentation of a library of formalized mathematics is simply a list of (possibly pretty-printed) lemmas, where the proof scripts may or may not be included. A documentation of the same library should include the lemmas, possibly in a simplified form (i.e., in a way that their meaning is clear, although the theorem prover may not recognize them as valid statements); it should *not* include the proof scripts; and it should contain other relevant information, such as explanation of non trivial definitions or proofs, comments on the development, and discussion on alternative developments where appropriate.

This philosophy of working was made more precise and taken as a fundamental issue in the development of C-CoRN, as will be explained in Section 2.3.

The Algebraic Hierarchy of the FTA-library

Since the FTA-library is at the heart of the whole work here described, it is essential to have a general understanding of its content and its notation. This section explains the construction and use of the Algebraic Hierarchy and the real numbers, needed for the formalization of Real Analysis. A more detailed description of this hierarchy can be found in [32].

The basics of Coq, including an overview of the type theory it relies on, a description of its syntax and the conventions used throughout this thesis for displaying Coq terms are explained in Appendix A.

At the basis of the FTA-library lies the notion of setoid. Equality in Coq is an intensional (syntactical) notion, which is not appropriate for constructive mathematics: in the usual models of the real numbers, redundancy is essential in the sense that there will be many (syntactically) different representations of (extensionally) the same element. Therefore, it is necessary to have a special type corresponding to sets in constructive mathematics. This will be the type CSetoid of setoids[3].

Furthermore, all work in the FTA-library (and in C-CoRN) done so far deals with sets where a more primitive relation than equality exists: a binary relation #, called *apartness*, which satisfies the following properties for all x,

[3]Throughout the Algebraic Hierarchy the types of structures always begin with a C, standing for Constructive.

y and z:

- irreflexivity: $\neg x \# x$;
- symmetry: if $x \# y$ then $y \# x$;
- co-transitivity: if $x \# y$ then either $x \# z$ or $z \# y$.

The apartness is said to be *tight* (with respect to the equality) if furthermore $x = y$ iff $\neg x \# y$.

The type CSetoid is defined as the following record type:

```
Record CSetoid : Type :=
  { cs_crr   :> Set;
    cs_eq    :  (Relation cs_crr);
    cs_ap    :  (Relation cs_crr);
    cs_proof :  (is_CSetoid cs_crr cs_eq cs_ap)}.
```

where the last field states that cs_ap is an apartness relation which is tight with respect to cs_eq; from this one can prove that cs_eq is an equivalence relation. (Record types are explained in Appendix A.2.) Notice that tightness could be used as a *definition* of equality; but often it is useful to have a more direct definition and then prove that it is tight.

The notation and implicit arguments of Coq allow the apartness (cs_ap S) and equality (cs_eq S) to be written infix respectively as [#] and [=]. The square brackets are needed since # and = are already tokens in Coq, and this system does not allow overloading. In this presentation, however, the latter version will be used for simplicity, since there can never be any confusion about the meaning of the symbols.

Equality being a fundamental notion, most operations and relations are required to respect it; in other words, functions and relations should behave in an extensionally equal way on extensionally equal inputs. It should come as no surprise that this property is known as *extensionality.*

In the presence of a tight apartness, a stronger property (predictably known as *strong extensionality*) is often required. Being strongly extensional amounts to reflecting apartness; for example, if the two outputs of a unary function on two given arguments are apart, then the arguments are also apart. This requirement is justified by the intuitive meaning of "apartness": two objects are apart if they can be distinguished by some finite process. Applying a function to them and comparing the output is such a finite process.

Functions on setoids will always be required to be strongly extensional; otherwise they will be called *operators.* If S1, S2 and S3 have type CSetoid, then S1 → S2 (respectively S1 → S2 → S3) is the type of operators from S1 to S2 (respectively S1 to S2 to S3), whereas (CSetoid_fun S1 S2) (respectively (CSetoid_bin_fun S1 S2 S3)) are the types of the corresponding setoid functions. The latter are again record types, containing the underlying operator and a proof of its strong extensionality. These types will be written as (CSetoid_un_op S) and (CSetoid_bin_op S), respectively, in the case S1=S2(=S3)=S.

Similar types are defined for binary relations and unary relations (predicates) on a setoid. Strong extensionality for these looks slightly different than for functions: if P is a unary predicate, then strong extensionality of P is written down as

$$P(x) \to P(y) \vee x \# y.$$

However, relations and predicates which are not strongly extensional (and occasionally not even extensional) will often be used.

The Algebraic Hierarchy of the FTA-library contained types for seven algebraic structures, each of which is a special case of the previous one: semigroups (CSemiGroup), (commutative) monoids (CMonoid), groups (CGroup), (commutative) rings (with identity) (CRing), fields (CField), ordered fields (COrdField) and real number structures (CReals)[4]. At the basis of this hierarchy lies the type of setoids.

This hierarchy is linear: each structure is an instance of all the previous ones. This is due to the fact that, so far, work has only been done on real and complex numbers; it is hoped that this hierarchy will get richer and gain a more complicated structure once subjects like group theory are incorporated in the library.

The type of each structure is a record type. The first field of this record (the carrier) is a coercion to the structure immediately below in the hierarchy; thus, carrier of a group is a monoid, the carrier of a ring is a group (its additive group), the carrier of an ordered field is a field, and so on.

The next field(s) of the record characterize the rest of the signature of the structure. For example, a group is a monoid plus a unary operation on the carrier (the inverse); a ring is a group with a binary operation on the carrier (multiplication) and an element of the carrier (multiplicative unit); and an ordered field is a field with a binary relation on the carrier (the strict order).

The last field of the record contains the specification of the algebraic structure, i.e., the properties that it satisfies. These are encapsulated within

[4]After this work had been finished, this hierarchy was refined so that monoids and groups no longer have to be commutative, and an extra type CAbGroup of abelian groups was included before that of rings. In practice this makes no difference for the purposes of the following chapters; in this presentation the above described version will be considered, as it corresponds to what is described in [32].

a predicate (typically named is_*type*, where *type* is the type of the structure in question). Intuitively (since adding a field to a record type intuitively corresponds to defining a subtype of the same type) this corresponds to selecting from the structures with the given signature the ones that satisfy the desired axiom(s). This method has also been followed in the Algebraic Hierarchy of Metaprl, described in [64].

As an example, the definitions of CGroup and COrdField are presented:

```
Definition is_inverse (S:CSetoid) (op:(CSetoid_bin_op S)) (one x x_inv:S) :=
  ((op x x_inv) = one) ∧ ((op x_inv x) = one).

Definition is_CGroup (G: CMonoid) (inv: (CSetoid_un_op G)) :=
  ∀x:G.(is_inverse + 0 x (inv x)).

Record CGroup : Type:=
  { cg_crr   :> CMonoid;
    cg_inv   :  (CSetoid_un_op cg_crr);
    cg_proof :  (is_CGroup cg_crr cg_inv)}.

Record strictorder (S:Set) (R:(CSetoid_relation S)) : Set:=
  { so_trans :  (transitive R);
    so_asym  :  (antisymmetric R)}.

Record is_COrdField (F:CField) (less:(CSetoid_relation F)) : Set:=
  { ax_less_strorder  :  (strictorder less);
    ax_plus_resp_less :  ∀x,y:F.(less x y) → ∀z:F.(less (x + z) (y + z));
    ax_mult_resp_pos  :  ∀x,y:F.(less 0 x) → (less 0 y) → (less 0 (x × y));
    ax_less_conf_ap   :  ∀x,y:F. (x # y) ↔ ((less x y) ∨ (less y x))}.

Record COrdField : Type:=
  { cof_crr   :> CField;
    cof_less  :  (CSetoid_relation cof_crr);
    cof_proof :  (is_COrdField cof_crr cof_less)}.
```

The ring operations will be denoted by their usual symbols: $+$ for addition, with identity 0; $-$ for the additive inverse and for subtraction (where $x - y$ is defined as $x + -y$); and $\times$ for multiplication, with unit 1. Division is introduced as Definition 3.1.3 and the slightly more complicated notation for it can be found afterwards in Section 3.1. The (strict) order relation is denoted by $<$; the non-strict order $x \leq y$ is defined as $\neg(y < x)$.

Finally, a term $\mathbb{R}$ of type CReals is assumed as axiom; a concrete term of that type was built separately and proved isomorphic to all other terms of the same type. This formalization, described in [31], is also included in the FTA-library. This model will be relevant in Chapter 6; its description will be postponed until then.

Logic in the FTA-library

The FTA-library also differed from the standard Coq formalizations in the way the logic was used.

As explained in Appendix A, there is a preferred sort for propositions in Coq, namely **Prop**, whereas datatypes usually have type **Set**. One of the main purposes of this distinction, which will be explained in more detail in Chapter 6, is to separate the computational content of proofs from the pure logical reasoning. In particular, data is not allowed to depend on propositions; in other words, if $\mathsf{A} : \mathbf{Prop}$ and $\mathsf{S} : \mathbf{Set}$, then case analysis on elements of type A is not allowed when defining an element of type S.

But this is precisely what mathematicians do all the time! Defining functions by cases (i.e., analyzing the structure of a term of type $\mathsf{A} \vee \mathsf{B}$) is a daily task when proving theorems in Analysis, and this is precisely one of the things that the previous rule forbids—even though it is clearly justified in constructive mathematics.

There are two known solutions to this problem. The first one is to add the desired elimination rule as an axiom; however this raises consistency problems, as it is then necessary (at least from a moral point of view) to check that not all types become inhabited when this axiom is assumed. Another option is to ignore **Prop** altogether, and simply redefine all the logic in **Set**.

The latter approach was taken in the FTA-project. Although this means that rather uncommon notations are used for the logical connectives, this fact will be ignored throughout this work and conjunction, disjunction and the like will be printed using the usual symbols. Still, predicates and relations on a type S will have types $\mathsf{S} \to \mathbf{Set}$ and $\mathsf{S} \to \mathsf{S} \to \mathbf{Set}$, respectively, instead of the standard $\mathsf{S} \to \mathbf{Prop}$ and $\mathsf{S} \to \mathsf{S} \to \mathbf{Prop}$.

In Chapter 6 the issue of logic will be brought up again and analyzed in more detail. At the end of that chapter a better solution will be proposed, corresponding to the current day situation within C-CoRN.

2.3 The C-CoRN project

After the FTA-project was completed, i.e., after a proof of the FTA had been completely formalized, it was clear that most of the goals had been successfully achieved. The abstract way of working was patent in the Algebraic

Hierarchy, with nice consequences like e.g. the fact that real numbers, complex numbers and polynomials over both could be manipulated in a similar way thanks to a number of general lemmas about rings. The documentation was there, allowing humans to access the contents of the library in a way designed for them, rather than having to browse through the source code. And there were general purpose tools (not specifically geared towards the proof of the FTA) available, such as nice algebraic notation and powerful automation facilities.

To test the reusability of the library, though, it was fundamental that work be done in an altogether different direction, preferably by people not involved with the FTA-project. The subject of Real Analysis was a natural candidate, with the Fundamental Theorem of Calculus as emblematic target for formalization.

It soon turned out that, if the spirit of the original FTA-project was to be preserved, it would be advisable to develop a methodology to follow during work. And thus C-CoRN gradually began to emerge.

C-CoRN, the Constructive Coq Repository at Nijmegen, is meant as an extension of the FTA-library to a more general purpose repository. One of the goals of C-CoRN is to provide an environment where significant portions of mathematics can be formalized, and where the formalization process can be studied. The aim is to gain insight into how one can eventually build a computer system with which mathematicians will want to work developing real mathematics.

The methodology of C-CoRN basically systematizes the philosophy behind the FTA-project. This is described in more detail in the remainder of this section.

Documentation

Providing a good documentation for the formalized library in parallel with its development was a central preoccupation since the early stages of the FTA-project. Having a human-readable description of what has been formalized can be very useful in communicating not only content but also ideas, notations and even some technical aspects of the formalization process.

Such a documentation should at any given moment reflect the state of the library, and as such should be intrinsically linked to the script files. At present, Coq provides a standard tool, called `coqdoc` (see [29]), that automatically generates postscript and `html` documentation from the Coq input files. Additional information can be introduced in the documentation via comments in the script file.

The C-CoRN documentation is presently produced using `coqdoc`. It includes all definitions, axioms and notation as well as the statements of all the lemmas in the library, but no proofs, as discussed earlier. In line with the idea of what a good documentation should contain, tactic definitions are omitted from the documentation, but not their description: although the actual code is not presented, the behaviour of the existing C-CoRN specific tactics is explained as well as how and when they can be used.

In the `html` version, hyperlinks between each occurrence of a term and its definition allow the users to navigate easily through the documentation, being able to check quickly any notion they are not familiar with.

Structuring

There are several ways that the lemmas and files in a library of formalized mathematics can be organized. As mentioned earlier, the current trend in most major systems seems to be adding individual files to the library as independent entities, and seldom if ever changing them afterwards.

However, C-CoRN is intended as a growing system upon which new formalizations can be made. The approach above described directly conflicts with this purpose, for it typically leads to dispersion of related lemmas throughout the library and unnecessary duplication of work: each user will be focused on finishing his proofs, not on placing his auxiliary lemmas where everyone can find them.

For this reason, lemmas in C-CoRN are organized in files according to their statements, and files are distributed among directories according to their subjects. Thus, different areas of mathematics appear in different directories and different subjects within one area will be different files in the same directory. Of course, this requires central control over the repository: after an extension, the library has to be reconsidered to put the definitions and lemmas in the "right" place. So far this works well due to the still small dimension of C-CoRN, but in the future this may become problematic if many files are contributed within a short time.

No part of the library is, strictly speaking, immutable: new lemmas can be added at any time to existing files, if they are felt to belong there. In this way, new lemmas then become immediately available to other users. In practice, though, more basic and older parts of the library tend to change less.

Coupled with this method of working, the documentation system described above makes looking for a particular statement a simpler process than in most other systems discussed in Section 2.1. But in addition to this, naming conventions are adopted throughout C-CoRN that allow experienced users to find a specific lemma even quicker without needing to consult the documentation. These naming conventions are too specific to be explainable in an abstract way, but hopefully their flavour can be felt from the examples throughout this thesis.

Automation

An important part of the C-CoRN library consists in tools designed to aid in its own development. Together with definitions, notations and lemmas, several automated tactics are defined throughout C-CoRN. These will be the subject of Chapter 4, so they will not be discussed further at this stage.

Program Extraction

Besides the direct interest of formalizing mathematics *per se*, one particular application was explored in the setting of C-CoRN.

One of the consequences of working constructively, and therefore without any axioms[5], is that, according to the Curry–Howard isomorphism, every proof is an algorithm. In particular, any proof term whose type is an existential statement is also an algorithm whose output satisfies the property at hand.

In Coq there is an extraction mechanism available which readily transforms proof terms into executable ML-programs (see [47]). Marking techniques are used to significantly reduce the size of extracted programs, as most of the information in the proofs regards *correctness* rather than *execution* of the algorithm and can thus safely be removed. Chapter 6 will extensively discuss program extraction.

2.4 The mathematics

The next chapters revolve around the formalization of Real Analysis which marked the transition from the FTA-library to C-CoRN.

As mentioned earlier, the reference for this formalization was Bishop's constructive development of Real Analysis presented in [10, Chapter 2]. This section describes the mathematics developed there in some extent, focusing on some of its constructive aspects.

[5] Although an axiom $\mathbb{R}$ is assumed, elsewhere in the library a concrete term of that type is built.

As Bishop's intent is to develop from scratch a constructive version of classical Real Analysis, he begins by constructing the real numbers. First regular sequences are defined:

DEFINITION 2.4.1 A *regular sequence* $\{x_n\}_{n\in\mathbb{N}}$ of rational numbers is a sequence such that

$$\forall_{m,n\in\mathbb{N}}.\,|x_m - x_n| \leq \frac{1}{m} + \frac{1}{n}.$$

The regular sequences are the real numbers in Bishop's work.

The definitions of apartness and equality on this set are immediate:

$$\begin{aligned} x < y &\Leftrightarrow \exists_{n\in\mathbb{N}}.y_n - x_n > \frac{2}{n} \\ x \# y &\Leftrightarrow x < y \vee y < x \\ x = y &\Leftrightarrow \forall_{n\in\mathbb{N}}.\,|x_n - y_n| \leq \frac{2}{n} \\ &\Leftrightarrow \neg x \# y \end{aligned}$$

These definitions yield an irreflexive and cotransitive apartness which is tight with respect to the equality; the latter is an equivalence relation.

Definition 2.4.1 is stronger than the usual definition of Cauchy sequence (and in particular from the definition used in the model of the reals included in the FTA-library). This fact has some nice consequences. In the first place, the definition of the operator that assigns to any Cauchy sequence of real numbers its limit becomes much easier because of the fixed rate of convergence. Also proving the equation $x = y \Leftrightarrow \neg x \# y$ becomes trivial.

Section 2 of [10, Chapter 2] deals with the basic properties of these real numbers, and includes material that was already present in the FTA-library: the algebraic operations and their properties; order relation and its relation with the operations; and the concept of (partial) function.

These results can be summarized as follows.

PROPOSITION 2.4.2 With these definitions, the regular sequences of rational numbers form a real number structure in the sense of the FTA-library.

Due to the isomorphism result mentioned at the end of Section 2.2, the fact that this model is different from the one in [31] becomes a minor issue. This can be seen even more directly, as from the definition of Cauchy sequence the following is trivial to prove.

PROPOSITION 2.4.3 Every Cauchy sequence of rational numbers has a regular subsequence.

The next section deals with sequences of real numbers. Part of this material was also developed in the FTA-library, namely most of what regards definition and main properties of Cauchy sequences and completeness of real number structures, but there is an important part missing on subsequences and series. This theory is very close to its classical analogue, so it will not be discussed further.

At the end of this section, e and π are defined as examples of sums of specific series.

Continuity

Section 4 of [10, Chapter 2] starts dealing with functions, namely with continuity, and some subtle issues arise which make the theory start to differ somewhat from the classical one.

The first problem has to do with compactness. The classical definition of compact set (a set S is compact iff for every open cover $\{\mathcal{F}_i\}_{i\in I}$ of S there is a finite subcover $\{\mathcal{F}_{i_k}\}_{k=1,\ldots,n}$ which still covers S), while constructively making sense, is too strong for practical purposes: it is not even possible to prove that the interval $[0,1]$ is compact with the usual topology on the real line without assuming extra axioms; in recursive mathematics this is actually even *not* true, as is shown in [5]. Therefore, Bishop turns his attention to the property of total boundedness:

Definition 2.4.4 A set $A \subseteq \mathbb{R}$ is *totally bounded* iff for every $\varepsilon > 0$ there exist points $x_1, \ldots, x_n \in A$ such that

$$\forall_{y\in A}.\exists_{i\in\{1,\ldots,n\}}.\,|y - x_i| < \varepsilon.$$

Classically, the previous definition of compactness is equivalent to Cauchy completeness (a set A is Cauchy complete if it contains the limit of every Cauchy sequence with values in A) and total boundedness. Constructively this is obviously not so, as the interval $[0,1]$ is easily seen to be complete and totally bounded but is not provably compact; but being totally bounded turns out to be a sufficient condition in most situations. Therefore, the following is a satisfactory definition of compact sets of real numbers:

Definition 2.4.5 A set $S \subseteq \mathbb{R}$ is *compact* iff it is Cauchy complete and totally bounded.

A compact interval is defined to be a set $\{x \in \mathbb{R} \mid a \leq x \wedge x \leq b\}$, with $a \leq b$, denoted usually by $[a,b]$. This is consistent with the previous definition, since it can be proved that compact intervals are compact sets.

The next problem that shows up is the definition of continuous function itself. Classically, this is a pointwise notion (a function f is continuous at a point x); however, this notion is constructively too weak. The usual theorems about continuous functions fail to hold unless extra axioms are assumed; for example, the fact that functions that are continuous at all points in a compact interval I are uniformly continuous in that interval is constructively not provable: there is a recursive counter-example, which can be found in [5].

It should be noticed that this same problem has been pointed out by people doing Nonstandard Real Analysis, showing that it is not a serious flaw of constructive mathematics but rather a coincidental feature of classical mathematics. Exercise III.3.15 on page 131 of [43] discusses how, in a nonstandard world, the classical definition of pointwise continuity does not imply uniform continuity on compact sets because it does not quantify over "enough" points. In that context, this issue is solved by using a different notion of pointwise continuity which does imply uniform continuity on compacts.

The same problem of not being able to quantify over enough points is encountered in Point-free Topology. In [53] it is shown how, once again, this can be avoided by using the topological definition with open sets instead of the problematic ε-δ definition.

Bishop has a more practical approach: simply forget about pointwise continuity altogether (which as argued above is not such an interesting concept anyway) and directly define uniform continuity. Interestingly, it has also been argued by Bridger and Stolzenberg in [12] that this approach also yields a simpler and equally powerful theory of classical Real Analysis.

Definition 2.4.6 A function $f : \mathbb{R} \to \mathbb{R}$ is *continuous* in a compact interval $[a, b]$ iff there is a function $\delta : \mathbb{R}^+ \to \mathbb{R}^+$ such that

$$\forall_{\varepsilon>0}.\forall_{x,y\in[a,b]}.\,|x - y| \leq \delta(\varepsilon) \to |f(x) - f(y)| \leq \varepsilon.$$

Notice that this definition still differs from the classical one in the use of $\leq$ instead of $<$; but it is easy to see that they are equivalent, and constructively $\leq$ is a negative concept, which will make reasoning with it somewhat easier (in particular, goals of the form $a \leq b$ can be proved by contradiction). This is a feature of constructive analysis that will be present throughout this whole section.

The function δ is called a *modulus of continuity* for f. For practical purposes, though, it is not needed that it be a function; that is, the definition of continuity could have been stated instead in the usual ε-δ way, and this would have been equivalent to Definition 2.4.6 in the presence of the Axiom of Choice[6]. Since the extensionality of δ plays no visible role in this development, the option was taken to choose for the more usual formulation when formalizing this concept.

The notion of continuity can be generalized to arbitrary proper intervals (intervals with more than one point):

DEFINITION 2.4.7 A function f is continuous in a proper interval J iff it is continuous (in the sense of Definition 2.4.6) in every compact interval $I \subseteq J$.

This definition is consistent with the previous one in the sense that if I is a proper compact interval and f is continuous in I then it will also be continuous in I under this generalized interpretation.

Continuity of functions in both senses is preserved by the ring operations (pointwise addition, subtraction and multiplication) and also by taking of the maximum and the absolute value. However, composition is slightly more complicated because, although continuous functions have a supremum and an infimum in every compact interval, they do not necessarily attain it. Therefore, continuity of the composition must be stated read as follows:

LEMMA 2.4.8 Let I and J be intervals and $f : I \to J$ and $g : J \to \mathbb{R}$ be two continuous functions such that every compact interval included in I is mapped by f into a compact subinterval of J. Then $g \circ f$ is continuous in I.

The extra condition can be restated as: for every $[a, b] \subseteq I$, there exist c and d such that $f([a, b]) \subseteq [c, d] \subseteq J$. Though this is true in many settings, such as classical or intuitionistic mathematics and formal topology, constructively one can only prove the weaker $f([a, b]) \subseteq (c, d) \subseteq J$, which might not suffice; therefore, the stronger hypothesis has to be assumed.

A particular situation is division, where the side condition can be simplified using the knowledge of the precise domain of the function being composed with (taking of the reciprocal):

LEMMA 2.4.9 Let $f : I \to \mathbb{R}$ be a continuous function such that for every compact $J \subseteq I$ there is a real number $c > 0$ with $|f(x)| \geq c$ for every $x \in J$. Then $\frac{1}{f}$ is continuous on I.

This section concludes with a theory of sequences and series of (continuous) functions. For reasons similar to those discussed above only uniform convergence of sequences is considered. This part of the theory again closely parallels the development of the analogue theory for real numbers.

[6]Without assuming this axiom δ will be an operator, i.e., not necessarily (strongly) extensional.

Differentiation

The "real" Real Analysis begins in Section 5 of [10], which focuses on differentiation, differentiable functions and one-variable differential calculus in general.

All of what was said before about the problems of pointwise continuity can be applied *mutatis mutandis* to the constructive analysis of the notion of pointwise derivative. Therefore, it should come as no surprise that the uniform concept is considered the fundamental one.

However, there is still one small detail left. If the classical definition is directly used, the condition for g to be a derivative of f in an interval $[a, b]$ reads as follows:

$$\forall_{\varepsilon>0}.\exists_{\delta>0}.\forall_{x,y\in[a,b]}.\ |x-y| \leq \delta \to \left|\frac{f(x)-f(y)}{x-y} - g(x)\right| \leq \varepsilon. \tag{2.1}$$

This definition poses a tricky problem: the division on the left of the equation can only be done if x and y are apart. And adding $x \# y$ as a side condition yields something which is too awkward to use, requiring always two separate cases to be analyzed (when $x \# y$ and when x and y are "close enough", which is always a fuzzy concept and is often used where a long and tedious proof would be due). In order to bypass this problem, the following (equivalent) definition is used instead:

Definition 2.4.10 Let f and g be real-valued functions continuous on the proper[7] compact interval $[a, b]$. Then g is said to be a *derivative* of f iff

$$\forall_{\varepsilon>0}.\exists_{\delta>0}.\forall_{x,y\in[a,b]}.\ |x-y| \leq \delta \to |f(x)-f(y)-g(x)(x-y)| \leq \varepsilon\,|x-y|\,.$$

This definition generalizes to arbitrary proper intervals in a similar way to that shown for continuity.

Now it is straightforward to show that the usual rules for the derivative of the sum, product, division and composition hold; for the last two, side conditions similar to those in Lemmas 2.4.8 and 2.4.9 are needed.

It can also be shown that any two derivatives of f on an interval $[a, b]$ must coincide on that interval. This justifies the following notation, which will be used from this point on: if f is a differentiable function in $[a, b]$, then f' is the derivative of f whose domain is *precisely* the interval $[a, b]$.

As usual, the n^{th} derivative of a function f is defined as

$$\begin{aligned} f^{(0)} &= f \\ f^{(n+1)} &= \left(f^{(n)}\right)' \text{ if } f^{(n)} \text{ is differentiable} \end{aligned}$$

[7] An interval is proper if it contains more than one point.

With these definitions, constructive versions of both Rolle's and Taylor's theorem can be proved. These will now be discussed in some detail.

The classical version of Rolle's theorem reads as follows.

THEOREM 2.4.11 Let f be a differentiable function on the interval $[a, b]$ such that $f(a) = f(b)$. Then there exists an $x \in [a, b]$ such that $f'(x) = 0$.

Constructively, this statement is too strong. However, the conclusion of the theorem can be just slightly modified to yield a constructively valid statement:

THEOREM 2.4.12 Let f be a differentiable function on the interval $[a, b]$ such that $f(a) = f(b)$. Then, for any $\varepsilon > 0$, there exists an $x \in [a, b]$ such that $|f'(x)| \leq \varepsilon$.

(This situation is similar to that of the Intermediate Value Theorem, described in detail in Chapter 6, from page 156 onwards. As is shown there, the *hypotheses* of the theorem can also be strengthened to yield a constructive version with the same conclusion. Beeson [5] discusses from a more general perspective the different ways in which constructive analogues to classical theorems can be found.)

The following important corollary of this theorem is usually known as the Law of the Mean:

COROLLARY 2.4.13 Let f be a differentiable function on $[a, b]$. Then, given $\varepsilon > 0$, there exists an $x \in [a, b]$ such that

$$|f(b) - f(a) - f'(x)(b - a)| \leq \varepsilon.$$

At first sight, these constructive results seem much weaker than their classical counterparts. Although this is in part true, it is worthy to point out that *in practice* this apparent weakness tends to disappear. There is a very simple reason for this: even though both theorems, in their classical versions, state equalities concerning the value of f', they do so at an *unspecified* point, that is, they speak about an x which is existentially quantified. Therefore, when used in practice (and the Law of the Mean is in fact the version of Rolle's theorem which is the most useful in practice, as it makes almost no requirements over f), these theorems are used to obtain bounds: the classical version of the Law of the Mean trivially yields

$$|f(b) - f(a)| \leq \|f'\|_{[a,b]} \times |b - a| \tag{2.2}$$

where $\|f'\|_{[a,b]}$ is the norm of f' in $[a, b]$, defined as the supremum of the image of $[a, b]$ through $|f'|$.

Interestingly, Equation (2.2) is also constructively provable. This happens because, even though Corollary 2.4.13 does not provide an equality, it provides an approximation which can be made as good as desired. Therefore, one immediately sees that

$$|f(b) - f(a)| \leq \left(\|f'\|_{[a,b]} + \varepsilon\right) |b - a|$$

for any positive ε, from which it is easy to conclude that Equation (2.2) also holds.

It should be noted that Equation (2.2) is even presented by some (classical) authors as the actual Law of the Mean, as exemplified by Dieudonné's discussion in [25, 26].

A similar situation holds regarding Taylor's Theorem.

DEFINITION 2.4.14 Let f be an n times differentiable function on an interval I and let a be a point in I. The Taylor polynomial for f around a with degree n is the function

$$x \mapsto \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x - a)^k$$

Taylor's Theorem states how good an approximation to f this really is.

THEOREM 2.4.15 Let f be an $(n + 1)$ times differentiable function on an interval I and let a and b be points in I. Define R as

$$R \stackrel{\text{def}}{=} f(b) - \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(b - a)^k;$$

then there is a point c between a and b such that

$$R = \frac{f^{(n+1)}(c)}{(n + 1)!}(b - a)^{n+1}.$$

Like the previous two, it is also a theorem which classically states an equality involving an existentially quantified point, and whose constructive counterpart becomes an approximation theorem; but the same analysis of the main applications of Taylor's Theorem shows that this difference is essentially irrelevant.

However, Bishop's version of this theorem differs from the classical in yet another respect. The classical proof of Taylor's theorem as presented for example in [3] proceeds by defining an auxiliary function whose derivatives from order 0 to n all assume the value 0 at the endpoints of $[a, b]$ and iterating Rolle's Theorem. Unfortunately, this reasoning doesn't work directly with the constructive version of Rolle's Theorem, as this does not give exact zeroes of the auxiliary function. Bishop therefore gives a modified version of this proof in [10], which requires just one application of Rolle's Theorem, but which has the drawback of providing a slightly worse approximation.

THEOREM 2.4.16 If the conditions of Theorem 2.4.15 hold, then for any positive ε there is a point c between a and b such that

$$\left|R - \frac{f^{(n+1)}(c)}{n!}(b-c)^n(b-a)\right| \leq \varepsilon.$$

It has been pointed out that there are other variants of Taylor's Theorem which can be directly constructivized and therefore constitute an improvement on this formulation. An example is the version where the remainder is written as an integral, see Exercise 7.8.8 (page 370) of [3]. It was chosen not to formalize one of these in the first place because, as was discussed above, one of the goals of this work was to show how a chapter of an existing book could be *directly* translated into a computer-checkable proof.

Finally, in the case that f is an infinitely differentiable function, the sequence of Taylor polynomials around a fixed point defines a convergent series, known as the Taylor series. A corollary to Taylor's Theorem is

COROLLARY 2.4.17 Let c be a positive number such that for every $r \in (0, c)$,

$$\frac{r^n f^{(n)}}{n!} \to 0 \qquad \text{(as a sequence of functions)}.$$

Then the sum of the Taylor series for f around a coincides with f on the open interval $(a - c, a + c)$.

If this equation holds for any positive r, then the sum of this series defines a total function on the real line.

Integration

In the setting of Bishop's work, integrating real-valued functions is in some sense a more straightforward task than in classical mathematics. One of the reasons is the following: intuitively (although it cannot be proved without assuming non-classical axioms), every function whose domain includes an interval $[a, b]$ is continuous on that interval. This certainly simplifies the work where integration is concerned, as it implies that the Riemann integral—or some slight variation of it—is "enough".

The classical definition of the Riemann integral proceeds in two stages.

DEFINITION 2.4.18 A *partition* P of an interval $[a, b]$ is a finite ordered[8] sequence of points $\{x_0, \ldots, x_n\}$ such that $x_0 = a$ and $x_n = b$.

The maximum of the numbers $\{x_1 - x_0, \ldots, x_n - x_{n-1}\}$ is called the *mesh* of P. Given a partition P and a function f the partial sums

$$S_P \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} \sup_{y \in [x_i, x_{i+1}]} \{f(y)\} (x_{i+1} - x_i)$$

and

$$s_P \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} \inf_{y \in [x_i, x_{i+1}]} \{f(y)\} (x_{i+1} - x_i)$$

constitute first approximations of the integral of f in $[a, b]$. The lower integral of f is then defined as the supremum of the s_P's and the upper integral of f as the infimum of the S_P's. If the two coincide, f is said to be integrable and $\int_a^b f$, the integral of f in $[a, b]$, is defined to be this common value.

Constructively this is again a very problematic process, as there is no guarantee that all the necessary infima and suprema will exist (since it is not constructively true that every upper bounded set of reals has a supremum). Instead, closely following the classical proof that every continuous function is integrable provides a simple and elegant way around this problem.

This proof proceeds as follows: consider for the moment only *even partitions*, that is, partitions where the distance between two consecutive points is constant, and use (uniform) continuity to show that the difference between the lower and upper integral can be made as little as possible by making this distance small enough.

This reasoning is constructive and can be readily adapted to provide an alternative definition of integral. Given a function f continuous on an interval $[a, b]$, Bishop defines for every natural number n the even partition

$$\begin{aligned} P_n &\stackrel{\text{def}}{=} \{a + i\frac{b-a}{n}\}_{i=0}^{n} \\ &\stackrel{\text{def}}{=} \{x_i\}_{i=0}^{n} \end{aligned}$$

[8]In the classical case, being ordered is not, strictly speaking, relevant, as any list of points can always be ordered. However, this definition will also be used constructively, and there this requirement is essential.

with $(n+1)$ points. To this the sequence of sums

$$\begin{aligned} S_n &\stackrel{\text{def}}{=} \sum_{i=0}^{n-1} f(x_i)\,(x_{i+1}-x_i) \\ &= \frac{b-a}{n}\sum_{i=0}^{n-1} f(x_i) \end{aligned}$$

is naturally associated. The proof above then can be read as saying that this is a Cauchy sequence, and its limit is defined to be the integral of f, denoted by $\int_a^b f$.

In order to show that this definition makes sense, Bishop proves that this integral behaves as the classical one in the following manner: given any $\varepsilon > 0$, there is a positive real δ such that for every partition $P = \{x_0, \ldots, x_n\}$ with mesh smaller than δ and any choice of points $\{y_0, \ldots, y_{n-1}\}$ such that $x_i \leq y_i \leq x_{i+1}$ the following inequality holds:

$$\left| \int_a^b f - \sum_{i=0}^{n-1} f(y_i)\,(x_{i+1}-x_i) \right| \leq \varepsilon.$$

The usual properties (linearity, monotonicity) of the integral are then proved by proving similar properties of sums and passing to the limit.

The most important theorems in this section are, of course, the Fundamental Theorem of Calculus and its corollaries.

Bishop's formulation of the Fundamental Theorem of Calculus reads as follows:

Theorem 2.4.19 Let f be a continuous function on a proper interval I and $a \in I$. Let g be the function defined in I by the expression $g(x) = \int_a^x f(t)dt$; then:

1. f is a derivative of g in I;

2. for any function g_0, if f is a derivative of g_0 in I, then the difference $g - g_0$ is a constant function in I.

The following corollary (not included in [10]) is also frequently referred to as "the" Fundamental Theorem of Calculus (for example in [3]); in particular, it is the version of the theorem which is formalized in most of the work discussed in Section 2.1.

Corollary 2.4.20 In the conditions of Theorem 2.4.19, for every g_0 such that f is a derivative of g_0 in I and for all $x, y \in I$, $\int_x^y f(t)dt = g_0(y) - g_0(x)$.

Important consequences of this theorem are the possibility of permuting limits with integrals or derivatives when working with sequences of functions:

COROLLARY 2.4.21 Let $\{f_n\}$ be a sequence of functions converging to a function f on a non-void interval I and let a be a point of I. Write

$$g_n(x) = \int_a^x f_n(t)dt$$

and

$$g(x) = \int_a^x f(t)dt$$

for each positive integer n. Then $g_n \to g$ on I.

COROLLARY 2.4.22 Let $\{f_n\}$ be a sequence of functions converging to a function f on a proper interval I. Assume that $\{f'_n\}$ converges to a continuous function g on I. Then g is a derivative of f on I.

Transcendental Functions

The last section of Bishop's chapter on Real Analysis deals with transcendental functions. This section may be regarded in some sense as application and examples regarding all the theory that has been developed so far, as well as further evidence to support the claim that constructive mathematics allows as much to be done as its classical counterpart.

In Section 5, Bishop presented the constructive analogue of Taylor's Theorem, as well as some corollaries which provide sufficient conditions for convergence of series of functions. Among these are power series, i.e., series of the form

$$\sum_{n=0}^{\infty} a_n x^n;$$

and of particular interest is the case when a_n itself is of the form $a'_n/n!$, which is the form that gives rise to most of the usual functions used in Analysis. These series can be shown to converge on the whole real line assuming very general conditions on the a_n; for example, assuming that $a_{n+1} \leq c \times a_n$ for some positive c.

In order to motivate his definition of the exponential and elementary trigonometric functions, Bishop begins by examining the properties these functions are expected to have. In the case of the exponential function, these boil down to the differential equations

$$\begin{aligned} \exp' &= \exp \\ \exp(0) &= 1 \end{aligned}$$

which imply that any such function f must have as Taylor representation the series

$$\sum_{n=0}^{\infty} \frac{1}{n!} x^n.$$

This is in turn a convergent series, therefore defining a total real-valued function which is its own derivative.

At this point it is trivial to prove that $\exp(1) = e$, where e was defined at the end of Section 3, which will justify the use from this point onwards of the notation e^x for $\exp(x)$.

Corollary 2.4.17 provides a method for proving that two functions are equal:

Corollary 2.4.23 Let f and g be real-valued functions satisfying the conditions in Corollary 2.4.17 for every positive r, and suppose there is a point $x \in \mathbb{R}$ such that, for all $n \in \mathbb{N}$, $f^{(n)}(x) = g^{(n)}(x)$. Then f and g coincide.

This fact is used to prove the well-known rule $e^{x+y} = e^x e^y$: taking y as a parameter, two auxiliary functions are defined by

$$\begin{aligned} f_y(x) &= e^{x+y} \\ g_y(x) &= e^x e^y \end{aligned}$$

and it is then easy to show that Corollary 2.4.23 applies for any y.

From the two equations $e^1 = e$ and $e^{x+y} = e^x e^y$ all of the usual properties of the exponential function can then be easily derived.

This reasoning is very similar in spirit to the coinductive approach to calculus described e.g. in [57]. The option of formalizing such a development of transcendental functions was not considered, though, since as said earlier one of the main motivations for this work was to see how closely one could follow a reference book.

The logarithm is introduced directly as an indefinite integral, rather than as the inverse function to the exponential. There is a very good reason for this. If one wants to give a constructive definition of an inverse function to the exponential function, there are three steps involved:

- providing an algorithm that, given y, returns a witness x such that $\exp x = y$;[9]

[9] Assuming, of course, that y is in the image of exp.

- proving that this witness does indeed satisfy the condition $\exp x = y$;
- showing that this construction is functional, i.e., if two different representations of the same real number are given then the witnesses constructed will also represent the same real number.

After these three steps are finished, the logarithm can then be (finally) defined.

Alternatively, assuming that there is an inverse function log to the exponential and applying the chain rule for the derivative yields the relations $\log'(x) = \frac{1}{x}$ and $\log(1) = 0$. This justifies *defining* directly $\log(x) = \int_1^x \frac{1}{y} dy$, thus taking care of the first and third points at once. One can then prove as a lemma that this function is indeed the inverse to the exponential—which is precisely Bishop's approach. Once this is proved, all usual properties of the logarithm are then quite straightforward to obtain.

Once again, it should be pointed out that many presentations of classical mathematics also choose for this way to define the logarithm, as it has in itself more content than the definition as an inverse to the exponential.

Trigonometric functions are dealt with in a similar way. Sine and cosine are characterized by the four equations

$$\begin{aligned} \sin' &= \cos \\ \cos' &= -\sin \\ \sin(0) &= 0 \\ \cos(0) &= 1 \end{aligned}$$

and the same procedure which was used above gives rise to the usual power series representations for these two functions, which can then be shown to satisfy all their usual properties.

The cosine function is positive at the origin and then strictly decreases until it reaches the value -1. This fact is used by Bishop to define a Cauchy sequence which converges to a zero of this function, which he denotes by $\frac{\pi}{2}$. Although this gives rise to a definition of π which coincides with the one given earlier, this fact is not mentioned in [10].

Their two inverse functions arcsin and arccos are defined as indefinite integrals, as was the case with the logarithm. Although Bishop doesn't treat the tangent function, it can also be defined directly by $\tan x = \frac{\sin x}{\cos x}$ and shown to satisfy all its usual properties.

This concludes Chapter 2 of [10], the presentation of constructive Real Analysis to be formalized. Even without any further considerations, several problems which will arise during the formalization part can already be identified. These will be considered in the next section.

2.5 What had to be solved before formalizing

The previous sections show two different ways to look at mathematics, which must be in some sense reconciled in order to achieve a successful formalization of Bishop-style constructive Real Analysis in the setting of the FTA-project. Besides that, Bishop's work presents some new challenging issues that require some thought before and throughout the formalization process. In particular, the following questions must be addressed.

- In the FTA-project, much work was put in developing a general theory and a general setting where future developments could be made; therefore, many lemmas were proved in a very general and abstract form, and real numbers a.o. were characterized by an algebraic specification of their properties rather than through the analysis of a particular construction. However, Bishop was not concerned with generality, but with developing Real Analysis; hence, a concrete model of the real numbers was constructed and worked upon, and when needed, its specific properties were used rather than the properties of the algebraic structures they happened to instantiate.

- In the FTA-project, the main focus was on polynomials. These are very well-behaved functions: they are infinitely differentiable, analytic and, most important, total. In Real Analysis, though, many useful functions are not differentiable (e.g., absolute value in any interval containing the origin), analytic or even total (like the logarithm or the tangent). As a consequence, a satisfactory theory of partial functions is needed at the level of the Algebraic Hierarchy.

- After derivatives are introduced, they become omnipresent in proofs in Real Analysis. Both the proof of the Law of the Mean and of Taylor's Theorem require computing the derivative of very nasty-looking auxiliary functions and analyzing them. Also, manipulation of complex expressions involving real numbers is often required. In order to make these steps of the formalization feasible and user-friendly, substantial automation must to be present.

- Finally, at every stage implementation choices have to be made. Often there are several different, but equally appealing, ways to formalize the same mathematical concept. Which should be chosen, and what factors should be taken into account in this choice?

The first point is the easiest one to deal with. Even though Bishop starts by constructing a concrete model of real numbers, by the end of Section 2 he has proved lemmas equivalent to those in the FTA-library, which had been proved there from the axioms of the real number structures; and thereafter the proofs rely mostly on these lemmas rather than on the specific representation of the reals. This allows his work to be directly written in terms of a *generic* real number structure, with only some minor changes here and there.

Similarly, the question of generality of the lemmas can be solved quite straightforwardly. Consider as an example the proof that the integral is linear, that is, the equality

$$\int_a^b (\alpha f + \beta g) = \alpha \int_a^b f + \beta \int_a^b g.$$

This equality is proved as follows:

$$\begin{aligned}
&\int_a^b (\alpha f + \beta g) = \\
&\quad = \lim_{n\to\infty} \sum_{i=0}^{n-1} (\alpha f(x_i) + \beta g(x_i))\,(x_{i+1} - x_i) \\
&\quad = \lim_{n\to\infty} \left(\alpha \sum_{i=0}^{n-1} f(x_i)\,(x_{i+1} - x_i) + \beta \sum_{i=0}^{n-1} g(x_i)\,(x_{i+1} - x_i) \right) \\
&\quad = \alpha \int_a^b f + \beta \int_a^b g
\end{aligned}$$

In this proof, the first and the last step are simply folding and unfolding of the definition of integral. The relevant step is, therefore, the middle one; this can in turn be justified by two simple lemmas:

Lemma 2.5.1 Let x and y be two Cauchy-sequences of real numbers. If, for all $n \in \mathbb{N}$, $x_n = y_n$, then $\lim x = \lim y$.

Lemma 2.5.2 Let a_i and b_i denote real numbers, for $i = 0, \ldots, n$. Then the following equality holds for any $\alpha, \beta \in \mathbb{R}$:

$$\sum_{i=0}^{n} \alpha a_i + \beta b_i = \alpha \sum_{i=0}^{n} a_i + \beta \sum_{i=0}^{n} b_i.$$

These lemmas are implicitly assumed without proof in [10]. Of course, this is common practice in mathematics, as they are direct consequences of the properties of real number structures. But in the setting of a formalization they must be somehow proved before they are used; and here the philosophy of the FTA-project comes into play, saying that these lemmas should in the first place be recognized as useful and non trivial facts that have a right to be stated and proved on their own. Furthermore, they should be stated in as general a form as possible.

Lemma 2.5.1 states a general property of limits, which is valid for convergent Cauchy sequences in any structure where these concepts make sense. In the Algebraic Hierarchy the minimal such structure is an ordered field; therefore, this lemma should be restated and proved in terms of a generic ordered field.

Lemma 2.5.2 just deals with addition and sums; by a similar reasoning, it should be stated and proved in any abelian group.[10]

Finally, the linearity of the integral—which only makes sense in a real number structure—can be proved by application of these two lemmas, which speak about simpler algebraic structures.

This example should provide a clear picture of the methodology used throughout the whole of C-CoRN: in every proof, identify the steps which are instances of more general lemmas that can be stated in a more abstract form; then prove these lemmas for the simplest structure where they still hold.

Partial functions will be discussed in detail in Chapter 3. There different ways of representing partial functions will be analyzed, both from an informal mathematical perspective and a type-theoretic perspective; different representations will be compared by taking different aspects in consideration. Special attention will be paid to the faithfulness of the representation, i.e., how closely working with the different representations resembles the mathematician's approach; to the efficiency of the representation, both in terms of size and time required to manipulate the resulting terms; and to the user-friendliness of the representation, addressing questions such as readability and usability of each different definition.

Automation is the subject of Chapter 4. A lot of automatic routines had already been developed in the FTA-project, and these turned out to be extremely helpful in formalizing Real Analysis; but they did not suffice. The different methods to develop automated tactics available in Coq will be presented, briefly discussed and exemplified through the tactics present in the FTA-library. Then it will be shown how these tactics were extended to deal with the new concepts that were added to the Algebraic Hierarchy, in particular partial functions. Furthermore, the new tactics specific to C-CoRN (for example, that prove continuity or compute derivatives of functions with long expressions) are described in some detail and their performance is discussed.

[10] For simplicity, sums are only formalized at the level of abelian groups.

The formalization itself is the subject of Chapter 5. The mathematical theory described in Section 2.4 above is again presented, but this time in its formalized version as part of C-CoRN. All issues related to the process of formalization are addressed here, in particular why definitions and lemmas were stated in a specific way and what factors were taken into account when choosing between different possible representations of the same mathematical concept.

Chapter 3

Partial Functions

At the end of the previous chapter the need for a careful treatment of partial functions was stated and briefly explained. This chapter is devoted to the detailed analysis of this problem and some of its possible solutions.

Partial functions arise everywhere in real mathematics, and any serious attempt at formalization must at some point deal with the question of how to treat them. In some systems, such as PVS [61], this is taken care of by a mechanism of side conditions: each time a partial expression is typed in, a goal is generated requiring the user to prove that that expression is well typed. However, this has other disadvantages—in particular, type checking becomes undecidable in the presence of side conditions in the typing rules.

Another approach was followed in Mayero's development of classical Real Analysis in Coq, described in [49]. Here, all functions are simply assumed to be total; but in statements of lemmas side conditions are explicitly included to ensure that they are never applied outside of their domain. A similar approach is taken in other (classical) formalizations on systems based on type theory, such as HOL-light (see [38]) and Mizar (see [51]).

Unfortunately, this approach only works in a classical setting; in a constructive world there are examples of functions which are intrinsically partial, that is, that cannot be extended to a total function. One such function is the reciprocal function $x \mapsto \frac{1}{x}$ on the reals. Since there are extensions of constructive mathematics where all functions are continuous on their domain, any total function coinciding with the reciprocal function everywhere the latter is defined would have to be continuous (see [5]). But no such continuous function exists, therefore the reciprocal function can not be extended to a total function in a purely constructive setting. An approach such as described above would totally exclude such intrinsically partial functions from the theory.

The need for a more satisfactory way to formalize partial functions therefore remains. The analysis of the FTA-library, which in particular deals with division and square roots (two good examples of functions which are certainly not total), will be used as a departure point for discussion. Even though both of these functions were treated in this context as *ad hoc* cases and no effort was made at extending the library to include partial functions in their full, these two examples turn out to be generalizable and naturally give rise to two definitions of partial functions. In the next two sections these two definitions will be introduced and their basic properties established. Later, they will be compared regarding their behaviour from the point of view of the formalization.

Other treatments of partiality which might be usable in a formalization occur e.g. in topos theory. These will not be considered here, however, since they lie too far away from the setting of the FTA-library, where this work was done.

3.1 The FTA approach: subsetoids

The simplest example of a partial function is division.

As every child learns early in school, in order to be allowed to write an expression of the form $\frac{x}{y}$ it must first be verified that y is not equal to 0. Of course, in everyday practice one just forgets about explicitly justifying why the divisions are allowed. There are several reasons for this; if y is a concretely given number, say 2, then the fact that it is not 0 is in principle self-apparent, and to provide a justification for it would be perceived by most as silly. But even in other cases, when y is for example an expression such as $(b-a)$, the reader is supposed to "know" that it is not zero (usually from hypotheses which are explicitly or implicitly assumed); if this is not deemed to be obvious, then a small note might be included which explains why it is the case. In any case, it would certainly be treated as a side issue not really relevant to the mathematics being done.

The Calculus of Constructions, like many usual type systems, does not allow terms to be partially defined; in other words, if a function term f has type $\mathsf{A}\rightarrow\mathsf{B}$ then $(\mathsf{f}\ \mathsf{x})$ will be a valid term of type B for any x of type A. Therefore, when formalizing e.g. division, some way must be found to tell the computer that its second argument must be different from zero.

Instead of looking directly at division, attention can be restricted to the reciprocal function, i.e., the function $x \in \mathbb{R} \mapsto \frac{1}{x}$. In the presence of such a function, division can be defined simply via the relation $\frac{x}{y} = x \times \frac{1}{y}$.

One simple way to define the reciprocal function is to formalize it as a function on the set $\mathbb{R} \setminus \{0\}$, viewed as a subset of $\mathbb{R}$, rather than on the whole real line. This was the original approach in the FTA-project, and it has some interesting features which make it easily generalizable to arbitrary partial functions.

The first step along this path is to define a notion of subset. This is formalized in the FTA-library as follows.

DEFINITION 3.1.1 Let $\mathsf{S} : \mathsf{CSetoid}$ and $\mathsf{P} : \mathsf{S} \rightarrow \mathbf{Set}$ be a predicate on (the carrier of) S. Then the carrier of the subsetoid of elements of S that satisfy P has the following record type.

```
Record subcsetoid_crr (S:CSetoid) (P:S→Set) : Set :=
  {scs_elem :> S;
   scs_prf   :   (P scs_elem)}.
```

Intuitively, elements of type (subcsetoid_crr S P) are pairs $\langle \mathsf{x}, \mathsf{H} \rangle$, where $\mathsf{x} : \mathsf{S}$ and $\mathsf{H} : (\mathsf{P}\ \mathsf{x})$. This is written down as (Build_subcsetoid_crr S P x H). This is exactly the form of proof terms of existential statements; this is not a coincidence. In fact, since records are just a syntactical notation for inductive types, the previous structure will generate a type isomorphic to the Σ-type defining the existential quantifier; this correspondence between subsets and existential proofs has often been observed in Type Theory, as is mentioned for example in [48].

Notice that scs_elem is declared as a coercion. This means that, from the user's point of view, elements of subsetoids of S should always be usable where an element of type S is expected, conforming to the standard situation in everyday mathematics; but unfortunately scs_elem does not satisfy the uniform inheritance condition (see [17]) so it must be inserted by hand, though it is not displayed. Also, the parameters S and P will be assumed as implicit arguments, meaning that they will be inserted by the system and do not need to be typed in by the user. Thus, (scs_elem x) will be interpreted as (scs_elem S P x) and displayed as x.

At this stage nothing is required about P. Even though the work on partial functions will later require this predicate to be extensional (as will be discussed further ahead), this definition of subsetoid is perfectly general and there are applications where it is interesting to work with non-extensional predicates. One such example, due to Pollack, is the representation of rational numbers as irreducible fractions, which can be formalized as a subsetoid of the rational numbers seen as pairs of integers with the usual equality between fractions. The characteristic predicate of this subsetoid is *not* extensional. (However, it has also been remarked by Capretta that subsetoids should not be used at all in this situation.)

Furthermore, strong extensionality is even *un*desirable, as the most elementary predicates (in Real Analysis) such as $\lambda_x.a \leq x \leq b$ are not strongly extensional.

The subsetoid apartness, subcsetoid_ap, and equality, subcsetoid_eq, are defined in terms of the apartness and equality on S by simply forgetting the proof terms. In particular, if two different proofs $\mathsf{H1}, \mathsf{H2} : (\mathsf{P}\ \mathsf{x})$ are used to form an element of the subsetoid, the two pairs $\langle \mathsf{x}, \mathsf{H1} \rangle$ and $\langle \mathsf{x}, \mathsf{H2} \rangle$ will be two different representations of the *same* element.

This phenomenon, which will be reoccurring throughout this chapter and is essential to this whole work, is known as the principle of *proof irrelevance*, and states the fact (well known to mathematicians) that the concrete proofs of side conditions which need to be verified before the application of a function or theorem do not in any way influence the output of the function or the result of the theorem.

The concrete definitions of subcsetoid_ap and subcsetoid_eq are omitted here as they bring nothing new to the discussion. It is then straightforward to prove that the structure formed by these relations is a setoid, and these facts are used to define an operator such that, given S and P as in Definition 3.1.1, $\{\mathsf{S}|\mathsf{P}\}$ is the desired subsetoid structure.

An example is the following definition.

DEFINITION 3.1.2 The subsetoid of non-zero elements is defined on any monoid M.

Definition NonZeros (M:CMonoid) : CSetoid := $\{\mathsf{M} \mid \lambda_{\mathsf{x:M}}.(\mathsf{x}\ \#\ 0)\}$.

For simplicity, if $\mathsf{x} : \mathsf{M}$ and $\mathsf{H} : (\mathsf{x}\ \#\ 0)$, then $\mathsf{x}//\mathsf{H}\ :\ (\mathsf{NonZeros}\ \mathsf{M})$ will denote the corresponding subsetoid element[1].

Now, a reciprocal function on a ring R can be defined as a setoid function cf_rcpcl over (NonZeros R) such that, given $\mathsf{x} : \mathsf{R}$ and $\mathsf{H} : (\mathsf{x}\ \#\ 0)$, the lemma $\mathsf{x} \times (\mathsf{scs\_elem}\ (\mathsf{cf\_rcpcl}\ \langle \mathsf{x}, \mathsf{H} \rangle)) = 1$ can be proved.

The type CField of fields is defined in the FTA-library as a record type consisting of a ring and such a function. On these structures division can always be defined.

DEFINITION 3.1.3 Let $\mathsf{F} : \mathsf{CField}$. Then division is defined on F by

Definition cf_div (x:F) (y:(NonZeros F)) : F := $\mathsf{x} \times (\mathsf{cf\_rcpcl}\ \mathsf{y})$.

The term (cf_div x y) is denoted by x/y.

[1]The full representation of this term is (Build_subcsetoid_crr M $\lambda_{\mathsf{x:M}}.(\mathsf{x}\ \#\ 0)$ x H).

In general, an element of (NonZeros F) will not be directly given, but its components (an element y : F and a proof-term H : (y # 0)) will be provided instead. In this situation, the division of x by y will be written down as

$$\mathsf{x/y//H}, \tag{3.1}$$

which is translated internally by Coq into

```
cr_mult F
    x
    (scs_elem F λx:F.(x # 0)
        (cf_rcpcl (Build_subcsetoid_crr F λx:F.(x # 0) y H)))
```

plus some extra coercions to go from F : CField to a term of type CSetoid.

This definition as such is not yet suitable for generalization to an arbitrary partial function. The problem is that the reciprocal function happens to be an automorphism of its domain, but this is not true of partial functions in general; for example, the logarithm function has domain $\mathbb{R}^+$ but image $\mathbb{R}$. Therefore, some fine-tuning must be done before proceeding.

As it happens, there is a very trivial simplification which can be done. If one looks at the expanded form of x/y//H above, there is a projection (the function scs_elem) being applied to the result of the application of cf_rcpcl, which is an element of a subsetoid. So why not simply axiomatize cf_rcpcl as to produce only the first projection as output? This function will then have a different type, as it will no longer be a setoid function on (NonZeros F) but a setoid function from this setoid to F; but in return the expanded form of division will look more simply like

```
cr_mult F
    x
    (cf_rcpcl (Build_subcsetoid_crr F λx:F.(x # 0) y H))
```

plus the same extra coercions as before.

It should be remarked that the mathematical statement $\frac{1}{x} \# 0$ can actually be proved from the condition $x \times \frac{1}{x} = 1$, so the theory with this new definition is in fact equivalent to the old one but the stored terms will be smaller.

And, more interestingly, this definition can now be directly generalized to partial functions by abstracting over the domain.

DEFINITION 3.1.4 Let S : CSetoid. The type of partial functions over S is defined to be the following record type.

```
Record PartFunct :=
  {dom     :   S → Set;
   dom_wd :   (pred_well_def S dom);
   fun     :> (CSetoid_fun {S | dom} S)}.
```

This definition will be referred to as the *subsetoid approach* to partial functions.

The first field of this record is a predicate on the setoid, which will be the condition characterizing the domain of the function. This predicate is required to be extensional; although this is not, strictly speaking, necessary, it is a property that all common examples of function domains share (since domains of most usual functions can be expressed as unions of intervals, and these are always extensional) and that much simplifies further development of the theory. Finally, fun corresponds to the computational part of the function, and it is simply a function from the subsetoid of elements of S satisfying dom back into S. Declaring it as a coercion simplifies the usage of this type, allowing the user to write simply (F x) instead of the more cumbersome and less intuitive (fun F x). Finally, the parameter S will be implicit in all expressions using any of the record projections.

Proof irrelevance and extensionality of partial functions are implicit in the type of fun. The latter is a consequence of its computational part being a setoid function, which is extensional and therefore respects also the setoid equality (as this coincides with the subsetoid equality). The first is taken care of, as explained above, by the fact that the equality on the subsetoid identifies any two terms which just differ on their proof components.

From a mathematical perspective, though, this definition has some unsatisfactory characteristics. The use of subsetoids requires pairing operations to be used which will mix the computational part (the element of the original setoid) with a proof term (to show that it satisfies the characteristic predicate), and this is very unmathematical. As noted above, the usual approach to partiality consists of checking that side conditions are met, and *not* mixing them with the data with which one is working.

Also, as the example of division clearly shows, the user will most often provide the argument and the proof term separately, so an extra pairing operation will have to be inserted before applying the function; and most algorithms will require these two terms separately at different places, which means that the first thing they will do will be to unpair these terms. This turns out to have some very undesirable consequences in practice which will be discussed in the sequel.

Another particularly annoying problem arises with restriction of functions. Suppose that F : (PartFunct S) represents a partial function f, with

$\mathsf{S} : \mathsf{CSetoid}$, and that A is included in the domain of f. This last condition is expressed by the existence of a term $\mathsf{H} : \forall_{\mathsf{x}:\mathsf{S}}.(\mathsf{A}\ \mathsf{x}) \to (\mathsf{dom}\ \mathsf{F}\ \mathsf{x})$.

In informal terms, if x is an element of $\{y \in S|A(y)\}$, then x is a pair, with $\pi_1(x) \in S$ and $\pi_2(x)$ a proof of $A(\pi_1(x))$. To apply f to x one needs to build an element of the setoid $\{y \in S|\mathrm{dom}_f(y)\}$. This will in turn be a pair whose first component is $\pi_1(x)$; and a proof of $\mathrm{dom}_f(\pi_1(x))$ is for example $H(\pi_1(x), \pi_2(x))$. Hence, $f|_A$ can be represented by the term

$$\lambda_{x \in \{y \in S|A(y)\}}.f(\langle \pi_1(x), H(\pi_1(x), \pi_2(x))\rangle).$$

This can be translated almost directly in Coq. According to the previous definitions, π_1 is scs_elem and π_2 is scs_prf; therefore the functional part of $f|_A$ can be formalized as

```
λx:{S|A}.
    (fun F
        (Build_subcsetoid_crr S (dom F) (scs_elem x)
            (H (scs_elem x) (scs_prf x))))
```

which is strongly extensional because F is a partial function.

The worrying part is that if the expressions for $f(x)$ and $f|_A(x)$ are now built, given an x such that both applications make sense, proving that they are equal will be painstakingly slow. Even though f and $f|_A$ intuitively have the same computational part, they are represented by very different terms; and to prove the desired equality extensive $\delta\beta\iota$-reduction is required on both of them to yield equal expressions, as well as resource to the extensionality of f.

Although this is not a serious drawback, it is somewhat counter-intuitive and time consuming; and this issue of time consumption is a serious one which, rather than arising out of theoretical considerations, was identified through the practical problems encountered during the formalization. At two distinct places simplifications were needed to prove (mathematically) obvious statements, and these turned out to be too difficult for Coq, causing the system to run out of memory.

The first problematic point was proving the chain rule for the derivative of the composition of two functions. This proof involves manipulating some equalities involving two real numbers x and y. These appear both as arguments to two partial functions with a given domain and as real numbers which have to be subtracted and multiplied; this requires going back and forth between the real numbers x and y and (intuitively the same) subsetoid elements $\langle \mathsf{x}, \mathsf{H}_\mathsf{x}\rangle$ and $\langle \mathsf{y}, \mathsf{H}_\mathsf{y}\rangle$ of the domains of the said functions. As a result, some proof steps (the most time-consuming ones) are needed simply to replace e.g. $\pi_1(\langle \mathsf{x}, \mathsf{H_x}\rangle)$ with x. These steps are extremely inelegant and counter-intuitive, but necessary.

A different problem arises when proving the additivity rule for the integral, that is,

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx,$$

assuming that $a \leq c \leq b$ and f is continuous in all three intervals. The "natural" way to define the integral on a compact interval I is for functions whose domain is exactly I, so the expression above will actually correspond to

$$\int_a^b f|_{[a,b]}(x)dx = \int_a^c f|_{[a,c]}(x)dx + \int_c^b f|_{[c,b]}(x)dx.$$

Now a problem arises: to prove the required equalities, one eventually gets down to proving that $f|_{[a,b]}(x) = f|_{[a,c]}(x)$ for $x \in [a,b]$ (and similarly for $x \in [b,c]$), which should be trivial. However this requires unfolding the formal definition of $f|_{[a,b]}$ and $f|_{[a,c]}$ and checking that they only differ in the occurring proof terms; but these proof terms have such long expressions that, at the time when this work was done, Coq ran out of resources while performing the simplification.

3.2 The Automath way: explicit proof-terms

The discussion at the end of the previous section apparently suggests that the interaction between computational and non-computational objects, or between data and proofs, has a negative effect on both the clarity and simplicity of the notation and on the internal representation of partial functions and actual performance of the proof engine. In this section a different example from the FTA-library, the square root, will be used as departure point for the analysis of a totally different way to look at partial functions which was originally used in the Automath system, as is explained for example in [7].

It is worth mentioning that there is no intrinsic reason for the difference of treatment between division and square root in the FTA-library; square root could equally well have been formalized in a similar way to that above presented, using subsetoids, and division could also be defined following the approach about to be described. The difference is a mere coincidence, brought about by the facts that there was no *a priori* emphasis on treating the general case of partial functions and that different people with different ideas participated in the FTA-project.

The square root on the real numbers is only defined for non-negative numbers, i.e., numbers which are greater than or equal to zero. The definition of this function in the FTA-library is done in several steps. First, a lemma is proved that says that n^{th} roots of non-negative real numbers exist, for any $n > 0$. This is formalized as an existential statement.

Lemma nrootIR : $\forall_{c:\mathbb{R}}.\forall_{n:\mathbb{N}}.(0 \le c) \rightarrow (0 < n) \rightarrow \exists_{x:\mathbb{R}}.(0 \le x) \wedge (x^n = c)$.

The square root is then defined by filling in some of these arguments with fixed parameters: n is 2, and an explicit proof term lt_z_two : $(0 < 2)$ can be given. Here, ProjS1 is the projection that extracts the witness from a proof of an existential statement.

Definition 3.2.1 Let $\mathbb{R}$ be a real number structure. The square root is defined[2] on $\mathbb{R}$ as follows.

Definition sqrt (x:$\mathbb{R}$) (H : $(0 \le x)$) : $\mathbb{R}$:= ProjS1 (nrootIR x 2 H lt_z_two).

An immediate consequence of this definition is that the square root will become a *binary* function: besides its actual argument x, it also requires as input a proof that this x is positive.

From a purely mathematical perspective, this is a very strange situation. But it is far from being a novelty; as early as in the Automath project, more than thirty years ago, partial functions were formalized in precisely this manner, as described in [7]. This is therefore considered by some to be the "standard" way to model partiality in Type Theory.

In order to yield a workable theory (also from the mathematician's point of view), the presence of the proof term should however play no role: the proof irrelevance which has already been discussed. This is not a big problem in the setting of the FTA-library since, in constructive mathematics, extensionality must always be proved anyway. In this case, this reduces to the following statement.

Lemma sqrt_wd : $\forall_{x,y:\mathbb{R}}.\forall_{\mathsf{xpos}:(0\le x)}.\forall_{\mathsf{ypos}:(0\le y)}.$
$(x = y) \rightarrow (\mathsf{sqrt}\ x\ \mathsf{xpos}) = (\mathsf{sqrt}\ y\ \mathsf{ypos})$.

Unfortunately, because of the presence of the proof terms, the square root cannot be used to define a setoid function. Therefore, the properties of setoid functions do not apply to it, and in particular this lemma has to be used every time extensionality of the square root is needed. For this reason this definition is not totally satisfactory.

On the positive side, plugging in the same term t for x and y but different proof terms H1 and H2 that it represents a nonnegative real number, this

[2] Notice that this definition by itself does not yield a function but an operation, since it need not be strongly extensional.

lemma directly yields (sqrt t H1) = (sqrt t H2), which is the desired proof irrelevance.

It should be pointed out that proof irrelevance, rather than being directly shown, is a trivial consequence of extensionality; and this is in turn a corollary of strong extensionality, as was discussed in Section 2.2.

Strong extensionality seems then to be the key property from which everything else can be proved. This suggests the following simple definition of partial function, known as the *propositional* approach.

Definition 3.2.2 Let S : CSetoid. The type of partial functions over S is defined to be the following record type.

Record PartFunct : Type :=
  {dom : $\mathsf{S} \to \mathbf{Set}$;
  dom_wd (pred_well_def S dom);
  fun :> $\forall_{x:S}.(\mathsf{dom}\ x) \to \mathsf{S}$;
  strext : $\forall_{x,y:S}.\forall_{Hx:(\mathsf{dom}\ x)}.\forall_{Hy:(\mathsf{dom}\ y)}.(\mathsf{fun}\ x\ Hx) \# (\mathsf{fun}\ y\ Hy) \to (x \# y)$}.

A comparison with the subsetoid approach is in place. The first two fields are the same as before, focusing on the same idea that there is an explicit predicate characterizing the domain of the function and that this predicate should be extensional, for the reasons previously discussed. But now the functional part has a simpler type, being an arrow type instead of a more complicated setoid function type. The price to pay is the need for the extra field, which states that this functional part really defines a strongly extensional function. Notice however that this condition was in fact present in the previous definition inside the type of the functional part of the record.

As before, the parameter S will be assumed to be an implicit argument.

In line with what was said earlier, the following lemma is easily provable.

Lemma 3.2.3 Let S : CSetoid and F : (PartFunct S). Then there are terms

$$\mathsf{pfwdef} : \forall_{x,y:S}.\forall_{Hx:(\mathsf{dom}\ F\ x)}.\forall_{Hy:(\mathsf{dom}\ F\ y)}.(x = y) \to (F\ x\ Hx) = (F\ y\ Hy).$$

and, as a consequence,

$$\mathsf{pr\_irr} : \forall_{x:S}.\forall_{Hx,Hx':(\mathsf{dom}\ F\ x)}.\ (F\ x\ Hx) = (F\ x\ Hx').$$

It should once again be pointed out that the user will usually provide the argument and the proof term separately. This definition works then much better than the previous one: given S : CSetoid and F : (PartFunct F), if x : S and H : (dom F x) then the functional application of F to x can be directly written as (F x H) without further ado; Coq will insert the appropriate coercion, and the complete term will be simply

$$\mathsf{fun\ S\ F\ x\ H}$$

which looks much nicer than the previous representation. Division has been formalized also with this definition, for the purpose of the comparison; the same notation as before can be used, but now the full expression of the term x/y//H, which should be compared with (3.1), is now

$$\mathsf{cr\_mult\ F\ x\ (fun\ (cf\_rcpcl\ F)\ y\ H)}$$

plus the extra coercions needed to take F to a term of type CSetoid.

This shorter representation, which is an intrinsic property of this definition of partial function and holds not only for the case of division but in general, provides faster computation and lower simplification times. In the next section, both definitions are compared from different perspectives in more detail. Before that, though, it will be shown that this different approach in fact avoids most of the problems discussed at the end of the previous section.

One of the criticisms made to the subsetoid approach was the fact that it mixed computational and proof terms in an unnatural way at the time of function application, due to the use of subsetoids. The propositional approach totally avoids this problem, since proof terms are a separate argument. In fact, experience shows that during interactive theorem proving this approach also interacts with the user in a nicer way, since this separation between computational arguments and proof terms makes it much easier to ignore the latter, which the mathematician would not even want to see.

Another nice consequence of this separation between computational arguments and proofs is that simplification of expressions becomes much faster. Both problems discussed at the end of the previous section were solved simply by changing to this definition of partial function.

A specific case is when the partial function being applied happens to be total. Then, the proof argument is immediately thrown away—which in the previous case still required applying one projection and performing one ι-reduction step, but is now immediate.

The problem of restricting functions is also solved. As this is a process which only changes the domain of the function, it can be modeled simply by composing proofs (since identifying domains with predicates translates inclusion into implication); and this can now be done without ever touching the first argument. As before, suppose that $\mathsf{F} : (\mathsf{PartFunct\ S})$ represents a partial function f, with $\mathsf{S} : \mathsf{CSetoid}$, and that $\mathsf{H} : \forall_{\mathsf{x:S}}.(\mathsf{Ax}) \rightarrow (\mathsf{dom\ F\ x})$. Let $\mathsf{x} : \mathsf{S}$ with $\mathsf{Hx} : (\mathsf{A\ x})$. Reasoning informally, f can now be directly applied to x, while the proof term needs only a small transformation; and the restriction

$f|_A$ can hence be represented by the lambda term

$$\lambda_{x\in S}.\lambda_{Hx}.f(x, H(x, Hx)).$$

In Coq, this translates as the much simpler than before

$$\lambda_{\mathsf{x:S}}.\lambda_{\mathsf{Hx:(A\ x)}}.(\mathsf{fun\ F\ x\ (H\ x\ Hx)}) \tag{3.2}$$

which trivially defines a strongly extensional function. Furthermore, if the term for $f|_A(x)$ is written down, then *two* steps of $\beta\delta$-reduction[3] are enough to produce a term which only differs from that for $f(x)$ in the proof argument, and proof irrelevance can be now used to deduce that they represent extensionally equal elements.

3.3 Comparative analysis

The subsetoid approach and the propositional approach exemplify the two main ways to formalizing partiality in Type Theory which are consistent with constructive mathematics. In this section, the properties of these definitions will be examined in more detail and they will be compared in terms of expressiveness. Other approaches, such as assuming every function to be total and ignoring its behaviour on points outside the intended domain (as described by Mayero in [49]), are not constructively valid and will therefore not be considered.

Complexity and Expressive Power

It should be intuitive that the two definitions of partial function being considered are equivalent, as the propositional approach is simply an unfolding coupled with simplification of the notion of setoid function used in the subsetoid approach. A more formal proof of this fact will now be given.

The duality between the subsetoid approach and the propositional approach has been discussed in some detail by Carlström in [14] focusing, as was done above, on the particular case of the reciprocal function.

In the afore-mentioned paper, Carlström shows that representing the reciprocal function using subsetoids and as a propositional function yields for any $x \ \# \ 0$ two different representations for $\frac{1}{x}$, one of which is a $\beta\delta$-reduct of the other, and he argues that because of this it can be much more efficient to use the reduced representation.

[3]Not only the fact that only two steps are needed is important, but also that ι-reduction is avoided, as this is the most time-consuming of the three.

This argument can be made for the general case. Let S : CSetoid, suppose F : (PartFunct S) is a *defined* partial function (i.e., *not* a variable of that type), and x : S with H : (dom F x). Assuming Definition 3.1.4 (subsetoid approach) for the type of F, the form of the functional application of F to x will be

$$\mathsf{fun\ F\ (Build\_subcsetoid\_crr\ S\ (dom\ F)\ x\ H)}$$

According to this definition, (fun F) has type

$$\mathsf{CSetoid\_fun}\ \{\mathsf{S} \mid (\mathsf{dom\ f})\}\ \mathsf{S}$$

which is not immediately a function type. In order for this term to type-check, the appropriate projection of this record type (which is a coercion) has to be inserted. This is the term

$$\mathsf{csf\_fun} : \Pi_{\mathsf{S1,S2:CSetoid}}.(\mathsf{CSetoid\_fun\ S1\ S2}) \to (\mathsf{S1} \to \mathsf{S2});$$

the original expression in full form thus reads

```
csf_fun (Build_SubCSetoid S (dom F)) S
    (fun F)
    (Build_subcsetoid_crr S (dom F) x H)
```

where there are six $\delta\iota$-redexes: δ-reduction of F will give a term of the form

$$\mathsf{Build\_PartFunct\ S\ D\ D\_wd\ f}$$

by the rules of inductive types, and this yields a ι-redex with both fun, which reduces simply to f (the setoid function implicit in F), and dom, which reduces to D (its domain) in two places. The resulting term is

```
csf_fun {S | D} S
    f
    (Build_subcsetoid_crr S D x H)
```

But the type of f is itself inductive, which means that f must in turn δ-reduce to a term of the form

$$\mathsf{Build\_CSetoid\_fun}\ \{\mathsf{S} \mid \mathsf{D}\}\ \mathsf{S\ f'\ f'\_strext}$$

and this forms a ι-redex together with csf_fun, simplifying to f′. That is, the original term (fun F x H) becomes, after $\delta\iota$-simplification, simply

$$\mathsf{f'\ (Build\_subcsetoid\_crr\ S\ D\ x\ H)}$$

and most of the proof terms have been thrown away.

Finally, to compute f′ one will in general need to access both x and H, and the previous expression will often $\delta\iota$-reduce to a term of the form (F′ x H).

If one pursues the same reasoning assuming PartFunct to be defined according to Definition 3.2.2 (propositional approach), then the application of F to x looks like

fun F x H

where, as before, there is a $\delta\iota$-redex. Because (PartFunct S) is an inductive type, F must δ-reduce to

Build_PartFunct S D D_wd F' F'_strext

and like in the previous case this can now be ι-reduced by fun, whence the whole term simplifies to (F′ x H).

In conclusion, both definitions will produce exactly the same reduced terms when a concrete partial function is being applied. However, the corresponding reduction sequences deserve a comparison: the first one is

$f(x)$ is the term
csf_fun {S | (dom F)} S
(fun F)
(Build_subcsetoid_crr S (dom F) x H)
$\to_\delta$ csf_fun {S | (dom (Build_PartFunct S D D_wd f))} S
(fun (Build_PartFunct S D D_wd f))
(Build_subcsetoid_crr S (dom (Build_PartFunct S D D_wd f)) x H)
$\to_\iota$ csf_fun {S | D} S
f
(Build_subcsetoid_crr S D x H)
$\to_\delta$ csf_fun {S | D} S
(Build_CSetoid_fun {S | D} S f' f'_strext)
(Build_subcsetoid_crr S D x H)
$\to_\iota$ f' (Build_subcsetoid_crr S D x H)
$\to_{\delta\iota}$ F' x H

whereas the second is simply

$f(x)$ is the term
fun F x H
$\to_\delta$ fun (Build_PartFunct S D D_wd F' F'_strext) x H
$\to_\iota$ F' x H

In short, the subsetoid approach requires ten steps of $\delta\iota$-reduction (five of each type), while the propositional approach only requires two (one δ-step, one ι-step). Also, the intermediate terms in the first case are much longer, and will therefore take more space in memory. This allows the following conclusion to be drawn: the propositional approach is more efficient than the subsetoid approach if simplification is needed.

The practical implications of this difference in efficiency will also be discussed in Chapter 6.

From the theoretical perspective, this has another important consequence. The reasoning above can be precisely formalized in Coq (using the `inversion` tactic) to define a map from the record type of Definition 3.1.4 to that of Definition 3.2.2.

In order to go in the other direction, Definition 3.1.1 (subsetoid) is needed together with some packaging. Given $\mathsf{F} : (\mathsf{PartFunct\ S})$ and $\mathsf{x} : \{\mathsf{S} \mid (\mathsf{dom\ F})\}$, the term $(\mathsf{fun\ F}\ (\mathsf{scs\_elem\ x})\ (\mathsf{scs\_prf\ x}))$ is well typed and abstraction on x, using the strong extensionality of F, defines a setoid function to S. Together with $(\mathsf{dom\ F})$ and the proof that this predicate is extensional a partial function in the sense of Definition 3.1.4 is obtained. This completes the proof of the following theorem.

Theorem 3.3.1 The representations of partial functions via the subsetoid approach and via the propositional approach are equivalent.

As discussed at the end of Section 3.2, though, the *performance* of both definitions is quite different. The next paragraphs examine other aspects in which these definitions differ.

The user's perspective

A different way to compare different formalizations of the same concept is via their user-friendliness. This section aims at doing this for both definitions of partial function above presented.

The typical situation in mathematics when one is confronted with the issue of partiality is when an expression must be built which includes application of (potentially) troublesome functions—for example, taking a square root or a logarithm. Usually, in such a situation, a comment might be inserted as a footnote or in the text justifying that such a function application is legal.

In formalizing mathematics within Coq, this situation occurs whenever the context includes terms $\mathsf{F} : (\mathsf{PartFunct\ S})$, with $\mathsf{S} : \mathsf{CSetoid}$, and $\mathsf{x} : \mathsf{S}$ and the user wants to apply F to x. Regardless of which definition of partial function is being used, this requires the introduction of a proof term of type (dom F x).

In Coq one can use the tactics `cut` or `assert` to introduce such a hypothesis in the context and prove it (i.e., provide an explicit term of the required type) in an apart subproof. This provides a very good approximation to the mathematical approach, where proving that expressions are defined is in general kept separate from the main proof, although this kind of forward reasoning is not what Coq is best designed to do. Proof irrelevance ensures that the specific form of this term can make no difference in what follows; and this technique allows the user to work with smaller terms (as hypotheses typically have names consisting of a few letters, while proof terms are in general long) where the proof parts will not make reading the expressions a herculean task.

When the term that the user must type is considered, however, there can be some difference in both definitions.

If the subsetoid approach is used, the argument to the partial function must be a subsetoid element, hence a pairing operation must still be done. Of course, syntactic definitions can be used (and these have been implemented) to do part of this automatically, since the implicit arguments mechanism of Coq is strong enough to figure out the relevant information.

Following the propositional approach, the user can directly type the term in as a functional application; and no further processing of the input term is required other that the insertion of the fun coercion.

Therefore, from the perspective of the mathematically minded user, both definitions are equally satisfactory: in any case, a cut needs to be made to insert the necessary additional hypothesis in the context and prove it in a separate subproof; and thereafter the desired term can be input as a direct functional application. However, while this directly yields a legal term (modulo one coercion) when the propositional approach is used, the subsetoid method requires some extra notation to be defined in order for the same term to be accepted by Coq. This means, as was noted before, that the term internally built and stored will be bigger in the latter case.

In the future, it would also be interesting to devise a mechanism that would allow one just to write (f x) and have the system automatically insert the appropriate proof term, either finding it using some tactic or cutting it and inserting it into the context. This would bring working with partial functions in Coq closer to the situation in systems like PVS from the user's perspective. While such a tool does not exist, and implementing it would require changing the parsing mechanism of Coq, it sounds plausible that it might one day come to be; and once again it seems that the propositional approach to partial functions would prove better for this task.

Abstraction

From what was said so far, one might get the impression that the packaging of concepts into record structures and unpacking in order to use them is never a good idea. However, this is far from being the case. The whole idea of using subsetoids to formalize division corresponds to what is done most often in type theory in similar situations; both the type theory of PVS [61] and that of Nuprl [16] rely on side conditions on the typing rules to be able to deal with partial functions, and these conditions are determined from the type of the arguments of the function.

The difference in the underlying type systems, when compared to Coq, is in practice translated in the nonexistence of the explicit pairing and projection operators encountered in Definition 3.1.1; the price to pay is that type checking in these type theories becomes undecidable, because in order to be able to type (f x) one has to find a proof of the side condition $x \in \mathrm{dom}_f$ generated by the typing rule.

Compared to the definition of partial functions as propositional functions, this approach is from a methodological point of view more structured.

On the one hand, the ability to use the existing definition of setoid function makes it clear that the (strong) extensionality of partial functions is really the same thing as (strong) extensionality of total functions—something that is not at all obvious in Definition 3.2.2. Furthermore, the domain of a function is never seen as a setoid in its own right when the propositional approach is used, whereas it is very explicitly made so using the subsetoid mechanism.

On the other hand, algebraic structures are usually built in a structured way that heavily relies on the use of records and projections. Such an approach has been taken not only in the Algebraic Hierarchy of the FTA-project [32] but also more recently in that of Metaprl [64]; in both situations this has been seen as a major advantage (and a more abstract way to work) rather than as a drawback.

Why then did problems arise when the same method was applied to the definition of partial functions? One very clear reason is the fact that functions are intrinsically computational objects where one will always want and need to perform reduction and simplification, whereas algebraic structures are by their own nature objects one wants to talk about but never compute with. In fact, all the reasons previously presented relate to the computational behaviour and performance of both definitions; this will once again be relevant in Chapter 6. Unless computation is relevant (and, as was argued, in this case it was a fundamental issue), there is little doubt that the more abstract approach using records extensively is probably the most satisfactory.

Compatibility

In Section 3.1, it was shown how the definition of division originally present in the FTA-project could be generalized in a natural way to produce a notion of partial function of which division would then be a particular instance. Later, in Section 3.2, the example of the square root was treated in a similar way and a second (equivalent) definition was obtained of which the definition of the square root in the FTA-project was but a special case.

One question that naturally arises is the following: if these notions of partial functions are indeed generalizations of those particular concepts, does this mean that the formalization can be changed using these new concepts without making significant changes to the proofs?

In this section not only a positive answer to this question is given, but it is also argued that the new formalization is even simpler than the original one.

An important aspect related to this question is how the behaviour of the automated tactics is affected by these changes. At this stage it is enough to say that these can be updated without too much effort to a version which works equally well in this new setting. These changes will be thoroughly described in Chapter 4 when tactics are discussed in detail.

Changing the existing library to use the general notion of partial function means in fact changing the definition of division, in the first case, and of square root, in the second, to make them instances of partial functions; and then appropriately change everything that is built using these functions so that the whole library remains consistent. As in both situations the methodology is quite similar, only the slightly more complicated case of division will be discussed.

The first step to take is to redefine division. This requires simply changing the definition of the reciprocal function in the definition of CField so that $(\mathsf{cf\_rcpcl}\ \mathsf{F})$ will have type $(\mathsf{PartFunct}\ \mathsf{F})$. Since records with defined fields are not allowed in Coq, this was done by requiring only the functional part of this function together with the proof that it is strongly extensional. The definition of CField thus becomes the following.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Record\ CField : Type :=}} \\
\quad \{\mathsf{cf\_crr} & :> & \mathsf{CRing}; \\
\quad \ \ \mathsf{cf\_rcpcl} & : & \Pi_{\mathsf{x:cf\_crr}}.(\mathsf{x} \# 0) \to \mathsf{cf\_crr}; \\
\quad \ \ \mathsf{cf\_rcpsx} & : & \forall_{\mathsf{x,y:cf\_crr}}.\forall_{\mathsf{x\_:(x\#0)}}.\forall_{\mathsf{y\_:(y\#0)}}. \\
 & & \quad (\mathsf{cf\_rcpcl\ x\ x\_}) \# (\mathsf{cf\_rcpcl\ y\ y\_}) \to (\mathsf{x} \# \mathsf{y}); \\
\quad \ \ \mathsf{cf\_proof} & : & (\mathsf{is\_CField\ cf\_crr\ cf\_rcpcl})\}.
\end{array}
$$

Then a partial function $\mathsf{f\_rcpcl}$ is built with computational part $\mathsf{cf\_rcpcl}$, and division is defined as

Definition cf_div (F:CField) (x y:F) (y_:y # 0) : F := x × (f_rcpcl y y_).

The reason for not defining this directly in terms of cf_rcpcl is a practical one: with this definition, the types of its sub-terms ensure extensionality of the resulting function, which otherwise would be less straightforward to prove. In practical terms this definition works much more nicely.

The immediate effect of this change is that the construction of instances of reciprocal functions must now be changed. This is present in two different places of the FTA-library: in the construction of the Cauchy model of the real numbers; and in the definition of a field of complex numbers from an arbitrary real number structure. To define F : CField from R : CRing, one needs now to give a term of type $\Pi_{x:R}.(x \# 0) \to R$ and a proof that this is well defined. But *both of these components are already present in the old formalization*:

- the function: the functional part of the old reciprocal function had type (NonZeros R) → (NonZeros R). This can be directly translated to a function of type $\Pi_{x:R}.(x \# 0) \to R$ by applying the pairing operation, then the original implementation of the function, and then a projection. In practice, it turns out that this can be done in an even better way: since all implementations of cf_rcpcl in the FTA-library started by unpacking its argument, the pairing step can be removed altogether; and instead of adding a projection at the end it is of course simpler to omit the part that builds the second component of the pair;

- the proof of strong extensionality: this was also already present, being required for setoid functions; and since the new definition of the function follows the old one so closely the existing proof script can always be directly used.

The only thing lost is the fact that $\frac{1}{x} \# 0$, which was the second component of the pair built by original models of cf_rcpcl. Once again, this is not a problem: the corresponding part of the original definition can directly be used to provide a proof of that lemma for the specific implementation. Better yet, the lemma can be algebraically proved just from the (required) fact that $x \times \frac{1}{x} = 1$.

The next step is to recover the basic properties of the reciprocal function. This turns out to be extremely easy, because once again they depend mostly of the axiomatic properties of the reciprocal function, i.e., of the two properties $\frac{1}{x} \# 0$ and $x \times \frac{1}{x} = 1$. The only changes that need to be made refer to the fact that the output of cf_rcpcl now has type F and not (NonZeros F); and rather surprisingly this actually *simplifies* many of the proofs.

The definition of division is then also simplified as was shown in the discussion following Definition 3.1.3. Once again, this requires some (very) minor changes in the immediate lemmas, which state the basic properties of division; and these few changes usually amount, as before, to simplifications.

The rest of the library, amazingly enough, requires almost no further changes. This may come at first as a surprise, but is actually a direct consequence of the modular approach followed throughout the whole FTA-project.

The process for changing the representation of the square root function is even simpler, as this is a defined function on an arbitrary real number structure. Here only the proofs of the basic properties need to be changed.

Typing issues

There is however one unsatisfactory characteristic of both definitions of partial functions above presented, and that is their type. According to the typing rules of the Calculus of Constructions, (PartFunct S) has type **Type** for every setoid S; intuitively this arises because the type of partial functions over S is parameterized on subsets of S, and becomes therefore “too large” to still fit into **Set**.

In practice, this has two undesirable consequences.

The first one is that it is impossible to formulate existential statements of the form “there is a partial function f such that…” directly. The type T corresponding to such a statement would itself forcefully live in **Type**, and it would not be possible to extract f from a proof-term of type T. In Section 5.3 a way around this problem is presented for a specific situation (actually the only one which actually arose during the formalization).

The second problem is that the collection of partial functions over a setoid S also forms a setoid, which both inherits part of the algebraic structure of S and has a monoid structure itself (with composition as binary operation). Once again, this is not directly representable in the FTA-library, as the carrier of any setoid must itself have type **Set**. This question will be addressed again in Section 5.1.

3.4 The final choice

Hopefully this chapter will have made clear that a uniform concept of partial function is not only interesting from the theoretical sense and important for the formalization of Real Analysis, but is also desirable as it allows the

FTA-library, in particular the Algebraic Hierarchy, to be made more uniform, as functions such as division and square root can now be viewed as instances of a more general concept.

The discussion in the last section should also make it clear that while the subsetoid approach to partial functions is arguably a more abstract one and more natural from the mathematical point of view, it actually produces results of a poorer quality from various perspectives. Not only does the representation yield larger terms which take much longer to simplify, but also from the user's point of view there are some unmathematical things going on.

Therefore, it should come as no surprise that the propositional approach prevailed and Definition 3.2.2 was chosen as *the* definition of partial function to be used in extending the FTA-library. In Chapter 5 it will be shown how the Algebraic Hierarchy can be extended with this concept and how Real Analysis is developed thereupon. In Chapter 6 the efficiency issues will be brought up again, and the propositional approach will yet again be shown to be superior in quality to the subsetoid approach.

Chapter 4

Automation

As any person attempting at formalizing mathematics using a proof assistant or theorem prover soon discovers, having a nice theoretical basis and well thought definitions is not enough. Almost immediately, one finds oneself constantly spending time proving boring and mathematically obvious results, which in any informal presentation would be ignored or deemed as obvious but for which any computer system will require a proof.

It should therefore come as no surprise that all successful proof assistants nowadays have built-in automation tools. These exist in many different styles, as can be seen from a few examples.

- In the Mizar system [51], the automation is totally built in. All steps in the proof are followed by a justification (`by`), which is checked by a highly complex mechanism which the user cannot control.

  The type system of Mizar is also quite complicated, and the type checking algorithm can perform some complex reasoning to verify that terms are well typed.

  Apart from these two mechanisms, there is no extra automation available, and there is no interface which allows the user to define his own routines.

- The PVS system is very similar in mostly not allowing the user to control or see the details of what its automatic procedures (known as *strategies*) do. However, there are tools which allow the user to define new strategies, hence making the system more powerful. These new strategies can be defined totally inside the language of the system, but the user is also allowed to write LISP code (thereby accessing the level of the implementation).

  A more detailed description of these mechanisms can be found in [61].

- The HOL and HOL-light systems are similar to each other in this respect. In these systems, the user interacts with the proof assistant directly via an ML programming environment shell.

  Interaction happens through *proof procedures*, which are described in detail in [62], for the HOL system, and in [38] for HOL-light. These proof procedures are in reality no more than small ML-programs; defining new automation procedures is simply writing new programs. Thus, the standard user can easily define his own routines and control their behaviour in an easy way.

- The behaviour of Isabelle, by contrast, is totally controlled by the user, as can be seen in [56]. There are automation tactics available which can be parameterized (e.g., there is a search tactic which can be given names of lemmas to include in the search space), but like in Mizar the user has no way to add his own procedures to the system.

Compared to these different systems, Coq lies at a midpoint. On the one hand, there is quite a bit of automation available in the system, not only decision procedures for specific domains but also generic tactics which are parameterized and can easily be made more powerful. On the other hand, Coq is open source, so it is, at least theoretically, always possible to descend to the level of the implementation and add new tactics to the system written directly in the ML programming language. Furthermore, since version 7.0, there is an easy-to-use tactic language which also allows the user to define altogether new tactics without needing to jump into the implementation language. These user-defined tactics can be quite powerful, as will be exemplified.

In the following section several different approaches to automation in Coq will be described, compared and exemplified through tactics present in the standard distribution of Coq. Then it will be shown how the same methods were used in the FTA-project to define two tactics (`Algebra` and `Rational`) to help formalize equational reasoning; these were extended in C-CoRN so that they would work within the Algebraic Hierarchy extended with the partial functions of Definition 3.2.2. Equational reasoning in C-CoRN was also made easier thanks to the development of the `Step` tactic, whose evolution from its original version in the FTA-library will be presented.

Afterwards, some of the tactics specific to C-CoRN, dealing with typical problems in Real Analysis, will be shown and discussed: `Included`, `Contin` and `Deriv`.

The chapter ends with a comparative analysis of the different possibilities for automation in Coq.

4.1 Automation in Coq

In this section the different ways to automate reasoning in Coq will be discussed and illustrated with examples from the standard distribution of Coq.

One way to classify the Coq tactics, due to Wiedijk, is along the following two orthogonal lines.

- *Search techniques* vs. *decision procedures*: on the one hand, search methods from Artificial Intelligence can be used to define general-purpose tactics which search parts of the library according to some algorithm and eventually find a proof term of the desired type.

  On the other hand, inside specific domains one can directly program a decision procedure as a tactic that will also produce a valid proof of goals of some particular family.

- *Internal* vs. *external* tactics: tactics of the first kind are totally defined in the type theory, and they can be proved correct inside the system; this has as a consequence that they will usually yield more compact proof terms.

  By contrast, in some situations it is easier (or more efficient) to write a program (outside the Calculus of Constructions) that will directly output a proof term for goals of some form. The price to pay for this is having to extensively type-check every term thus generated; and the method cannot be proved correct inside the system.

This situation is graphically depicted in Figure 4.1, where examples of tactics of each kind are presented. Some of these tactics will be discussed in the sequence. More detailed information on these tactics can be found in the Coq Reference Manual [17]. Notice the absence of internal search tactics: by their own nature, search tactics look for a proof term, which is much more efficient to do outside of the system.

	Internal	**External**
Search	—	auto (with hints)
Dec. Proc.	ring field romega	omega tauto intuition

Figure 4.1: Classification of Coq tactics

Guided search: `auto` with hints

The easiest automation tool to use among those that are available in Coq is doubtless the `auto` tactic family. Tactics in this family automate in a generic way reasoning in a variety of domains.

At the basis of `auto`'s functioning lie a number of hints databases. These contain several already proved lemmas, categorized according to their field of application; the databases provided with the standard distribution of Coq include lemmas on natural numbers (`arith`), integers (`zarith`) and booleans (`bool`), besides a special database `core` which contains very basic facts about the logical connectives.

Within each database, lemmas are categorized according to the goals to which they can be applied: a lemma of type $\Pi_{x_1:A_1}.\ldots\Pi_{x_n:A_n}.(T t_1 \ldots t_k)$ will be classified as "a lemma that can be applied if the head of the goal is T".

Finally, each lemma is assigned a cost, defined to be the number of subgoals that its application generates—which is the number of non-dependent products in its type.

By default, `auto` uses only the `core` database along with the hypotheses in the context. The user can override this by calling `auto with` *`database`* `...` *`database`*. Then, the system implements a limited depth, Prolog-style depth-first search for a term which has as type the current goal. The available lemmas are tried in order of cost, so that the produced proof is (hopefully) as short as possible. If no such term is found, the tactic leaves the goal unchanged.

Practice shows that this tactic is very good at solving simple goals, namely those which follow from the context by higher-order unification. However, if the proof starts to become more complicated, and in the presence of a moderately large hints database, `auto` might not only not find it, but also take a very long time before discovering it cannot do so.

Furthermore, this tactic is somewhat unstable and occasionally displays erratic behaviour. It has been noticed to change its result when the place of a lemma is changed within a file, for example; or when used in combination with `Undo` in specific contexts. These are, however, minor issues that do not significantly hamper the development.

External decision procedures

In many specific areas there are decision procedures which can decide whether a given statement is a valid theorem. These are natural candidates to encode as tactics in theorem provers, solving the problem of proof search for concrete domains.

Often these decision procedures are just too complicated to implement directly in the type theory. When this is the case, one usually has the option of descending to the implementation level of Coq and implementing the decision procedure as an ML program.

An example of such a tactic is Coq's `tauto`, a decision procedure for intuitionistic propositional calculus described in [17, Section 8.11.4]. This tactic can solve any goal that follows from the context by purely propositional (intuitionistic) reasoning. Although its scope of application is not very wide, it usually does a good job when it can be used, quickly providing proofs of not-so-simple goals.

A more complicated example is the `omega` tactic included in the Coq standard distribution. This tactic implements a decision procedure for Presburger arithmetic, as described in more detail in [17, Chapter 16]. It is in practice a very useful tool when working with linear arithmetic; but it also has its limitations.

The main disadvantage of this approach is that all decision procedures thus implemented have no existence in the type theory of Coq. Therefore, nothing can be proved about them—in particular their correctness. They need to produce an explicit proof term which they then pass to the system, and this term is type-checked as any other proof term.

In the case of `omega`, this results in practice in relatively long waiting times unless the goals are trivial, and in very long proof terms even when they are.

In the case of `tauto`, the waiting time is usually not very long if the tactic succeeds, but failure can take a while to be reported. The proof terms produced, though not very long, are not very interesting either: a proof done by hand would in most cases be shorter.

In the next paragraphs it will be discussed how these problems can be at least partially overcome by internalizing decision procedures. Since no external decision procedures besides the ones from the standard distribution of Coq were used in either the FTA-project or C-CoRN, they will not be discussed further.

Reflection

As a motivation for the method of reflection, consider the following situation.

After one has worked with concepts such as continuity for a while, one no longer relies on the basic lemmas to prove that a given function is continuous. For example, let f be the function defined by

$$f(x) = 5x^3 + \frac{e^{\sin x} + \cos e^{-x^2}}{4}.$$

When asked if this function is continuous and why, most people would answer affirmatively, and justify this by saying that it is a direct consequence of its structure. No one would think of saying that it is a sum of two functions, the first of which is continuous because it is a scalar multiplied by a power of the identity function, which is continuous; and the second of which is a scalar multiplied by a sum of two functions which in turn are continuous because… However, if applied to this situation, all methods described so far would produce a proof that would correspond in some sense to this reasoning.

It was therefore suggested by Barendregt that some effort should be made to try to find a method to mimic the human reasoning in a proof assistant, thereby (hopefully) obtaining a more efficient and more general proof technique.

According to him, the idea would be as follows: within the class of all real-valued functions, there are some "basic" functions (constant functions, identity, exponential, trigonometric functions) which are by nature continuous; and there are some internal operations on the class of continuous functions, such as addition, multiplication, division and composition (although the last two may require some extra restrictions). Then, any function that can be built from the basic functions using these operations must be continuous—directly. In Section 4.4 the actual implementation of such a procedure will be discussed.

Abstracting from this idea, one arrives at the method known as *reflection*, originally introduced in [2] and which has now been around for some decades. This method is briefly explained in the following paragraphs; a more detailed description can be found for example in [34].

Let $\mathcal{D}$ be a predicate over a domain S and suppose that there exist an inductive type T and an interpretation $[\![\cdot]\!] : T \to S$ together with a function $f : T \to \{\bot, \top\}$ such that[1]

$$f(x) = \top \text{ iff } \mathcal{D}([\![x]\!]) \text{ holds.}$$

This situation is graphically depicted in the following diagram.

$$\begin{tikzcd} T \arrow[rr, "{[\![\cdot]\!]}"] \arrow[dr, "f"'] & & S \arrow[dl, "\mathcal{D}"] \\ & \{\bot, \top\} & \end{tikzcd}$$

[1] The equivalence is relevant if one wants completeness. However, in many applications, the direct implication is already helpful enough.

In Coq terms, this is formalized as follows: S and T are types and $\mathsf{D} : \mathsf{S} \rightarrow \mathbf{Prop}$.[2] The type T is usually called the *syntactic level*, S is the *semantic level* and $\mathsf{int} : \mathsf{T} \rightarrow \mathsf{S}$ is the interpretation function. The function $\mathsf{f} : \mathsf{T} \rightarrow \mathbf{Prop}$ is defined inductively, and the following must be proved as a lemma.

Lemma reflection_lemma : $\forall_{\mathsf{x}:\mathsf{T}}.(\mathsf{f}\ \mathsf{x}) \rightarrow (\mathsf{D}\ (\mathsf{int}\ \mathsf{x}))$.

Suppose now that a situation arises where the goal is $(\mathsf{D}\ \mathsf{t})$, and t is convertible with $(\mathsf{int}\ \mathsf{x})$ for some x. Then the goal can be simply proved by the term

$$\mathsf{reflection\_lemma}\ \mathsf{x}\ \mathsf{I}$$

where I is the canonical term of type $\top$. There is a trade-off here between space and time: this term is evidently compact (in general the length of x will be linear in the length of t), but type-checking it requires verifying that $(\mathsf{f}\ \mathsf{x}) \rightarrow_{\delta\beta\iota} \top$.

The only thing missing is a way to tell the system how to find x given a specific t. This cannot, in general, be done directly: the inverse operation to int is usually not a type-theoretic function. This happens because S will typically have no inductive structure (otherwise T would not be needed), so one cannot reason by cases over it.

However, using the tactic language of Coq, a small program can usually be written that, given t, returns such an inverse x when it exists (and possibly produces frightening error messages when it does not). Then the goal can be proved, as said above, by the term (reflection_lemma x I). (Notice that, unless t is previously replaced with $(\mathsf{int}\ \mathsf{x})$ using e.g. `change`, the parameter x must be given explicitly.)

The `ring` tactic is a good example of reflection at work. This tactic, described in detail in [17, Chapter 18], simplifies algebraic expressions over an arbitrary ring R in a syntactic way, making use of the ring equations.

In this case, the syntactic level T is the set of polynomial expressions over a ring generated from a countable set of variables. These variables will be assigned the actual atomic terms—those that cannot be decomposed in terms of ring operations, e.g. a or $f(x)$—in concrete expressions.

Instead of directly defining the function f, one first implements a normalization procedure[3] $\mathcal{N} : S \rightarrow S$ and proves that it satisfies the property

[2] In fact, the type of D can also be $\mathsf{S} \rightarrow \mathbf{Set}$ or $\mathsf{S} \rightarrow \mathbf{Type}$, and the remainder of the discussion remains valid without change.

[3] In the case of boolean rings, this normalization procedure coincides with reduction to the disjunctive normal form.

$\forall_{\mathsf{x}:\mathsf{R}}.[\![x]\!] = [\![\mathcal{N}(x)]\!]$. The correctness lemma now reads as follows.

$$\forall_{x,y:T}.\mathcal{N}(x) = \mathcal{N}(y) \to x = y$$

Thus, the function f is a (syntactic) check of the equality of the normal forms.

Actually, the `ring` tactic is more powerful, because it allows not only to decide equalities but also to simplify expressions in the goal making use of the normalization function. The tactic behaves as follows: the user calls `ring` with any number of terms as arguments. Each term t is then represented as the interpretation of a specific polynomial, which is normalized; this normalized form is translated back into the ring, and the resulting expression t' is substituted for t everywhere in the goal.

Further on a similar tactic which works in the setting of the FTA-library, `Rational`, will be described together with more sophisticated examples of reflection. Among these will be an implementation of a tactic capable of solving the goal introduced as motivation at the beginning of this section.

4.2 Equational reasoning in the FTA-library

While formalizing algebra, most of the proofs one has to do deal with equality. The widely used fact that almost all functions and relations are extensional translates in practice in the ability to replace in the goal any element $\mathsf{a} : \mathsf{S}$, where $\mathsf{S} : \mathsf{CSetoid}$, with another element $\mathsf{b} : \mathsf{S}$, generating the subgoal $(\mathsf{a} = \mathsf{b})$.

However, many of the goals thus generated are frustratingly trivial. Most fall into one of three categories:

- two structurally similar expressions that differ only in a subterm which is assumed to be equal; for example:

$$\frac{\mathsf{a} = \mathsf{b}}{(\mathsf{x}+1) \times \mathsf{a} = (\mathsf{x}+1) \times \mathsf{b}}$$

- two expressions which can be proved equal directly from group, ring or field axioms; for example:

$$\frac{}{(2 \times \mathsf{x}) + (0 \times \mathsf{y}) = (\mathsf{x}+\mathsf{x})}$$

- two expressions that combine both of the previous; for example:

$$\frac{\mathsf{x} = \mathsf{a}}{(2 \times \mathsf{x}) + (0 \times \mathsf{y}) = (\mathsf{x}+\mathsf{a})}$$

As it turns out, the goals in the first category can usually be directly and quickly solved by simple tactics based on `auto`.

The goals of the second group, however, are much more difficult to prove using simple search techniques. The problem here is the absence of any relationship between the expressions and the proofs of their equality, unlike in the previous case. In this situation, the simple heuristics used by `auto` are just not powerful enough, in general, to find the possibly complicated corresponding proof terms.

As for the third group, the situation is even worse and it will be seen later that cooperation between tactics of different families is needed to deal with them in a satisfactory way.

Simple search: `Algebra`

At the time of developing the Algebraic Hierarchy, all lemmas stating equalities which involve the algebraic operations were added to a hints database called `algebra`.[4] Then, `Algebra` was defined as an abbreviation of `auto with algebra`.

The behaviour of `Algebra` is simple to describe: it chooses from the `algebra` database the lemmas that apply to the current goal—typically only one or two—and eventually reduces it to directly provable equalities like $(\mathsf{x} = \mathsf{x})$ or hypotheses in the context.

The strong point of `Algebra` is its ability to look at the context. Further ahead another family of tactics will be presented that can solve a much wider class of goals in a much more efficient way, but which lacks this ability.

The possibility of dynamically extending the hints databases should also not be underestimated. The `Algebra` tactic is defined at the same time as the CSetoid structure, but this does not prevent it to be enlarged and later used to prove equalities among real numbers which use properties of these that cannot even be stated outside real number structures. For interactive development this is a *very* useful feature, the importance of which can hardly be overstated; this point will be re-stressed further ahead.

A clear illustration of this point can be obtained by considering the extension of the Algebraic Hierarchy to include partial functions. In the discussion following Definition 3.2.2, one of the key lemmas proved (Lemma 3.2.3) stated extensionality of partial functions. This lemma can be added to the `algebra` database immediately after it is proved; from that moment onwards, `Algebra`

[4]The lemmas were in fact split among different databases in order to avoid repeated and unnecessary application of the symmetry axiom for equality, which can always be applied and has a very low cost. For the purpose of this discussion, however, that is not a relevant point and will be ignored in the presentation.

will know how to use it to prove equalities between expressions involving partial functions.

Reflection: `Rational`

Recall the three classes of equational problems described above on page 63. An enhanced version of reflection will now be shown to solve most of those in the *second* category.

In the original FTA-library, reflection was successfully implemented in the `Rational` tactic to solve any equality between two expressions on a given $\mathsf{F} : \mathsf{CField}$ that could be proved by equational reasoning only using the field axioms. This implementation is described in some detail in [34].

The construction here differs from the abstract method of reflection presented earlier in two ways. First, the FTA-project was developed using Coq version 6.3.1, where the tactic language wasn't available; for this reason the partial inverse to int had to be programmed directly in ML.

Secondly, because of the possible presence of proof terms (due to the partiality of division, see Section 3.1), a tactic based directly on reflection would require the type of syntactic expressions and the interpretation function to be defined in parallel. Although this is in theory possible (it is a very simple example of an inductive-recursive definition, where an inductive type is defined which depends on a function having the type itself as domain), it is not allowed in Coq. Therefore, a generalization of reflection, called *partial reflection*, was implemented.

Partial reflection generalizes reflection by allowing the interpretation function int itself to be partial—or even be a non-functional relation. This allows undefined expressions (such as $\frac{1}{0}$) to be written at the syntactical level, as long as they are not interpretable in the semantic domain. A more detailed explanation of partial reflection can be found in [34].

The mechanism of `Rational` is very similar to that of the `ring` tactic previously described when goals of the form $(\mathsf{t} = \mathsf{t}')$ are considered. The syntactic domain E is the set of rational functions over an abstract field; a countable set of variables is used for the atomic subterms occurring in the goal. The definition of E is as follows[5].

$$E ::= \mathbb{V} \mid \mathbb{Z} \mid E + E \mid E \times E \mid E/E$$

Here $\mathbb{V}$ is a set of variables indexed on $\mathbb{N}$ (i.e., if $n : \mathbb{N}$ then $v_n : E$).

On this syntactic type a normalization function N is defined. Normal forms are best described in three steps.

[5]Informal mathematical notation, rather than the actual Coq representation, is used for E to make the presentation more readable.

1. A monomial is in normal form if it is of the form $v_{n_1} \times \ldots \times v_{n_k} \times i$ with the indices of the variables in non-decreasing order.

2. A polynomial is in normal form if it is of the form $e_1 + \ldots + e_k + i$ with each e_i a monomial in normal form and such that the e_i's appear in lexicographical order with no expression occurring twice.

3. An expression is in normal form if it is of the form e/f, where e and f are polynomials in normal form.

Intuitively, monomials are lists of variables (with $\times$ corresponding to the `cons` constructor and the integer coefficient to the empty list) and polynomials are lists of monomials (with $+$ as `cons` and again integers as the empty list). A normal form is then a (formal) quotient of two sorted lists of monomials without repetitions, the monomials themselves being again sorted lists of variables.

For example, the normal form of $1/(v_0 - v_1) + 1/(v_0 + v_1)$ is

$$\frac{v_0 \times 2 + 0}{v_0 \times v_0 \times 1 + v_1 \times v_1 \times (-1) + 0}.$$

Unfortunately, these normal forms are not unique, since common factors may appear in the two polynomials and eliminating these common factors is known to be a difficult problem. This is not a major drawback, though: the normal form of any expression representing 0 *is* of the form $\mathsf{0/f}$, so given a goal of the form $(\mathsf{t} = \mathsf{t}')$ the tactic first applies a lemma to reduce it to $(\mathsf{t} - \mathsf{t}') = \mathsf{0}$ and then normalizes the syntactic representation of $(\mathsf{t} - \mathsf{t}')$, checking whether this is zero.

The behaviour of the tactic, from the situation where the goal to be proved is $(\mathsf{t} - \mathsf{t}') = \mathsf{0}$, can be represented schematically as follows.

$$\begin{tikzcd} e \arrow[r, "\mathsf{N}"] \arrow[d, "\mathsf{int}"'] & f \arrow[d, "\mathsf{int}"] \\ \mathsf{t} - \mathsf{t}' \arrow[r, "="] & \mathsf{0} \end{tikzcd}$$

Practice shows that `Rational` is an indispensable tool. Thanks to it, arbitrarily complex equalities can be concisely proved. Also, the proof terms produced by `Rational` are roughly linear in the size of the input, in contrast with the potentially exponential ones produced by `Algebra`.

Upgrading `Rational` to work within the extended Algebraic Hierarchy, however, is far less trivial than the corresponding procedure for `Algebra`. First, the inductive domain must be extended with a constructor for partial functions. This is done by adding a constructor providing a countable set of operators to the syntactic type; these operators will be interpreted as the partial functions occurring in any goal to which the tactic is applied. The type E now becomes the following.

$$E ::= \mathbb{V} \mid \mathbb{F}(E) \mid \mathbb{Z} \mid E + E \mid E \times E \mid E/E$$

Next, new rules have to be added to the rewriting system to treat expressions built using this constructor. The tricky part is to define the ordering on monomials, since these may now include expressions built from functional application. This is done by defining an order on the whole set of syntactic expressions. Variables come first, followed by integers, then by sums, products, division and partial function application. Within each category expressions are again ordered using either the order on the set of variables or integers or (recursively) the ordering on expressions.

Unfortunately, the result is not as satisfactory as the original `Rational`, due to the non-uniqueness of normal forms mentioned above. This is because the tactic now needs to decide whether arguments of a partial function are equal or not, and this is done by comparing their normal forms.

The practical consequence is that `Rational`, when extended with partial functions, will not be able to solve goals such as $f(x) = f(\frac{x}{2} + \frac{x}{2})$. Assuming f is represented by f_0 and x by v_0, the normalized syntactic expression for $f(x) - f(\frac{x}{2} + \frac{x}{2})$ will look like

$$\frac{f_0\left(\frac{v_0 \cdot 1 + 0}{1}\right) \cdot 1 + f_0\left(\frac{v_0 \cdot 4 + 0}{4}\right) \cdot (-1) + 0}{1};$$

the 4's in the argument of the second operator coming from the cross multiplications when normalizing the argument of the function.

However, the tactic *will* solve $f(2x) = f(x + x)$. In this case, both $f(2x)$ and $f(x + x)$ will normalize to

$$\frac{f_0\left(\frac{v_0 \cdot 2 + 0}{1}\right) \cdot 1 + 0}{1},$$

which allows the normalization procedure to continue and yield as result $\frac{0}{1}$.

In practice, this limitation turns out to be more of a minor inconvenience than a serious drawback. The present version of `Rational` is this extended one, developed jointly with Wiedijk. It should be pointed out that if no partial functions (apart from division) appear in the goal then the tactic behaves just like the old one.

Still, there is no such thing as a perfect solution for all problems; and there are two big drawbacks to `Rational` even in its original formulation.

In the first place, there is no dynamic knowledge database for reflection tactics. Therefore, `Rational` is a fixed and immutable tactic, in the sense that it cannot be extended with some extra knowledge. For example, it would be nice to teach `Rational` to work with the exponential function using its basic properties, but doing that would require changing it in a fundamental way, and afterwards it would no longer work in an arbitrary field. Symmetrically, it cannot be used in situations where less knowledge is available. One would like to be able to ignore division and use `Rational`'s wonderful capabilities in structures of type `CRing`; but this is not possible, because of the typing of the almighty lemma on which everything hangs. The only solution is again to clone the code, afterwards removing everything that does not make sense for rings. This duplication is present in the FTA-library.

The second problem, which is somewhat related, is that `Rational` has no way of looking at the context. Therefore, even though it can prove very complex equalities which fall in the second of the three categories above discussed, it will fail in even the simplest of those in the first group. And obviously, as a consequence, it cannot tackle any of those from the third class either.

This last point is not a fundamental property of reflection tactics. Both `ring` and `field` look at the context, and in Section 4.4 other tactics will be defined which also do so; it simply wasn't done for `Rational` because it was felt that the benefits that would come from such an improvement did not compensate for the extra work.

The first point can be solved by defining a hierarchy of `Rational`like tactics on top of each other that are usable at all levels of the Algebraic Hierarchy. A description of how this has been done can be found in [23].

4.3 Plugging it all together: the `Step` tactic

In the previous section `Algebra` and `Rational` were introduced and shown to be very useful for proving equalities within the FTA-library. However, in order to do equational reasoning in an efficient and appealing way, a nicer interaction with the system is needed. With this in mind, the `Step` tactic is motivated and introduced in its original version from the FTA-project.

This tactic can be optimized using the tactic language of Coq, and a more satisfactory version (from the point of view of easy usability), which was originally developed within C-CoRN, is presented. The last version is at the basis of the present-day `Step` tactic, implemented at the ML-level in cooperation with the Coq team and to be distributed with future versions of that system.

The need for `Step`

Besides proofs of specific equalities, to formalize a specific chain of equational steps other lemmas are needed.

As an example, consider the proof that the sum of two continuous functions is continuous.

LEMMA 4.3.1 Let f and g be continuous functions on $[a, b]$ with moduli of continuity respectively δ_f and δ_g. Then $\delta(\varepsilon) \stackrel{\text{def}}{=} \min\{\delta_f(\frac{\varepsilon}{2}), \delta_g(\frac{\varepsilon}{2})\}$ defines a modulus of continuity for $f + g$.

PROOF. Let $x, y \in [a, b]$ such that $|y - x| \leq \delta(\varepsilon)$. Then:

$$\begin{aligned}
|(f(y) + g(y)) - (f(x) + g(x))| &= |(f(y) - f(x)) + (g(y) - g(x))| \\
&\leq |f(y) - f(x)| + |g(y) - g(x)| \\
&\leq \frac{\varepsilon}{2}|y - x| + \frac{\varepsilon}{2}|y - x| \\
&= \varepsilon|y - x|
\end{aligned}$$

□

In order to formalize this proof, one needs not only to prove the first and last equalities, but also to use the fact that $\leq$ is an extensional relation in order to be allowed to replace expressions on the left or on the right of the inequality with equal ones. Thus, the original goal is

$$|(f(y) + g(y)) - (f(x) + g(x))| \leq \varepsilon|y - x|.$$

To this one must apply extensionality of $\leq$ on the left and prove the first equality above, reducing the goal to

$$|(f(y) - f(x)) + (g(y) - g(x))| \leq \varepsilon|y - x|.$$

Next, one applies extensionality of $\leq$ on the right and proves the last equality, thus being left with

$$|(f(y) - f(x)) + (g(y) - g(x))| \leq \frac{\varepsilon}{2}|y - x| + \frac{\varepsilon}{2}|y - x|,$$

to which one can apply transitivity of $\leq$ to reduce to the two subgoals

$$|(f(y) - f(x)) + (g(y) - g(x))| \leq |f(y) - f(x)| + |g(y) - g(x)|$$

and

$$|f(y) - f(x)| + |g(y) - g(x)| \leq \frac{\varepsilon}{2}|y - x| + \frac{\varepsilon}{2}|y - x|,$$

which can then be proved.

This proof can be translated into Coq in a straightforward way. The beginning of the resulting proof script[6] is shown in Figure 4.2. Not only does this look very confusing, as it mixes applications of extensionality lemmas with calls to `Algebra` and `Rational`, but it also requires the user to know the names of the lemmas which state extensionality of $\leq$.

Goal: $|(f(y)+g(y))-(f(x)+g(x))| \leq \varepsilon|y-x|$.

`apply leEq_wdl with` $|(f(y)-f(x))+(g(y)-g(x))|$.
`Rational.`
`apply leEq_wdr with` $\frac{\varepsilon}{2}|y-x|+\frac{\varepsilon}{2}|y-x|$.
`Rational.`

Figure 4.2: Proof script for the proof of Lemma 4.3.1

The `Step` family of tactics was originally designed to address these issues. For each relation in the Algebraic Hierarchy (apartness, equality, less than, less or equal than, less than in absolute value) two tactics `Step_`*`rel`*`_lft` and `Step_`*`rel`*`_rht` were defined; here *`rel`* is the standard name for each relation (as defined in the FTA-library, e.g., `ap` for apartness and `leEq` for less or equal) and `lft` or `rht` indicates on which side of the relation the term is to be substituted. For equality these names are shortened to `Step_lft` (or `Step`) and `Step_rht` (or `Stepr`).

All of these tactics work in a similar way: they apply the relevant extensionality lemma and try to prove the generated equality with `Algebra`. As a consequence, a sequence of (in)equalities in an informal proof is translated into a sequence of calls to `Step` in the proof script.

If the equality is to be solved using `Rational` instead of `Algebra`, variants of these tactics named `Step_Rat_`*`rel`*`_lft` and `Step_Rat_`*`rel`*`_rht` are available which work in an analogous way.

The script in the previous example could now be much more simply written down as shown in Figure 4.3.

`Step_Rat_leEq_lft` $|(f(y)-f(x))+(g(y)-g(x))|$.
`Step_Rat_leEq_rht` $\frac{\varepsilon}{2}|y-x|+\frac{\varepsilon}{2}|y-x|$.

Figure 4.3: Proof script for the proof of Lemma 4.3.1, using `Step`

[6]Since the interest of this script lies on the actual tactics being used, the intervening Coq terms are presented in informal mathematical notation.

Notice that the uniformity in names is the greatest advantage of these tactics, allowing the user to focus more on the proof and less on memorizing (or looking for) names of technical lemmas.

The next Step

As the FTA-project evolved into C-CoRN, it became clear that this kind of tactics could be written in a more uniform way using the tactic language of Coq. The previous definition of `Step` is very unaesthetic, as for every new relation four different tactics have to be written, all of which differ only in minor details.

Early in the development of C-CoRN it became apparent that a better solution to the same problem could be found with the tools at hand. The idea is very simple: since for every relation and each argument (left or right) there is only one lemma available, and this is only unifiable with goals involving that particular relation, the `first` tactical[7] can be used to merge all tactics in just two: `Step_lft` (or simply `Step`) and `Step_rht` (or `Stepr`), plus the corresponding versions using `Rational`.

The definition of this new version of `Step` reads as follows.

```
Ltac Step_lft x := first
  [ apply eq_transitive_unfolded with x
  | apply ap_well_def_lft_unfolded with x
  ...
  | apply leEq_wdl with x];
  [idtac | Algebra].
```

The last line has the following meaning: `first` will leave two goals; the first should be left alone, while `Algebra` will be applied to the second one.

The names of these tactics were later changed to `AStepl`, `AStepr`, `RStepl` and `RStepr` to reflect their common identity: their name is `Step`, to which the `A` for `Algebra` or the `R` for `Rational` is prefixed and the `l` for left or the `r` for right is appended.

The nice feature of these tactics is that they can be defined at the beginning of the formalization[8], even if the relevant lemmas do not exist. Thus, their definitions can be kept apart from the development, and whenever (if ever) a new basic relation is added to the Algebraic Hierarchy the definition of the tactic can be changed independently without any difference in behaviour in the old scripts.

[7] This tactical chooses the first lemma of a list that can be applied to the goal.

[8] This was true until Coq version 7.4.

The final version

The tactics of the `Step` family can all be viewed as implementing a three parameter algorithm; this algorithm takes as arguments a Coq-term `c`, a tactic `tactic` and a boolean argument `left?` and acts on a goal of the form $(\mathsf{R}\ \mathsf{a}\ \mathsf{b})$, where a and b have the same type T and $\mathsf{R} : (\mathsf{T} \to \mathsf{T} \to \mathsf{s})$, with s a Coq-sort (**Set**, **Prop** or **Type**). Assuming a list of extensionality lemmas is stored somewhere, together with the indication, for each lemma, of the relation and the argument whose extensionality they assert, the algorithm acts as follows.

1. If `left?`, apply the left extensionality lemma for R to replace the goal by $(\mathsf{R}\ \mathsf{c}\ \mathsf{b})$; otherwise apply the right extensionality lemma to obtain $(\mathsf{R}\ \mathsf{a}\ \mathsf{c})$.

2. Prove the generated subgoal $\mathsf{b} = \mathsf{c}$ or $\mathsf{a} = \mathsf{c}$ using `tactic`.

There is absolutely no reason to restrict the equality to setoid equality; this algorithm would work equally well if Leibniz equality was being used—or any other binary relation. Likewise, there is no reason to consider only `Algebra` and `Rational` as usable tactics.

If these considerations are taken into account, it becomes clear that `Step` can be a very powerful tactic indeed, and applicable in a context much more general than that of C-CoRN. With this in mind, the suggestion was made by the Coq-team to write this more general tactic directly in ML and include it in the standard distribution. The actual ML code of the tactic is due to Herbelin.

The behaviour of the tactic depends on two lists of lemmas, one for left extensionality and the other for right extensionality. These lemmas have a fixed form: if R is a binary relation on a type T and eq is an equality[9] on T, then the lemma for left extensionality will have type

$$\forall_{\mathsf{x},\mathsf{y},\mathsf{z}:\mathsf{T}}.(\mathsf{R}\ \mathsf{z}\ \mathsf{y}) \to (\mathsf{eq}\ \mathsf{x}\ \mathsf{z}) \to (\mathsf{R}\ \mathsf{x}\ \mathsf{y}),$$

and similarly for right extensionality.

Lemmas are added to the database using the `Declare Step` tactic:

```
Declare Left Step lemma.
```

will add lemma as a left extensionality lemma, and similarly for the right extensionality. This lemma will then be tried when the user types in `stepl`

[9] From the point of view of the tactic, it can be any binary relation on T; but the motivation comes from equality, as explained earlier.

t tactic, where t is the term to be replaced in the goal and tactic is the tactic to be used to prove the equality.

Using these tactic the old C-CoRN tactics can be redefined; for example, AStepl is now defined as follows.

```
Ltac AStepl x := stepl x Algebra.
```

Using this approach, the new extensionality lemmas have to be added to the database using Declare Step at the time that the relations they speak about are defined, i.e., those for # and = after the definition of CSetoid, and those for <, ≤ and AbsSmall after the definition of COrdField. And of course, Step can also be used in contexts not involving setoids; some experiments within number theory (using Leibniz equality) have been made within C-CoRN.

Cooperation is the key

In the last paragraphs it was shown how cooperation between different tactics could work to solve goals on which no individual tactic could make progress. This issue will now be more thoroughly explored, focusing on the central issue of equational reasoning in C-CoRN.

On page 63 three classes of problems were presented. In Section 4.2 it was shown how Algebra could deal with those of the first group and Rational with those in the second group. It is now time to show how these two tactics can cooperate to successfully tackle the problems of the third kind.

This class of problems is characterized by its requiring both context-dependent information and heavy equational reasoning. While Rational can successfully deal with the second problem, it cannot take any context information into account; and Algebra does this naturally, but it only solves simple goals.

Some sort of cooperation between both tactics, then, seems to be desirable. In fact, this turns out not to be such a difficult task. Let the following example, which was presented already on page 63, be considered:

$$\frac{\mathsf{x} = \mathsf{a}}{(2 \times \mathsf{x}) + (0 \times \mathsf{y}) = (\mathsf{x} + \mathsf{a})}$$

The trick here is to separate the reasoning in two steps—equational steps on one side, congruence and extensionality properties on the other. This can be done applying transitivity of equality with the term $(\mathsf{x} + \mathsf{x})$. Even though Algebra cannot solve the original goal, it *can* solve the new subgoal

$(\mathsf{x}+\mathsf{x}) = (\mathsf{x}+\mathsf{a})$, since this only depends on very basic congruence and reflexivity properties. Furthermore, this proof is structural, which means that it is found very quickly and is not very long.

The remaining goal, though, can be directly solved using `Rational`, as it no longer depends on the context. Thus, the following proof script solves the original problem.

```
AStepr x+x.
Rational.
```

Notice that this not simply a clever trick. This method also embodies the typical presentation of equational reasoning in mathematics books: for clarity's sake, authors tend to separate algebraic steps where only the field equations are used from those where specific properties of concrete functions need to be applied.

This same method allows the user to solve more complicated goals taking advantage of the extensibility of the `algebra` hints database. In the current version of C-CoRN, this contains all sort of information over basic trigonometric identities; the following example shows how powerful this tactics can be.

Suppose that at some point the equality $\sin(2x) = (\cos(x)+\sin(x))^2 - 1$ needs to be established. This can be shown to hold via the following chain of equations:

$$\begin{aligned}
\sin(2x) &= 2\sin(x)\cos(x) \\
&= 2\sin(x)\cos(x) + 1 - 1 \\
&= 2\sin(x)\cos(x) + (\cos^2(x) + \sin^2(x)) - 1 \\
&= (\cos(x)+\sin(x))^2 - 1
\end{aligned}$$

As it turns out, this level of detail is already enough for the machinery of C-CoRN; alternate steps of `Algebra` and `Rational` can prove the successive equalities: the first and the third rely on lemmas that `Algebra` knows about, while the second and the fourth are purely equational reasoning. Figure 4.4 shows how this piece of informal mathematics can be mimicked quite closely in a valid proof script.

4.4 The C-CoRN tactics

In the previous two sections it was shown how the various tactics from the FTA-library made equational reasoning easy.

Goal: $\forall_{\mathsf{x}:\mathbb{R}}.(\mathsf{Sin}\ 2\times\mathsf{x}) = ((\mathsf{Cos}\ \mathsf{x}) + (\mathsf{Sin}\ \mathsf{x}))^2 - 1$

Proof script:

```
intro x.
AStepl 2 × (Sin x) × (Cos x).
RStepl (2 × (Sin x) × (Cos x) + 1) − 1.
AStepl (2 × (Sin x) × (Cos x) + ((Cos x)^2 + (Sin x)^2)) − 1.
Rational.
Qed.
```

Figure 4.4: `Step` in action

However, equational reasoning is not everything in the domain of Real Analysis. Among the common problems encountered during the formalization of e.g. differential calculus are: proving that a given function f is defined in a given interval I; that it is continuous and differentiable in that same interval; and computing its derivative f' on I, or proving that this function coincides with another function g. The techniques earlier explained can be used to define tactics which will tackle these problems more or less successfully. Case by case, it will also be discussed which is the best way to proceed and why.

Domains of partial functions

Partiality of functions often generates goals which are tedious but straightforward to prove. Automating these was an early step in the development of C-CoRN.

The operations on any algebraic structure can be directly extended to the set of partial functions on that structure by simply defining them pointwise and computing the domain where the resulting expression is defined. In most cases this is simply the intersection of the domains of the original functions. If f and g are partial functions with domains respectively A and B, then $f+g$, $f-g$ and $f\times g$ all have domain $A\cap B$, while $-f$ has domain A. Only in the cases of division and composition do other conditions appear: the domain of f/g is $A\cap\{x\in B \mid g(x)\neq 0\}$, that of $g\circ f$ is $\{x\in A \mid f(x)\in B\}$; but since Bishop-style constructive mathematics avoids the use of division, as discussed in Section 2.4, these last situations do not occur that often.

It seems reasonable to define a tactic that can deal with the most common goals of the form $A\subseteq \mathrm{dom}_f$. This can be achieved quite easily using the `auto with` mechanism. First, a number of lemmas are proved that show

how the domain is (not) affected by the algebraic operations. The first one applies to total functions; the following to the ring operations. In all of them, $\mathsf{R} : \mathbb{R} \to \mathbf{Set}$ and $\mathsf{F}, \mathsf{G} : \mathsf{PartIR}$[10].

Lemma included_IR : (included R $\lambda_{x:\mathbb{R}}.\top$).

Lemma included_FPlus :
(included R (dom F)) $\to$ (included R (dom G)) $\to$
(included R (dom F+G)).

Lemma included_FInv :
(included R (dom F)) $\to$ (included R (dom $-$ F)).

Lemma included_FMinus :
(included R (dom F)) $\to$ (included R (dom G)) $\to$
(included R (dom F$-$G)).

Lemma included_FMult :
(included R (dom F)) $\to$ (included R (dom G)) $\to$
(included R (dom F$\times$G)).

Lemma included_FScalMult :
(included R (dom F)) $\to \forall_{c:\mathbb{R}}$.(included R (dom c $\times$ F)).

Lemma included_FNth :
(included R (dom F)) $\to \forall_{n:\mathbb{N}}$.(included R (dom F^{n})).

These lemmas are then added to a database `included`. Practice shows that `auto with included` can solve the most common goals of this type.

On occasion, composition and/or division do appear; it is then usually helpful to use `assert` or `cut` at the beginning of the proof to add the information that the division or composition being done is defined in the relevant domain to the context. These facts will often be needed repeatedly throughout the proof, so adding them at the beginning will make the proof shorter; also, the following lemmas, which are also part of the `included` database, can then be used to treat these more complicated situations.

Lemma included_FRecip : (included R (dom G)) $\to$
($\forall_{x:\mathbb{R}}$.(R x) $\to \forall_{Hx:(\text{dom } G\ x)}$.(G x Hx) # 0) $\to$
(included R (dom 1/G)).

[10] PartIR is short for (PartFunct $\mathbb{R}$)

Lemma included_FDiv :
  (included R (dom F)) $\rightarrow$ (included R (dom G)) $\rightarrow$
  ($\forall_{x:\mathbb{R}}$.(R x) $\rightarrow \forall_{\mathsf{Hx:(dom\ G\ x)}}$.(G x Hx) $\#$ 0) $\rightarrow$
  (included R (dom F/G)).

Lemma included_FComp : (included R (dom F)) $\rightarrow$
  ($\forall_{x:\mathbb{R}}$.(R x) $\rightarrow \forall_{\mathsf{Hx':(dom\ F\ x)}}$.(dom G (F x Hx'))) $\rightarrow$
  (included R (dom G $\circ$ F)).

Still, in some situations the user would like the system to prove the inclusion as far as possible and just leave the side conditions as subgoals. The `auto` tactic is not able to do this, as it either completely solves the goal it is applied to or leaves it untouched. The tactic language of Coq then comes in handy: since the structure of the function uniquely determines which lemma should be used, either the `first` or the `match` tacticals can be used to reduce the function to its simplest terms. In the first case, the tactic would look like

```
Ltac Lazy_Included := repeat first
  [ apply included_IR
  | apply included_FPlus
  | (...)
  | apply included_FDiv
  | apply included_FComp ].
```

Using `match`, the following can also be defined.

```
Ltac Lazy_Included' := repeat match goal with
  | |-(included _ (dom (λx._))) => apply included_IR
  | |-(included _ (dom FId)) => apply included_IR
  | |-(included _ (dom (_+_))) => apply included_FPlus
    (...)
  | |-(included _ (dom (_/_))) => apply included_FDiv
  | |-(included _ (dom (_∘_))) => apply included_FComp
  end.
```

Even though the latter probably looks more sophisticated and more efficient, both tactics actually behave similarly; the explicit matching that the second tactic does is implicit in the first one, since `first` will try to unify each of the lemmas presented to it with the goal, and that is precisely what is done explicitly in the second case using `match`.

Continuity and differentiability vs. search techniques

A similar approach can also be applied in the case of proving that complicated expressions denote continuous or differentiable functions. However, the resulting tactics' performance is surprisingly poorer. In the case of continuity, the result is still usable for not-too-big expressions; as before, a number of lemmas are first proved about the preservation of continuity through algebraic operations, such as the following. Here, Hab is a compact interval and the context contains information about the continuity of F and G in Hab as well as about the side conditions referred to in Lemma 2.4.8.

Lemma Continuous_I_const : $\forall_{c:\mathbb{R}}$.(Continuous_I Hab λx.c).

Lemma Continuous_I_plus : (Continuous_I Hab F+G).

Lemma Continuous_I_div : (Continuous_I Hab F/G).

These lemmas are then collected in a hints database `continuous` and, as before, this database can be used by `auto` to prove many such goals.

Also in parallel with the previous situation, `first` and `match` can be used to define similar tactics that do not need to solve the goal totally.

Unfortunately, the situation here is less simple than when only definedness is concerned. As was discussed in Section 2.4, there are two notions of continuity, both of which were formalized; and therefore all the previous results have to be established for both of them and added to the database. Furthermore, the contexts typically assume hypotheses stronger than those expected by these lemmas: one might know that f is a continuous function on $\mathbb{R}$, from which it should be straightforward to show that $f+1$ is continuous on $[0,1]$. But this requires also the presence of extra lemmas regarding preservation of continuity through restriction of functions and the simultaneous use of the `included` database.

The effect is that the search procedure at this stage begins to take too long to be useful, and the proof terms too deeply nested to be found by `auto`. And even though `first` and `match` perform somewhat better, a more powerful approach feels needed.

Differentiation is an even worse case. Building a similar hints database including the usual derivation rules is practically useless: when one needs to prove a goal of the form $f' = g$, it is very seldom the case that the derivation rules are directly applicable. Consider the proof of the Law of the Mean (Corollary 2.4.13), where given a function f and parameters $a, b \in \mathbb{R}$, the following auxiliary function is defined:

$$h(x) \stackrel{\text{def}}{=} (x-a)(f(b)-f(a)) - f(x)(b-a).$$

The derivative of h is then given by the expression

$$h'(x) = f(b) - f(a) - f'(x)(b - a) \tag{4.1}$$

but direct application of the derivation rules gives

$$h'(x) = (1-0)(f(b)-f(a))+(x-a)(0-0)-(f'(x)(b-a)+f(x)(0-0)) \tag{4.2}$$

or, with some good will and careful bracketing,

$$h'(x) = (1 - 0)(f(b) - f(a)) - f'(x)(b - a) \tag{4.3}$$

where both products are being regarded as multiplications by a scalar rather than as products of two functions. Still this last form differs from Equation 4.1; hence in order to profit from this kind of automation, the user must type in by hand a larger expression than would be desirable.

Reflection—again

It will now be seen that reflection can also be used to yield more elegant proofs of continuity of functions. Furthermore, the resulting tactic will directly be adaptable, with only the need for an extra layer at the top, to the more complicated situation of differentiable functions.

Because division and composition seldom occur within C-CoRN, the simpler method of pure reflection, which cannot treat these cases, was chosen for implementation instead of the more powerful but also much more difficult partial reflection. It should be noted, still, that *even* these cases can be addressed by the resulting tactic with little extra effort, as will become clear in the next few paragraphs.

To put the situation in terms of the reflection approach, the (semantic) domain S being considered is the type PartIR of real-valued partial functions. Rather than a specific decision problem $\mathcal{D}$, the method can in this case be used to solve a family of decision problems[11] $\{\mathcal{D}_{[a,b]}\}_{[a,b]\subseteq\mathbb{R}}$, where for each a and b the problem $\mathcal{D}_{[a,b]}$ is the predicate $\lambda_{\mathsf{F:PartIR}}.(\mathsf{Continuous\_I\ Hab\ F})$ for some proof term $\mathsf{Hab} : (\mathsf{a} \leq \mathsf{b})$. From this point on, $\mathsf{a}, \mathsf{b} : \mathbb{R}$ are supposed to be fixed parameters.

The syntactic type of continuous functions on the interval $[a, b]$ contains the basic functions $\lambda\mathsf{x.c}$ (constant function with value c) and FId (identity function on the reals). Besides these, extra constructors are available to allow hypotheses from the context to augment the family of known continuous functions. There are four such constructors $\mathsf{hyp\_c}$, $\mathsf{hyp\_d}$, $\mathsf{hyp\_d'}$ and $\mathsf{hyp\_diff}$:

[11]This example uses notation which is explained in Section 5.2.

the first one takes care of functions assumed to be continuous in the interval being considered, the second and third rely on the fact that if g is a derivative of f on an interval then both f and g are continuous on that interval; and the last deals with differentiable functions, which are also continuous wherever differentiable.

The presence of these constructors allows the user to bypass the lack of knowledge about division and composition built in the tactic: by `cut`'ting, `assert`'ing or `generalize`'ing appropriate hypotheses in the context, these cases will also be considered. For example, if one adds to the context the hypothesis (Continuous_I Hab 1/FId), the tactic will also use the fact that the reciprocal function is continuous.

The full inductive definition of the syntactic type cont_function of continuous functions can be seen in Figure 4.5.

The next step is to define the injection cont_to_pfunct going from this type into PartIR. This is totally straightforward to do, and the resulting function is shown in Figure 4.6. Afterwards, it is proved that all the functions in the co-domain of cont_to_pfunct are continuous. This is very simple to do, using `induction` and the lemmas already proved about preservation of continuity through the algebraic operations. Hence the following term is defined.

$$\mathsf{continuous\_cont} : \forall_{\mathsf{Hab}:(\mathsf{a}\leq\mathsf{b})}.\forall_{\mathsf{f}}.\ (\mathsf{Continuous\_I\ Hab\ (cont\_to\_pfunct\ f)}).$$

The last step is to define a partial inverse to cont_fun. Using the tactic language, taking advantage of the `match` tactical, this operator can be defined as a meta function pfunct_to_cont (i.e., not a term in the Calculus of Constructions) as shown in Figure 4.7. The only tricky part is to benefit from the presence of hyp_c and similar constructors; this is done through the call to `match goal with`, which allows also the context to be examined.

Finally, for usability, and since the typical user will not even want to know that all these auxiliary functions exist, the tactic shown in Figure 4.8 is defined to deal with goals of the form (Continuous_I Hab F). The algorithm is as follows.

1. Apply pfunct_to_cont to F; if this fails to return a Coq-term, the tactic fails. Otherwise, let r be the resulting term.

2. Let a and b be the endpoints of the interval (which can be found by `match`'ing with the goal).

3. Replace F in the goal with (cont_to_pfunct a b r) using the fact that these terms are convertible.

4. Apply the lemma continuous_cont to solve the goal.

```
Inductive cont_function : Type :=
| hyp_c      : Π_{Hab:(a≤b)}.Π_{F:PartIR}. (Continuous_I Hab F) → cont_function
| hyp_d      : Π_{Hab:(a<b)}.Π_{F,F':PartIR}. (Derivative_I Hab F F') → cont_function
| hyp_d'     : Π_{Hab:(a<b)}.Π_{F,F':PartIR}. (Derivative_I Hab F F') → cont_function
| hyp_diff   : Π_{Hab:(a<b)}.Π_{F:PartIR}. (Diffble_I Hab F) → cont_function
| cconst     : Π_{c:ℝ}.cont_function
| cid        : cont_function
| cplus      : cont_function → cont_function → cont_function
| cinv       : cont_function → cont_function
| cminus     : cont_function → cont_function → cont_function
| cmult      : cont_function → cont_function → cont_function
| cscalmult  : ℝ → cont_function → cont_function
| cnth       : cont_function → ℕ → cont_function
| cabs       : cont_function → cont_function.
```

Figure 4.5: The syntactic type of continuous functions

```
Fixpoint cont_to_pfunct (r:cont_function) : PartIR := match r with
| hyp_c Hab F H       ⇒ F
| hyp_d Hab F F' H    ⇒ F
| hyp_d' Hab F F' H   ⇒ F'
| hyp_diff Hab F H    ⇒ F
| cconst c            ⇒ λx.c
| cid                 ⇒ FId
| cplus f g           ⇒ (cont_to_pfunct f)+(cont_to_pfunct g)
| cinv f              ⇒ −(cont_to_pfunct f))
| cminus f g          ⇒ (cont_to_pfunct f)−(cont_to_pfunct g)
| cmult f g           ⇒ (cont_to_pfunct f)×(cont_to_pfunct g)
| cscalmult c f       ⇒ c×(cont_to_pfunct f)
| cnth f n            ⇒ (cont_to_pfunct f)^n
| cabs f              ⇒ FAbs (cont_to_pfunct f)
end.
```

Figure 4.6: Translation from cont_function to PartIR

```
Ltac pfunct_to_cont a b f := match constr:f with
| (λx.?X3)      ⇒ constr:(cconst a b X3)
| FId           ⇒ constr:(cid a b)
| ?X3+?X4       ⇒
    let t1:=pfunct_to_cont a b X3 with t2:=pfunct_to_cont a b X4
    in constr:(cplus a b t1 t2)
| −?X3          ⇒ let t1:=pfunct_to_cont a b X3 in constr:(cinv a b t1)
| ?X3−?X4       ⇒
    let t1:=pfunct_to_cont a b X3 with t2:=pfunct_to_cont a b X4
    in constr:(cminus a b t1 t2)
| ?X3×?X4       ⇒
    let t1:=pfunct_to_cont a b X3 with t2:=pfunct_to_cont a b X4
    in constr:(cmult a b t1 t2)
| c×?X4         ⇒ let t:=pfunct_to_cont a b X4 in constr:(cscalmult a b c t)
| ?X3^?X4       ⇒ let t1:=pfunct_to_cont a b X3 in constr:(cnth a b t1 X4)
| (FAbs ?X3)    ⇒ let t1=pfunct_to_cont a b X3 in constr:(cabs a b t1)
| ?X3           ⇒ let t:=constr:X3 in match goal with
     | H:(Continuous_I (a:=a) (b:=b) ?X1 t)|-_       ⇒
           constr:(hyp_c a b X1 t H)
     | H:(Derivative_I (a:=a) (b:=b) ?X1 t ?X4)|-_ ⇒
           constr:(hyp_d a b X1 t X4 H)
     | H:(Derivative_I (a:=a) (b:=b) ?X1 ?X4 t)|-_ ⇒
           constr:(hyp_d' a b X1 X4 t H)
     | H:(Diffble_I (a:=a) (b:=b) ?X1 t)|-_          ⇒
           constr:(hyp_diff a b X1 t H)
     end
end.
```

Figure 4.7: The inverse to cont_to_pfunct

```
Ltac Contin := match goal with
  | |-(Continuous_I (a:=?X1) (b:=?X2) ?X4 ?X3) =>
      let r := pfunct_to_cont X1 X2 X3 in
      (change (Continuous_I X4 (cont_to_pfunct X1 X2 r));
       apply continuous_cont)
  end.
```

Figure 4.8: Tactic script for Contin

Notice that the only step in the tactic that might fail is the first one. If this happens, then the goal is not provable by this specific tactic: it deals with functions that are not syntactically representable in cont_function, and the context does not provide enough information to proceed.

If the first step succeeds, then everything must also go through due to typing conditions and the fact that pfunct_to_cont, where defined, is inverse to cont_to_pfunct. Therefore, the original goal is then completely solved. In no case is the user left with goals involving the auxiliary functions.

The same principle can be applied, although with some extra complexity added now, to situations where derivatives of functions are involved.

When the goal at hand consists simply in showing that a given function is differentiable, a tactic can be written in pretty much the same fashion as the one previously shown. Two constructors have to be omitted in the syntactic type, as continuous functions need by no means to be differentiable (and indeed, some are not) and the absolute value of a differentiable function is also not always differentiable (as the simple case of the identity function clearly exemplifies). Once these two constructors are removed, however, everything runs in pretty much a parallel way and this is thus not such an interesting case to study at this point.

When the goal to be proved is of the form (Derivative_I Hab F G), requiring one to prove that the derivative of F on a given interval with endpoints a and b (and Hab : (a < b)) is G, the situation changes drastically. Whereas tactics based on `auto` or a reflection approach perform equally well when continuity is concerned, the higher complexity of the latter paying off only when the functions involved began to have large expressions, in this situation reflection *must* be used. And a more intricate version of reflection, as it happens, even though it is still total reflection: the same trick as before will be used to make the inability to deal with division and functional composition a minor drawback.

The key thing to notice in this situation is that there is a domain, which can be inductively defined, of functions that not only are differentiable, but whose derivative can actually be computed just from the expression of the function.

Defining such a domain deriv_function is straightforward, as Figure 4.9 shows. The interpretation function deriv_to_pfunct is analogous to the above cont_to_pfunct, and has thus been omitted.

Structurally computing the derivative from the syntactic representation is done by the deriv_deriv function of Figure 4.10. In this function, the second case (dealing with differentiable functions) needs some explanation: being differentiable is defined via an existential quantifier, i.e., saying that "there

```
Inductive deriv_function : Type :=
| hyp       : Π_{Hab:(a<b)}.Π_{f,f′:PartIR}. (Derivative_I Hab f f′) → deriv_function
| hyp'      : Π_{Hab:(a<b)}.Π_{f:PartIR}. (Diffble_I Hab f) → deriv_function
| const     : Π_{c:ℝ}.deriv_function
| id        : deriv_function
| rplus     : deriv_function → deriv_function → deriv_function
| rinv      : deriv_function → deriv_function
| rminus    : deriv_function → deriv_function → deriv_function
| rmult     : deriv_function → deriv_function → deriv_function
| rscalmult : ℝ → deriv_function → deriv_function
| rnth      : deriv_function → ℕ → deriv_function.
```

Figure 4.9: The syntactic type of differentiable functions

```
Fixpoint deriv_deriv (r:deriv_function) : PartIR := match r with
| hyp Hab f f' H ⇒ f'
| hyp' Hab f H   ⇒ PartInt (ProjS1 H)
| const c        ⇒ λx.0
| id             ⇒ λx.1
| rplus f g      ⇒ (deriv_deriv f)+(deriv_deriv g)
| rinv f         ⇒ −(deriv_deriv f)
| rminus f g     ⇒ (deriv_deriv f)−(deriv_deriv g)
| rmult f g      ⇒
     (deriv_to_pfunct f)×(deriv_deriv g)+(deriv_deriv f)×(deriv_to_pfunct g)
| rscalmult c f  ⇒ c×(deriv_deriv f)
| rnth f n       ⇒ match n with
  | O   ⇒ λx.0
  | S p ⇒ λx.(p + 1)×(deriv_deriv f)×(deriv_to_pfunct (rnth f p))
  end
end.
```

Figure 4.10: Computation of derivatives from the syntactic representation

exists a function which is the derivative of…". This should make the use of the ProjS1 constructor clear; then, a map is applied that maps this function (a total function on the interval $[a, b]$) into a partial function.

This function is a type-theoretic function, and it can be proved to be correct in the sense that the following lemma holds.

Lemma deriv_deriv_function : $\forall_{\mathsf{Hab:(a<b)}}.\forall_{\mathsf{f:deriv\_function}}.$
(Derivative_I a b Hab (deriv_to_pfunct f) (deriv_deriv f)).

The partial inverse pfunct_to_deriv is then defined just like in the case of continuity, so it will not be displayed here.

The actual tactic implements the following algorithm.

1. Find a syntactic representation r of F.
2. Compute, syntactically, a derivative F′ of (deriv_to_pfunct r).
3. Using the fact these terms are convertible, replace in the goal F by (deriv_to_pfunct r).
4. Using extensionality of the derivative relation, replace G by F′. This will leave two subgoals.
5. Use the fact that the syntactic computation of the derivative is correct to solve (Derivative_I Hab (deriv_to_pfunct r) (deriv_to_pfunct r′)).

The actual script is shown in Figure 4.11.

```
Ltac Deriv := match goal with
  | |-(Derivative_I (a:=?X1) (b:=?X2) ?X4 ?X3 ?X5) =>
      let r := pfunct_to_deriv X1 X2 X3 in
      (change (Derivative_I X4 (deriv_to_pfunct X1 X2 r) X5);
      apply Derivative_I_wdr with (deriv_deriv X1 X2 r);
      [unfold deriv_deriv, deriv_to_pfunct in |- *
        | apply deriv_deriv_function])
  end.
```

Figure 4.11: Tactic script for Deriv

As was the case with Contin, the only step that may fail is the first. However Deriv always leaves as goal the proof that G is equal to the derivative syntactically computed. This is an extremely non-trivial problem: when doing mathematics, everyone has a tendency to simplify expressions as they appear, as was discussed earlier with the example of the Law of the Mean.

Using this tactic in this proof, the goal which will be left for the user to prove is the equality between the expressions in Equations 4.1 and 4.2.

Now, even though this seems to be an equality that can be proved simply through equational reasoning using the ring axioms, there is a problem: there is no ring structure on PartIR. In fact, there is no algebraic structure at all on this type, since there is an additive identity (the zero function) but no additive inverse in general, since there is no way to add a function to a (strictly) partial function and obtain a total function. Similarly, there is a multiplicative identity (the constant one function) but no multiplicative inverse.

Still, one could in principle unfold the definition of equality between two functions to reduce it to proofs of definedness and pointwise equality to make use of `Rational` and `Included`. Unfortunately, even though this works in the specific case of the Law of the Mean, in general it is still not enough; and worse, in many situations (e.g., in the proof of Taylor's Theorem) these tactics take so long even to fail that just trying them proves a very bad idea indeed.

The reason is that, in the intuitive reasoning that is usually followed when working with derivatives, one constantly makes use of another property that has barely been mentioned so far: the uniqueness of the derivative function. That is, one usually writes f' for the derivative of f regardless of the specific representation of that function, since extensional equality is all one usually cares about. But in the formalization different representations of the same function are not the same object, and explicit use of this lemma is then required.

Therefore, after using this tactic, the user still has to do a proof of equality by hand. Nevertheless, this is felt as a great improvement, as typically using `Included` and `Rational` in an intelligent way and resorting to the uniqueness of the derivative this goal can typically be solved by hand much more quickly than the original one would have been.

As an example, in the proof of the Law of the Mean `Deriv` is called on the following goal, where a and b are real numbers, F is a partial function with derivative F', and HA and HB are proofs that F is defined at a and b, respectively.

$$\begin{array}{l}\mathsf{Derivative\_I\ Hab}\\ \quad (\mathsf{FId}-\lambda \mathsf{x.a})\times\lambda \mathsf{x.((F\ b\ HB)}-\mathsf{(F\ a\ HA))}-\mathsf{F}\times\lambda \mathsf{x.(b}-\mathsf{a)}\\ \quad \lambda \mathsf{x.((F\ b\ HB)}-\mathsf{(F\ a\ HA))}-\mathsf{F'}\times\lambda \mathsf{x.(b\text{-}a)}\end{array}$$

The user is left with the following goal.

```
Feq [a,b]
    ((FId−λx.a)×λx.0+(λx.1−λx.0)×λx.((F b HB)−(F a HA)))−
        F×λx.0+F'×λx.(b−a)
    λx.((F b HB)−(F a HA))−F'×λx.(b-a)
```

The gigantic expression in this goal is the derivative computed by the tactic.

4.5 Comparative analysis

Figure 4.12 summarizes the C-CoRN tactics presented in the previous sections and their applicability.

Intuitively, there are many differences between the two kinds of tactics described in the previous sections, not only at the level of how the algorithms that they implement work, but also as regards the ideas behind them. These differences also create a big difference in the way that they are useful for interactive theorem proving.

Being simple guided search algorithms, all tactics built from `auto` by adding hints to databases share the limitations of these algorithms. In particular, they all have exponential complexity in the worst case scenario (i.e., when they cannot solve the goal), which in the case of `Algebra` has practical consequences: since symmetry of equality and commutativity of addition and multiplication can always be applied, the tactic will always have many paths to explore.

Also, there is no guarantee that the proof terms found by `auto` are at all like anything a human being would write. In the FTA-library several strange examples have been found, including a situation where the final proof term had a proper subterm of the same type[12].

Finally, when the context grows large, even goals that can be proved exactly by one of the hypotheses can take absurdly long to prove. If the user is familiar with how the Coq system works, he can avoid this problem by using a different tactic; but one of the objectives of defining tactics like `Algebra` is precisely to prevent the user from having to do so. And of course, if the goal can be reduced to the type of a hypothesis in the context by application of a lemma in the `algebra` database, then avoiding this problem requires knowing the name of the relevant lemma.

These problems are all avoided by tactics built using reflection. As regards complexity, they usually take some time to build the term in the syntactic type that will allow them to solve the goal; but once that is done the proof term can be directly written down without further ado, as it has a fixed

[12] This was pointed out by Letouzey.

Tactic	Description	Example
`Algebra`	Search tactic for equational reasoning; looks at both the context and `algebra` hints database	Will solve $\dfrac{\mathsf{a}=\mathsf{b}}{(\mathsf{x}+1)\times\mathsf{a}=(\mathsf{x}+1)\times\mathsf{b}}$
`Rational`	Reflection tactic for equational reasoning	Will solve $\dfrac{}{(2\times\mathsf{x})+(0\times\mathsf{y})=(\mathsf{x}+\mathsf{x})}$
`Step`	Allows the user to replace expressions on either side of an (in)equality by equals	From the goal $\mathsf{a}+\mathsf{b}\leq\mathsf{e}$, the tactic $\mathsf{RStepr}\ \mathsf{e}/2+\mathsf{e}/2$ will produce $\mathsf{a}+\mathsf{b}\leq\mathsf{e}/2+\mathsf{e}/2$
`Included`	Search tactic to prove inclusion of a set in the domain of a function	Will prove that the interval $[0,1]$ is included in the domain of $2\times\mathsf{Sine}\times\mathsf{Cosine}$
`Contin`	Reflection tactic to prove continuity of a function; can look at the context	Is able to prove that $(\mathsf{F}+\mathsf{FId})\times(\lambda\mathsf{x}.2)^3-4\times\mathsf{F}$ is continuous from the hypothesis that F is continuous
`Deriv`	Reflection tactic that reduces a proof that a function G is the derivative of F to a proof of equality between G and the computed derivative of F	In a context stating that Cosine is the derivative of Sine, it will reduce the goal "the derivative of Sine^2 is $2\times\mathsf{Sine}\times\mathsf{Cosine}$" to "$2\times\mathsf{Sine}\times\mathsf{Cosine}$ is equal to $2\times\mathsf{Cosine}\times(\mathsf{Sine}^1)$"

Figure 4.12: C-CoRN tactics and their use

form. Afterwards there is some work to be done by the type checker, as was discussed above in Section 4.1, but it is important to notice that this step will be done *only* when the tactic is successful: it is necessary because at some stage a partial inverse is defined which is not a term in the type theory, and so nothing about it can be proved (namely, that it is a partial inverse to some function). But if it is indeed well defined, then it will fail if the tactic cannot completely solve the goal, and it will do so rather quickly.

The proof terms produced by reflection tactics also do not in general resemble anything a human would write down. Nevertheless, they embody human-like reasoning in a different way, representing the fact that something so trivial is being proved that it is not interesting to show the details. This is in direct contrast to proof-terms produced by `auto`, which show all the steps in the reasoning they represent.

Another important feature of proof terms generated via reflection is that they grow linearly with the input. This is a great advantage when comparing, for example, `Algebra` with `Rational`: the presence of symmetry of equality, commutativity and distributivity laws means that direct proofs (those generated by `Algebra`) will typically contain many steps that do not reduce the size of the expressions involved, hence yielding proofs whose size can be exponential in the size of the input. In the case of `Rational`, these steps are only implicitly present when the output proof term is type-checked (in the computation of the normalization function); the proof term itself is linear in the size of the input.

When dealing with the other examples that were presented earlier, however, this argument becomes less powerful. When proving continuity of a function all applicable lemmas either solve the goal or reduce it to proving the continuity of functions with a smaller representation; because of the dependent types that these lemmas have, the resulting proof-term is still super-linear, but not necessarily larger than the one generated by reflection, since that one has linear complexity but with large coefficients. For the tactics dealing with derivatives the situation is pretty much the same; but as was said before, the crucial step is computing derivatives syntactically. Doing this requires setting up the full machinery of reflection, and then defining a tactic becomes such a trivial step that it seems unreasonable not to take it.

In spite of all this, it is most certainly *not* the case that reflection is always the best option. There are several reasons why the clumsy-looking search tactics are sometimes preferable, and very often cooperation between both kinds turns out to be the best solution.

The major disadvantage of reflection is its lack of flexibility. When continuity and differentiation were treated, this was in part compensated for by adding special constructors that allowed hypotheses from the context to be used in building syntactic expressions.

As a complement, a database `derivative` has also been defined, and it contains all lemmas about differentiation and differentiation rules. This can be at any time extended; for example, when transcendental functions were defined, the lemmas about them were also added to this database.

As an example of this combined use of search and reflection tactics, suppose that $\mathsf{H} : (\mathsf{0} < \mathsf{1})$ and the goal to prove is

$$\mathsf{Derivative\_I\ H\ Sine}^2\ 2\times\ \mathsf{Sine}\ \times\ \mathsf{Cosine}.$$

It would be nice to use `Deriv` to solve this goal as far as possible, but this will fail as this tactic knows nothing about trigonometric functions. What one can do is use `assert` to add (Derivative_I H Sine Cosine) to the context. This very simple goal can be automatically proved by `auto with derivative` (not requiring the user to know the name of the relevant lemmas), and `Deriv` will now work on the original goal and leave for the user the much simpler goal of equality between $\mathsf{2} \times \mathsf{Cosine} \times (\mathsf{Sine}^1)$ and $\mathsf{2} \times \mathsf{Sine} \times \mathsf{Cosine}$ in the interval $[0, 1]$.

At this stage, all the machinery is in place for the more detailed description of the formalization which will be presented in the next chapter.

From the beginning automation was perceived as a very important question to be addressed before and throughout the formalization process. In this chapter the two major methods of developing tactics for Coq were examined, their merits and disadvantages presented and discussed, and examples from the FTA-library shown alongside with their extension to C-CoRN and new tactics in this new setting.

In order to fulfill the proposed task of formalizing Bishop-style Real Analysis following [10], these tactics have to be used in an intelligent way, taking advantage of the capabilities of each of them. Cooperation turns out to be the key, as was exemplified in the previous section.

With all the pieces in place, the formalization itself can now be presented from a more technical perspective. This will be done in the next chapter.

Chapter 5

Building a Library of Real Analysis

In the previous two chapters it was shown how partial functions could be added to the Algebraic Hierarchy of the FTA-library and how several tactics were defined to make the task of formalizing Real Analysis easier.

In this chapter the concrete formalization will be discussed, care being taken to explain the design choices. As presentation is concerned, the order of Bishop's book [10, Chapter 2] will be followed; however, in contrast to Chapter 2, the focus will be now on the actual formalization, and its technical aspects will be analyzed. For this reason, actual (albeit pretty printed) Coq code will be shown in several places; this code is correct input except when explicitly stated. Although this code is sometimes complex, displaying it makes it easier to examine the relationship between the informal and formal presentations of the same results, one of the goals of this chapter.

A brief introduction to Coq syntax and the notation herein used is provided in Appendix A. The specific notation and results from the FTA-library, which will be used without comment, have all been introduced in Section 2.2.

The work described here represents a larger amount of Coq code than that originally present in the FTA-library. Therefore, it was decided to abandon that name for the existing library, and thus was born C-CoRN: the Constructive Coq Repository at Nijmegen, the goals and methodology of which were already discussed in Section 2.3.

5.1 Extending the Algebraic Hierarchy

The first step in the formalization of Real Analysis was to extend the Algebraic Hierarchy with the necessary notions. These can be divided into two groups: generic algebraic results, dealing mostly with partial functions, their properties and operations on them; and results about real numbers, including in particular many new lemmas on Cauchy sequences and subsequences, as well as a theory of series of real numbers.

Algebraic Structure of Partial Functions

As was discussed extensively in Chapter 3, it was decided to formalize partial functions using a propositional approach similar to that of Automath [7]. Hence, an n-ary partial function is simply an $(n + 1)$-ary function whose last argument is a proof term. Throughout this chapter, only unary partial functions will be used; furthermore, since only partial functions on the reals are needed, their type ($\mathsf{PartFunct}\ \mathbb{R}$) will be abbreviated to PartIR.

In statements of lemmas, for the sake of generality, the proof arguments of partial functions will usually be universally quantified. Since their type is always clear from the context, it will be omitted in the quantifier.

Definition 3.2.2 showed how the type of partial functions over an arbitrary setoid S was defined. The pros and cons of this definition having already been debated, it is now time to start using it.

The first question which naturally arises is whether the algebraic structure of a setoid propagates to the collection of its partial functions. The answer to this question, as was briefly discussed in Section 4.4, is negative: although any algebraic operation on a setoid can be extended to the collection of partial functions on that setoid simply by pointwise definition, partiality breaks down most *properties* of that operation. For example, in any monoid structure M the constant function with value the unit of the monoid is trivially the unit for the addition of partial functions on that monoid; however this operation can have neither a left nor a right inverse: if f is the everywhere undefined function on M, then $f + g = f = g + f$ for every partial function g on M—and in particular f cannot have an inverse on either side.

There is a crucial question, however, which must be answered before this argument can be made more precise: just what notion of equality is being used to compare two functions? As it turns out, there are at least two different possibilities—and the one which is perhaps the most natural is not the one which is the most useful for the purposes of this work. The reason for this will be explained below.

Another point worth mentioning is that, where the formalization is concerned, there is another, more subtle, issue at hand: as was explained at the end of Section 3.3, the type of partial functions on a setoid is **Type**, and thus partial functions cannot form a setoid altogether simply because of typing constraints (as the carrier of a setoid must have type **Set**). After discussing the possible equalities between partial functions, though, a partial solution to this will turn out to exist.

Any plausible notion of equality of partial functions should satisfy two basic properties:

- it should be an equivalence relation;
- values of two equal functions should coincide at points where both of them are defined.

The most natural definition is the traditional one: f and g are equal iff $f(x)$ is defined precisely when $g(x)$ is, and if that happens then $f(x) = g(x)$.

In practice, though, such a definition has two disadvantages: on the one hand, it generates goals that can be quite hard to prove; on the other hand, it is too strong in many circumstances.

Both problems can be easily understood if one considers the intended use of this equality. Since extensionality is a central concept in constructive mathematics, it is to be expected that the properties of being continuous, differentiable and the like should be extensional. In other words, if f is e.g. a continuous function and g is equal to f, then g should be continuous.

However all these notions are parameterized by intervals, as was mentioned in Section 2.4. Suppose then that f is continuous on an interval I; then it has to be everywhere defined on that interval (see Definition 2.4.6, where this is implicitly assumed though not explicitly stated). Therefore, what is in fact being required of g is that it be defined at least on I. Typically, this will be easier to prove directly (rather than that $f(x)$ defined implies $g(x)$ defined for $x \in I$). And, on I, f is everywhere defined, whence proving [$g(x)$ defined] implies [$f(x)$ defined] is also somewhat redundant.

As for what happens outside I, it is totally irrelevant for the purpose of this specific application! Even the request that f and g coincide wherever they are both defined can be safely dropped.

This argument is not tailored to the notion of continuity: since all properties of functions are relativized to an interval, a similar reasoning can be made for all of them.

Hence, rather than formalizing the standard definition, it is more interesting to consider a parameterized family of equality relations $\{=_I\}_{I \subseteq \mathbb{R}}$. In practice, I will be taken to be an interval, but this is not necessary.

Definition 5.1.1 Let I be a subsetoid of $\mathbb{R}$. Two partial functions f and g on $\mathbb{R}$ are said to be *equal on* I, denoted $f =_I g$, iff the following three conditions hold:

- if $x \in I$ then $f(x)$ is defined;

- if $x \in I$ then $g(x)$ is defined;
- for all $x \in I$, $f(x) = g(x)$.

Even though this definition can be made in general for any setoid S, it was only needed in the field of Real Analysis. Therefore, it was only formalized in this particular case.

Before the formal definition can be presented, though, some thought has to be given to the issue of how sets of real numbers are to be represented.

For reasons similar to those discussed in Chapter 3, it is not useful to work with the subsetoids of Definition 3.1.1; rather, sets of real numbers will be identified throughout this chapter with predicates of type $\mathbb{R} \to \mathbf{Set}$. Thus, a statement like $x \in A \subseteq \mathbb{R}$ will be formalized by means of $\mathsf{x} : \mathbb{R}$ and the existence of a term of type $(\mathsf{A}\ \mathsf{x})$.

Operations on subsets (inclusion, intersection) can then be directly translated as logical operations on the corresponding predicates (implication, conjunction); the improper subset of the real numbers corresponds to the predicate $\lambda_{\mathsf{x}:\mathbb{R}}.\top$.

The greatest advantage of this approach is that it becomes very easy to deal with elements assumed to be in different subsets at the same time. That is, if the real number x is an element of A and B it will be represented by $\mathsf{x} : \mathbb{R}$ (which can be added, multiplied and the like to any other real number, thanks to its type) with extra hypotheses in the context which can be used whenever needed.

With this in mind, the formal version of Definition 5.1.1 should now be clear.

Definition Feq $(\mathsf{I}{:}\mathbb{R} \to \mathbf{Set})$ (F G:PartIR) :=
  (included I (dom F)) $\wedge$ (included I (dom G)) $\wedge$
  $(\forall_{\mathsf{x}:\mathbb{R}}.(\mathsf{I}\ \mathsf{x}) \to \forall_{\mathsf{Hx},\mathsf{Hx}'}.(\mathsf{F}\ \mathsf{x}\ \mathsf{Hx}) = (\mathsf{G}\ \mathsf{x}\ \mathsf{Hx}'))$.

It is worth pointing out that this definition could be written in an equivalent way using existential quantification:

$$f =_I g \;\leftrightarrow\; \forall_{x \in I}.\exists_{H_x, H'_x}.f(x, H_x) = g(x, H'_x)$$

However, the latter characterization is in practice much more difficult to use. Using Definition 5.1.1, to prove $f =_I g$ one has to prove two inclusions, which can often be dealt with using the automation tactics described in Section 4.4, and an equality of setoid elements, which can also often be dealt with using the `Step` tactic as explained in Section 4.3; and the proof terms are universally quantified, meaning they will be perceived as irrelevant (which they are, since values partial functions do not depend on their logical argument) and have short names (since they are variables).

Proving equality via an existential statement, on the other hand, requires explicitly giving proof terms as arguments to the system; and these will be fully displayed throughout the whole proof, making everything much harder to read and to interact with. Also, there is no obvious way to automate any but the last part of the proof.

It should be pointed out, however, that the equivalence between these two definitions highly depends on the assumption of proof irrelevance. Without it, Definition 5.1.1 is not only stronger, but also more intuitive. Also, it is interesting to note that, with this definition, the relation $f =_I f$ holds iff proof irrelevance holds for f within I.

For any subset I of the real numbers, the relation $=_I$ is an equivalence relation. Furthermore, it is a congruence with respect to the (pointwise defined) algebraic operations: if $f_1 = f_2$ and $g_1 = g_2$, then $f_1 + g_1 =_I f_2 + g_2$, $-f_1 =_I -f_2$, $f_1 \times g_1 =_I f_2 \times g_2$, $1/f_1 =_I 1/f_2$ and $f_1/g_1 =_I f_2/g_2$. All these properties are proved in the C-CoRN library.

There is another interesting insight provided by this definition. To define $=_I$ one looks only at functions which are always defined on I; this suggests that a parallel Algebraic Hierarchy of partial functions could be built parameterized by a subsetoid. This turns out to work quite nicely: given a subsetoid I of a setoid S, the collection of total setoid functions from I to S also forms a setoid, with apartness given by

$$f \#_I g \stackrel{\text{def}}{=} \exists_{x \in I}.f(x) \# g(x).$$

On the other hand, these functions can be bijectively mapped to partial functions on S, as described in the proof of Theorem 3.3.1. It then turns out that $f =_I g$ iff $\neg f \#_I g$, identifying functions with their image through this bijection.

Unfortunately, though as much algebraic structure exists on these structures as on the original setoid S, Coq's coercion mechanism is not strong enough to allow this algebraic structure to be used in practice. If one uses algebraic notation to write e.g. $\mathsf{F} + \mathsf{G}$ for F, G : ($\mathsf{CSetoid\_fun}$ $\{\mathsf{S} \mid \mathsf{I}\}$ S), then a term like $((\mathsf{F} + \mathsf{G})\ \mathsf{a}\ \mathsf{H_a})$ will fail to type check because the system isn't able to coerce $(\mathsf{F} + \mathsf{G})$ to an abstraction.

For this reason, no such notation is ever used throughout C-CoRN. Instead, algebraic operations on partial functions are defined pointwise and denoted by tokens differing from their setoid equivalents in the use of curly braces. Thus, $\mathsf{F}\{+\}\mathsf{G}$ denotes $f+g$, and so on. Throughout this presentation, however, this distinction will be ignored and notations like $\mathsf{F} + \mathsf{G}$ and $\mathsf{F} \times \mathsf{G}$

will be used uniformly, as it is always clear from the context whether the arguments to the operations are elements of an algebraic structure or partial functions. Notice that, unlike for numbers, division of two partial functions is always defined, so that F/G and $\mathsf{1}/\mathsf{F}$ are partial functions without the need for any extra proof term.

Composition of two partial functions F and G is defined by the Coq term (Fcomp F G), hereafter denoted[1] $\mathsf{G} \circ \mathsf{F}$. Given $\mathsf{c} : \mathsf{S}$, the constant function (Fconst c) and the pointwise multiplication (Fscalmult c F) are defined, as well as the identity function FId. The first two will be denoted in this text by $\lambda \mathsf{x}.\mathsf{c}$ and $\mathsf{c} \times \mathsf{F}$, respectively; in the latter, the types of c and F make it clear that the operator is not multiplication of ring elements or of partial functions.

Sequences and Series of Real Numbers

The original FTA-library included very little material on sequences. Present were several (equivalent) definitions of Cauchy sequences and elementary properties of limits. However, for the purpose of formalizing Real Analysis many more results were needed, in particular to pave the way for the definition of functions as power series.

There were two different directions in which this section of the library was extended.

First, there was a concern for completeness. The choice of lemmas formalized in the FTA-project was very strongly geared towards the proof of the Fundamental Theorem of Algebra, and results which weren't relevant for that end were more often than not left unstated. In C-CoRN, care was taken to complete the set of existing results with similar or related ones in a more systematic way. The new results include (strong) extensionality of the limit function; generalization of the facts proved regarding inequalities to apartness, e.g.

$$\lim_{n\to\infty} x_n \# \lim_{n\to\infty} y_n \to \exists_{N\in\mathbb{N}}.x_N \# y_N;$$

and more results regarding arithmetic operations, such as

$$\lim_{n\to\infty} x_n^{-1} = \left(\lim_{n\to\infty} x_n\right)^{-1}$$

under suitable conditions.

In the second place, the remaining results in Bishop's book—many of them only implicitly assumed—were also formalized. The emphasis here lies

[1] The order of the arguments is inverted, following common mathematical practice.

on results about subsequences: even though he never mentions it, Bishop often uses results about Cauchy sequences whose hypotheses place restrictions on all terms of the sequences in situations where these only hold from some point onwards. Of course, this is known to mathematicians to be no problem; but using it in the formalization required some extra lemmas to be added to the library. As an example, consider the following already existing result.

Lemma leEq_seq_so_leEq_Lim : $\forall_{\mathsf{seq}:(\mathsf{CauchySeq}\ \mathbb{R})}.\forall_{\mathsf{y}:\mathbb{R}}.$
$(\forall_{\mathsf{i}:\mathbb{N}}.\mathsf{y} \leq (\mathsf{seq\ i})) \to (\mathsf{y} \leq (\mathsf{Lim\ seq}))$.

The following new version is much more useful.

Lemma str_leEq_seq_so_leEq_Lim : $\forall_{\mathsf{seq}:(\mathsf{CauchySeq}\ \mathbb{R})}.\forall_{\mathsf{y}:\mathbb{R}}.$
$(\exists_{\mathsf{N}:\mathbb{N}}.\forall_{\mathsf{i}:\mathbb{N}}.(\mathsf{N} \leq \mathsf{i}) \to \mathsf{y} \leq (\mathsf{seq\ i})) \to (\mathsf{y} \leq (\mathsf{Lim\ seq}))$.

A notion of *almost everywhere equal* is also introduced (two sequences are almost everywhere equal iff their terms coincide from some point on) and it is proved that two almost everywhere equal sequences will both converge if one of them does, and in this case to the same limit.

Then, Bishop's theory of series was formalized. To begin with, for every sequence the sequence of its partial sums is defined. This is formalized as an operator on sequences.

Definition seq_part_sum $(\mathsf{x}:\mathbb{N}\to\mathbb{R}) := \lambda_{\mathsf{n}:\mathbb{N}}.(\mathsf{sum0\ n\ x})$.

Here sum0 sums the first n terms of a sequence.

If the sequence of partial sums of $\{x_n\}$ is Cauchy, then $\{x_n\}$ is said to be *convergent* (as a series). The sum of the series is defined as the limit of the sequence of partial sums.

Definition convergent $(\mathsf{x}:\mathbb{N}\to\mathbb{R})$:= (Cauchy_prop (seq_part_sum x)).

Definition series_sum $(\mathsf{x}:\mathbb{N}\to\mathbb{R})$ (H:(convergent x)) :=
(Lim (Build_CauchySeq (seq_part_sum x) H)).

The comparison test and the ratio test (Propositions 9 and 10 of [10, Chapter 2]) were formalized. Even though the formalization of these proofs presents nothing new, it is interesting to compare the statements of the theorems in the informal and the formal presentation. The comparison test is examined as an example; for the ratio test, the situation is pretty analogous.

Proposition 5.1.2 If $\{y_n\}$ is a convergent series of nonnegative terms, and if $|x_n| \leq y_n$ for each n, then $\{x_n\}$ converges.

A straightforward formalized counterpart of this proposition would be the following.

$$\forall_{x,y:\mathbb{N}\to\mathbb{R}}.(\mathsf{convergent}\ y)\to(\forall_{n:\mathbb{N}}.(0\leq(y\ n)))\to$$
$$(\forall_{n:\mathbb{N}}.(\mathsf{AbsIR}\ (x\ n))\leq(y\ n))\to(\mathsf{convergent}\ x).$$

Interestingly, just writing down the lemma in this way makes very clear something which is not obvious at all in the informal statement: the hypothesis that each y_n is nonnegative is redundant, since this is a trivial consequence of the facts that $|x_n|\leq y_n$ and that absolute values are always nonnegative.

Therefore, the following (simpler) version was formalized instead.

Lemma comparison : $\forall_{x,y:\mathbb{N}\to\mathbb{R}}.(\mathsf{convergent}\ y)\to$
$(\forall_{n:\mathbb{N}}.(\mathsf{AbsIR}\ (x\ n))\leq(y\ n))\to(\mathsf{convergent}\ x)$.

In practice, though, the hypotheses in this lemma are still too strong. Asymptotic behaviour is what determines convergence or divergence of a series, so the requirement that $|x_n|\leq y_n$ can be made only after a certain point. Thus the following was also formalized.

Lemma str_comparison : $\forall_{x,y:\mathbb{N}\to\mathbb{R}}.(\mathsf{convergent}\ y)\to$
$(\exists_{k:\mathbb{N}}.\forall_{n:\mathbb{N}}.(k\leq n)\to(\mathsf{AbsIR}\ (x\ n))\leq(y\ n))\to(\mathsf{convergent}\ x)$.

For the proof of the ratio test, one relies on the fact that geometric series of ratio between 0 and 1 converge; this was never explicitly mentioned in Bishop, but it had to be formalized and proved.

The C-CoRN library also includes some results on algebraic operations: if f and g are convergent series and c is a real number, then $f+g$, $-f$, $f-g$ and $c\times f$ are also convergent series with the expected sums.

Bishop's definitions of e and π as series are also formalized; the latter requires proving a criterion for convergence of alternate series. In each case, three steps are needed: first, defining the sequence to be summed; then, proving that this sequence is convergent (as a series); finally, defining the constant as the sum of that series. For example, for $e=\sum_{n=0}^{\infty}\frac{1}{n!}$ this is translated as follows, where[2] H : $\forall_{n:\mathbb{N}}.(\mathsf{fac}\ n)\ \#\ 0$.

Definition e_series := $\lambda_{n:\mathbb{N}}.1/(\mathsf{fac}\ n)//(\mathsf{H}\ n)$.

Lemma e_series_conv : (convergent e_series).

Definition E := (series_sum e_series e_series_conv).

In the case of π, the formalized proof of convergence turned out to be much more complex than the one in the reference book. Bishop's proof reads as follows.

[2] Recall the notation for division presented in Chapter 3.

PROPOSITION 5.1.3 A series $\sum_{n=1}^{\infty}(-1)^n x_n$ converges whenever $x_n \geq 0$ for all n and the sequence $\{x_n\}$ is decreasing and converges to 0.

PROOF. Consider positive integers m and n, with $m \geq n$. Then

$$\begin{aligned} 0 &\leq (x_n - x_{n+1}) + (x_{n+2} - x_{n+3}) + \ldots + (-1)^{m+n} x_m \\ &= (-1)^n \sum_{k=n}^{m} (-1)^k x_k \\ &= x_n - (x_{n+1} - x_{n+2}) - \ldots + (-1)^{m+n} x_m \\ &\leq x_n \end{aligned}$$

□

The problem is, this proof uses too many "obvious" facts about generalized associativity. When formalizing it, difficulties arise because there is no simple way to write the expression in the first line (at least without knowing whether the total number of terms is even or odd); and proving both equalities and the last inequality is best done by induction.

Thus, the formalized proof has to be done by even/odd-induction on m, i.e., by induction on m with two different induction cases according to whether m is even or odd. Besides being significantly more complicated than the simple proof above presented, this sheds no more light on why the stated result holds.

5.2 Continuous Functions

Bishop's presentation of continuous functions and differential calculus turned out to be quite precise, and yielded itself to formalization almost without any need for changes either in statements of lemmas or in their proofs (in contrast, as will be seen, to the situation that arose with integral calculus).

Interestingly, the main difficulties were met when dealing with some of the simplest notions.

First, Bishop's definition of totally bounded set (Definition 2.4.4) is an example of a frequently arising notion in mathematics where a variable number of existential quantifiers is present—namely, it depends on the first variable which is itself existentially quantified. Formalizing this requires some way to deal with finite sets, which is generally considered problematic within Type Theory.

The second problem is of a more general nature, and is related to the way concepts such as continuity and differentiability are defined for partial functions. Bishop defines such concepts first for compact intervals, and only later for arbitrary intervals, as is seen in Definitions 2.4.6 and 2.4.7. Although this is in itself not too difficult to handle, it requires some preliminary thought on how to represent the concept of "interval", and whether the first definition should be an instance of the second.

Finite sets and existential quantification

In mathematics and, in particular, in Real Analysis, statements of the form

$$\exists_{n\in\mathbb{N}}.\exists_{x_0}\ldots\exists_{x_n}.\varphi$$

very often appear. Common examples include the classical definition of compact set (more exactly, the notion of finite subcover); the notion of finitely derivable in logic; the property being in the transitive closure of a relation; the definition of uniform space; and, of course, the notion of totally bounded.

Formalizing these concepts always presents the same problem: the number of existential quantifiers at the beginning is itself an existentially quantified variable, and as such the whole expression cannot be written down directly in first order logic.

However, careful analysis of the examples presented above reveals a common characteristic: the *number* of quantifiers is not so relevant as the fact that the set of these is finite; i.e., the emphasis is on the existence of a *finite* set of elements satisfying some property, rather than on how many elements that set actually has.

A natural idea is, then, to regard the leading quantifiers as a single entity saying "there exists a finite set such that...", which can of course be expressed with a single existential quantifier. But how is the concept of finite set defined?

There are three different solutions commonly used in Type-Theory–based proof assistants in similar situations. One is to represent finite sets as functions with domain set $\{0,\ldots,n\}$ (with n existentially quantified); this is the standard approach in both PVS [61] and Mizar [51]. Another option is to identify finite sets with data structures such as lists and trees; this is a typical solution in programming languages, and is also part of the standard library of Coq [17]. A third approach is to define finite sets inductively: the empty set is finite, adding an element to a finite set yields a finite set; this has also been implemented in several different ways in Coq.

The last option was not considered when developing C-CoRN, the main reason being that it diverges significantly from standard mathematical practice. The advantages and disadvantages of representing finite sets as functions and as lists will now be discussed.

Mathematical tradition includes writing expressions like $x_0, \ldots, x_n$ to represent a finite set of x's. Taking into account that the notation x_i is also used in many contexts to denote the application of a function x with domain $\mathbb{N}$, it is natural to try to represent finite sets also as functions from an initial segment of the set of natural numbers.

Unfortunately, this is not trivial to do in Coq, and the same problems arise as with representation of partial functions in general. The type of a function from $\{x_0, \ldots, x_n\}$ to a set A will typically be something like $\Pi_{i:\mathbb{N}}.(i \leq n) \rightarrow A$; and all the questions discussed at the beginning of Chapter 3 arise again. The type of natural numbers has no setoid structure, so proof irrelevance has to be explicitly stated; and the result is that, although the original motivation was to stay as close as possible to mathematical tradition, the formalized statement looks very different indeed.

As a concrete example, Definition 2.4.4 of totally bounded set would look as follows.

$$
\begin{array}{l}
\mathsf{Definition\ totally\_bounded}\ (\mathsf{P}{:}\mathbb{R} \rightarrow \mathbf{Set}) : \mathbf{Set} := \\
\quad \forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \rightarrow \exists_{\mathsf{n}:\mathbb{N}}.\exists_{\mathsf{x}:\Pi_{\mathsf{i}:\mathbb{N}}.(\mathsf{i} \leq \mathsf{n}) \rightarrow \mathbb{R}}. \\
\qquad (\forall_{\mathsf{i,j}:\mathbb{N}}.\forall_{\mathsf{Hi,Hj}}.(\mathsf{i} = \mathsf{j}) \rightarrow (\mathsf{x\ i\ Hi}) = (\mathsf{x\ j\ Hj})) \wedge \\
\qquad (\forall_{\mathsf{y}:\mathbb{R}}.(\mathsf{P\ y}) \rightarrow \exists_{\mathsf{k}:\mathbb{N}}.\exists_{\mathsf{Hk}}.(\mathsf{AbsSmall\ e\ y} - (\mathsf{x\ k\ Hk}))).
\end{array}
$$

(This is a little simplified; the inability to existentially quantify over propositions to obtain a proof in **Set** makes the final part of the statement slightly more complicated than the above presentation suggests.)

Even though it diverges somewhat more from standard mathematical practice, the use of lists proves to be a little bit more satisfactory in general, and certainly in this case in particular. For every type A, $(\mathsf{list\ A})$ is the type of (finite) lists over A; and existential quantification is now direct. The only issue is, one can not directly quantify over members of a list, so a membership predicate member must be used. This can be defined using either intensional or extensional equality; in this situation, the latter is preferable, as all work is being done using extensional equality.

The previous definition now looks much simpler[3].

$$
\begin{array}{l}
\mathsf{Definition\ totally\_bounded}\ (\mathsf{P}{:}\mathbb{R} \rightarrow \mathbf{Set}) : \mathbf{Set} := \\
\quad \forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \rightarrow \exists_{\mathsf{l}:(\mathsf{list}\ \mathbb{R})}. \\
\qquad \forall_{\mathsf{x}:\mathbb{R}}.(\mathsf{P\ x}) \rightarrow \exists_{\mathsf{y}:\mathbb{R}}.(\mathsf{member\ y\ l}) \wedge (\mathsf{AbsSmall\ e\ x} - \mathsf{y}).
\end{array}
$$

[3] In both this and the previous example, the side conditions stating that the finite set is not only a set of real numbers, but also of real numbers satisfying P have been omitted for simplicity.

A small library of lists of real numbers, including among others mapping functions, minimum and maximum, was developed.

In contrast to this example, in Section 5.4 a similar situation will be described where the formalization via finite domain functions proves more satisfactory than its counterpart using lists. So far, the situation in C-CoRN seems to indicate that the choice between one way to formalize finite sets and the other must be made again and again for each specific application.

Intervals

Intervals form a particular class of subsets of the real line. Every interval I enjoys several nice properties: it is connected; it is extensional (in the sense that if $x \in I$ and $x = y$, then $y \in I$, for any real numbers x and y); and properties such as being non-empty or proper (having more than one point) can be nicely stated as relations which only involve its endpoints.

In order to deal with intervals in a nice way, it is useful to have a special (syntactic) type of intervals rather than just identifying them with a particular class of predicates on the real numbers. In this type there are nine constructors, corresponding to the nine kinds of intervals: the real line (one), semi-lines, open or closed at their endpoint (four), and finite intervals, open or closed at either end (four). The names of the constructors are supposed to be mnemonic.

To each element of this syntactic type a predicate is assigned in the obvious way by a function iprop. This function is then declared as a coercion, so that from the user's perspective an interval *is* a subset of the real numbers.

In Figure 5.1 the inductive definition of the type of intervals and the iprop function on that type are shown.

The notions of proper, finite and compact are all similarly defined for this inductive type. For every interval I, the term (iprop_wd I) proves that the characteristic predicate of I is extensional.

However, at the first stage of formalizing notions like continuity and differentiability, when only compact intervals are considered, this type is not useful at all. The fact that a compact interval is of the form $[a, b]$ is central to many applications, and it is handy to be able to access a and b directly rather than via some operation on intervals. Therefore, such definitions will always be made by explicitly parameterizing on a, b and a proof that $a \leq b$.

Continuity

Once the problems addressed above have been solved, formalizing Bishop's notion of continuity is quite straightforward. Continuity is defined in two

```
Inductive interval : Set:=
| realline : interval
| openl    : ℝ → interval
| openr    : ℝ → interval
| closel   : ℝ → interval
| closer   : ℝ → interval
| olor     : ℝ → ℝ → interval
| olcr     : ℝ → ℝ → interval
| clor     : ℝ → ℝ → interval
| clcr     : ℝ → ℝ → interval.
```

```
Definition iprop (I:interval) :=
λx:ℝ.match I with
| realline    ⇒ ⊤
| (openl a)   ⇒ a < x
| (openr b)   ⇒ x < b
| (closel a)  ⇒ a ≤ x
| (closer b)  ⇒ x ≤ b
| (olor a b)  ⇒ a < x < b
| (olcr a b)  ⇒ a < x ≤ b
| (clor a b)  ⇒ a ≤ x < b
| (clcr a b)  ⇒ a ≤ x ≤ b
end.
```

Figure 5.1: The inductive type of intervals and iprop

steps: first for compact intervals, parameterizing on the endpoints a and b, left implicit, and a proof Hab : $(a \leq b)$; $[a, b]$ denotes the compact interval $[a, b]$[4]. The possibly weird-looking application of a compact interval to a real number can be understood if one remembers that, subsets being predicates, membership is simply application.

$$\begin{array}{l}\mathsf{Definition\ Continuous\_I\ (F{:}PartIR) := (included\ [a,b]\ (dom\ F))} \wedge \\ \quad (\forall_{e:\mathbb{R}}.(0 < e) \rightarrow \exists_{d:\mathbb{R}}.(0 < d) \wedge \forall_{x,y:\mathbb{R}}.([a,b]\ x) \rightarrow ([a,b]\ y) \rightarrow \\ \qquad \forall_{Hx,Hy}.((\mathsf{AbsIR}\ x - y) \leq d) \rightarrow ((\mathsf{AbsIR}\ (F\ x\ Hx) - (F\ y\ Hy)) \leq e)).\end{array}$$

Continuity is then defined for an arbitrary interval I.

$$\begin{array}{l}\mathsf{Definition\ Continuous\ (F{:}PartIR) := (included\ I\ (dom\ F))} \wedge \\ \quad (\forall_{a,b:\mathbb{R}}.\forall_{Hab:(a \leq b)}.(\mathsf{included}\ [a,b]\ I) \rightarrow (\mathsf{Continuous\_I\ Hab\ F})).\end{array}$$

The image of a continuous function on a compact interval has a supremum and an infimum; the formalized proof follows Bishop's closely. The least upper bound of a set is defined; the least upper bound of a function f in an interval I is simply the relativization of that notion to the image $f[I]$. For the greatest lower bound the situation is similar. Using these notions, the norm of a continuous function is defined.

$$\begin{array}{l}\mathsf{Definition\ set\_lub\_IR}\ (P{:}\mathbb{R} \rightarrow \mathbf{Set})\ (a{:}\mathbb{R}) := \\ \quad (\forall_{x:\mathbb{R}}.(P\ x) \rightarrow (x \leq a)) \wedge (\forall_{e:\mathbb{R}}.(0 < e) \rightarrow \exists_{x:\mathbb{R}}.(P\ x) \wedge ((a - x) < e)).\end{array}$$

$$\begin{array}{l}\mathsf{Definition\ fun\_image\ (F{:}PartIR)}\ (P{:}\mathbb{R} \rightarrow \mathbf{Set}) : \mathbb{R} \rightarrow \mathbf{Set} := \\ \quad \lambda_{x:\mathbb{R}}.\exists_{y:\mathbb{R}}.(P\ y) \wedge (\mathsf{dom}\ F\ y) \wedge \forall_{Hy}.(F\ y\ Hy){=}x.\end{array}$$

[4]The actual Coq term is (compact a b Hab), where Hab : $(a \leq b)$.

Definition fun_lub_IR (F:PartIR) (P:$\mathbb{R}\rightarrow\mathbf{Set}$) (a:$\mathbb{R}$) :=
(set_lub_IR (fun_image F P) a).

Lemma Continuous_I_imp_lub : $\forall_{\mathsf{a,b}:\mathbb{R}}.\forall_{\mathsf{Hab}:(\mathsf{a}\leq\mathsf{b})}.$
$\forall_{\mathsf{F:PartIR}}.$(Continuous_I Hab F)$\rightarrow\exists_{\mathsf{x}:\mathbb{R}}.$(fun_lub_IR F $[\mathsf{a},\mathsf{b}]$ x).

Definition lub_funct
(a,b:$\mathbb{R}$) (Hab:(a $\leq$ b)) (F:PartIR) (contF:(Continuous_I Hab F)) :=
(ProjS1 (Continuous_I_imp_lub a b Hab F contF)).

Definition Norm_Funct
(a,b:$\mathbb{R}$) (Hab:(a $\leq$ b)) (F:PartIR) (contF:(Continuous_I Hab F)) :=
(Max (lub_funct a b Hab F contF) $-$(glb_funct a b Hab F contF)).

This norm operator has all the expected properties: it is nonnegative and it is the least upper bound of the absolute value of f on I. The first four arguments of lub_funct and Norm_Funct can and will be left implicit.

Continuity is preserved through the algebraic operations (addition, subtraction and multiplication), as well through the taking of the maximum, the minimum or the absolute value. For composition and division, side conditions have to be assumed as was discussed following Lemmas 2.4.8 and 2.4.9. To deal with these situations, auxiliary notions of a function f "mapping compacts of an interval I into compacts in the domain of g" and "being bounded away from 0 on an interval I" were defined, both assuming I compact and in the general case. In the following lemmas, F, G : PartIR are universally quantified.

Definition maps_into_compacts (a b c d:$\mathbb{R}$) (Hab:(a $\leq$ b)) (Hcd:(c $\leq$ d)) :=
(c $<$ d) $\wedge$ (included [c,d] (dom G)) $\wedge$
($\forall_{\mathsf{x}:\mathbb{R}}.\forall_{\mathsf{Hx}}.$($[\mathsf{a},\mathsf{b}]$ x)$\rightarrow$($[\mathsf{c},\mathsf{d}]$ (F x Hx))).

Definition maps_compacts_into (I J:interval) := $\forall_{\mathsf{a,b}:\mathbb{R}}.\forall_{\mathsf{Hab}:(\mathsf{a}\leq\mathsf{b})}.$
(included $[\mathsf{a},\mathsf{b}]$ I)$\rightarrow\exists_{\mathsf{c,d}:\mathbb{R}}.$(c $<$ d) $\wedge$ (included $[\mathsf{c},\mathsf{d}]$ J)$\wedge$
($\forall_{\mathsf{x}:\mathbb{R}}.\forall_{\mathsf{Hx}}.$($[\mathsf{a},\mathsf{b}]$ x)$\rightarrow$($[\mathsf{c},\mathsf{d}]$ (F x Hx))).

Definition bnd_away_zero (P:$\mathbb{R}\rightarrow\mathbf{Set}$) := (included P (dom F)) $\wedge$
$\exists_{\mathsf{c}:\mathbb{R}}.$(0 $<$ c) $\wedge$ ($\forall_{\mathsf{y}:\mathbb{R}}.$(P y)$\rightarrow\forall_{\mathsf{Hy}}.$(c $\leq$ (AbsIR (F y Hy)))).

Definition bnd_away_zero_in_P (P:$\mathbb{R}\rightarrow\mathbf{Set}$) := $\forall_{\mathsf{a,b}:\mathbb{R}}.$(a $\leq$ b)$\rightarrow$
(included $[\mathsf{a},\mathsf{b}]$ (dom F))$\rightarrow$(bnd_away_zero $[\mathsf{a},\mathsf{b}]$ F).

It should be stressed that the last two definitions, while both referring to arbitrary sets of reals, are *not* equivalent. The reciprocal function is bounded away from zero in every compact interval contained in $(0,+\infty)$, but it is not bounded away from zero on the latter set. That is, the lemma (bnd_away_zero_in_P 1/FId (openl 0)) is provable, whereas constructively (bnd_away_zero 1/FId (openl 0)) does not hold.

At this stage of the formalization, there is everywhere an apparent duplication of all results due to their presentation first for compact intervals and only then for arbitrary ones. Although this might at first appear strange, it is in fact very close to Bishop's way of working. Besides the point that the definitions are made in two steps, his proofs always begin with "it is enough to consider the case when I is compact..." or words to that effect; therefore, also his proofs are implicitly in two stages. Formalizing everything at these two levels clearly separates the core of the proof (which is always done at the level of compact intervals) from the justification of the "without loss of generality" step.

It is worth pointing out that the proofs of results in arbitrary intervals usually amount to no more than simply manipulating inclusions and using the like results for compact intervals.

Sequences and Series of Continuous Functions

Bishop's theory of sequences and series of functions, defined only for continuous functions, can be likewise straightforwardly formalized.

The formalized counterparts of these definitions and results resemble very closely the similar material on sequences and series of real numbers. Furthermore, since once again all definitions are first made only on compact intervals and later on generalized for arbitrary intervals, they are also presented twice.

As before, the results stated for compact intervals require formalizing the core of the proof as presented in Bishop, whereas the proofs of their generalizations to arbitrary intervals amount to a justification of the "without loss of generality" step.

All the results in this section which generalize results already proved for sequences or series of real numbers are uniformly named: if lemma refers to a result about real numbers, then fun_lemma refers to the same result about real functions on a compact interval, and fun_lemma_IR its generalization to arbitrary intervals. The various definitions of Cauchy sequence do not always correspond directly to previously existing ones, though, so this rule does not hold for their names.

For the remainder of the chapter, the following terms will be relevant:

- (conv_fun_seq a b Hab f contf) states that the sequence of continuous functions f : $\mathbb{N} \rightarrow$ PartIR converges on the compact interval $[a, b]$;

- the term (conv_fun_seq' a b Hab f F contf contF) also indicates that the limit is the continuous function F;

- if an arbitrary interval J is considered instead, the corresponding terms are (conv_fun_seq_IR J f contf) and (conv_fun_seq′_IR J f F contf contF).

An important new feature of this piece of the formalization (as compared to the corresponding theory of real number series) is the ability to define functions by power series. Power series are defined, given a sequence $\{a_n\}$ and a point x_0, by the formula $\sum_{n=0}^{\infty} a_n(x-x_0)^n$; the sequence being summed is directly formalized[5] as follows.

Definition FPowerSeries := $\lambda_{n:\mathbb{N}}.$(a n) $\times$ ((FId $-$ λx.x$_0$)^n).

This is defined together with the Dirichlet criterion establishing its convergence as a series on an interval $(x_0 - r, x_0 + r)$ under suitable hypotheses[6].

Hypothesis Ha : $\exists_{N:\mathbb{N}}.\forall_{n:\mathbb{N}}.$(N $\leq$ n)$\rightarrow$
(AbsIR (a (S n))) $\leq$ (1/r)$\times$(AbsIR (a n)).

Lemma Dirichlet_crit :
(fun_series_abs_convergent_IR (olor x$_0$$-$r x$_0$+r) FPowerSeries).

Of special interest for the applications to transcendental functions are power series of the special kind $\sum_{n=0}^{\infty} \frac{a_n}{n!}(x-x_0)^n$, which under similar conditions on a converge absolutely on the whole real line.

Definition FPowerSeries' := $\lambda_{n:\mathbb{N}}.$((a n)/(fac n)) $\times$ ((FId $-$ λx.x$_0$)^n).

Hypothesis Ha' : $\exists_{N:\mathbb{N}}.\exists_{c:\mathbb{R}}.$(0 < c) $\wedge$
($\forall_{n:\mathbb{N}}.$(N $\leq$ n)$\rightarrow$(AbsIR (a (S n))) $\leq$ c $\times$ (AbsIR (a n))).

Lemma FPowerSeries'_conv' :
(fun_series_abs_convergent_IR realline FPowerSeries').

5.3 Differential Calculus

After the discussion in the previous section, the formalization of Bishop's definition of derivative (Definition 2.4.10) should come as no surprise. In the first definition, a, b : $\mathbb{R}$ are implicit arguments and Hab : (a < b); in the second, I : interval is implicit and pI : (proper I).

Definition Derivative_I (F F':PartIR) := (included [a, b] (dom F)) $\wedge$
(included [a, b] (dom F')) $\wedge$ $\forall_{e:\mathbb{R}}.$(0 < e)$\rightarrow\exists_{d:\mathbb{R}}.$(0 < d)
$\forall_{x,y:\mathbb{R}}.$([a, b] x)$\rightarrow$([a, b] y)$\rightarrow\forall_{Hx,Hy,Hx'}.$((AbsIR x $-$ y) $\leq$ d)$\rightarrow$
(AbsIR ((F y Hy)$-$(F x Hx))$-$(F' x Hx')$\times$(y$-$x)) $\leq$ e$\times$(AbsIR y$-$x).

[5] Recall the notation introduced in Section 5.1.

[6] Where the proof term for r # 0 has been left out.

Definition Derivative (F F':PartIR) :=
(included I (dom F)) $\wedge$ (included I (dom F')) $\wedge$
($\forall_{a,b:\mathbb{R}}.\forall_{Hab:(a<b)}$.(included $[a,b]$ I) $\rightarrow$ (Derivative_I Hab F F$'$)).

There is one slight difference between Definition 2.4.10 and this formalization: Bishop only states the property of "being a derivative of" for continuous functions, whereas here it is done without that assumption. As it turns out, these formulations are equivalent, since from this definitions it is trivial to prove that both functions involved are continuous.

Lemma deriv_imp_contin_I : $\forall_{F,G:PartIR}.\forall_{a,b:\mathbb{R}}.\forall_{Hab':(a<b)}.\forall_{Hab:(a\leq b)}$.
(Derivative_I Hab$'$ F G) $\rightarrow$ (Continuous_I Hab F).

Lemma deriv_imp_contin'_I : $\forall_{F,G:PartIR}.\forall_{a,b:\mathbb{R}}.\forall_{Hab':(a<b)}.\forall_{Hab:(a\leq b)}$.
(Derivative_I Hab$'$ F G) $\rightarrow$ (Continuous_I Hab G).

Notice the difference in the two inequalities which are being quantified upon. In the definition of continuity, all that is needed is for $[a, b]$ to be a compact interval, i.e., $a \leq b$; however, the notion of derivative is only meaningful for *proper* compact intervals (with more than one point). It is straightforward to prove that $a \leq b$ whenever $a < b$; but for generality it is better to formalize the lemmas as just stated, as they will then be more widely applicable.

The usual rules for computing the derivative of a function are all formalized and proved. In the case of division and composition, similar side conditions to those of Lemmas 2.4.8 and 2.4.9 must be assumed. Of course, all these lemmas must be proved both for compact and for arbitrary intervals: if $\mathsf{a}, \mathsf{b} : \mathbb{R}$, Hab : $(\mathsf{a} < \mathsf{b})$ and F, G : PartIR with derivatives respectively F$'$ and G$'$, then the following hold.

Lemma Derivative_I_const : $\forall_{c:\mathbb{R}}$.(Derivative_I Hab λx.c λx.0).

Lemma Derivative_I_plus : (Derivative_I Hab F+G F'+G').

Lemma Derivative_I_mult : (Derivative_I Hab F$\times$G F$\times$G'+F'$\times$G).

Lemma Derivative_I_recip : (bnd_away_zero $[a,b]$ F) $\rightarrow$
(Derivative_I Hab 1/F $-$(F'/(F$\times$F))).

If J : interval and pJ : (proper J), the following are the corresponding lemmas.

Lemma Derivative_const : $\forall_{c:\mathbb{R}}$.(Derivative pJ λx.c λx.0).

Lemma Derivative_plus : (Derivative pJ F+G F'+G').

Lemma Derivative_mult : (Derivative pJ F$\times$G F$\times$G'+F'$\times$G).

Lemma Derivative_recip : (bnd_away_zero_in_P F J) $\rightarrow$
(Derivative pJ 1/F $-$(F'/(F$\times$F))).

The assumption $a < b$ for compact intervals is needed to prove the fundamental fact, only briefly referred to by Bishop, that there is extensionally at most one derivative of a function on a given interval.

Lemma Derivative_I_unique : $\forall_{\mathsf{F,G,H:PartIR}}.\forall_{\mathsf{a,b}:\mathbb{R}}.\forall_{\mathsf{Hab:(a<b)}}.$
(Derivative_I Hab F G) $\rightarrow$ (Derivative_I Hab F H) $\rightarrow$ (Feq $[\mathsf{a},\mathsf{b}]$ G H).

This last result has some interesting consequences, and illustrates quite well some of the difficulties arising when formalizing mathematics. Since there is at most one derivative of f on an interval I, the notation f' is used informally to represent this function, and *any* derivative of f on that interval will be represented by f'—which represents no problem from the mathematical perspective, since any such function will always be extensionally equal to f' and all mathematics is extensional.

But this is simply not doable inside a formal system. If f' and g are two derivatives of f on I then the previous lemma will always have to be explicitly invoked to justify replacing one by the other. This in turn often significantly complicates proofs, especially algebraic proofs where some terms are applications of f' and others are applications of g to the same arguments.

Furthermore, in this situation automation can be of little help. Though tactics have been implemented to help proving that a function is the derivative of the other, and these can be combined with the previous lemma to prove $f' =_I g$ by reducing this to "f' is a derivative of f on I" and "g is a derivative of f on I", several problems still remain. First, even in this simple situation, how can the system figure out what f should be? Even worse, in most situations the goal is not directly $f' =_I g$, but rather an expression like $f'(c) = g(c)$; the user can easily recognize this as being a consequence of $f' =_I g$ for a specific I, but teaching the system to do this is quite another matter. So far all proofs of this kind have to be begun by the user, automation only being able to help after the initial two or three steps of reasoning.

Another unexpected consequence is the at first sight strange aspect that some of the familiar theorems of Analysis get. Results like Rolle's Theorem or Taylor's Theorem, which speak about the derivative(s) of a function, now must become universally quantified over those derivatives. As an example, given a proper compact interval $[\mathsf{a},\mathsf{b}]$ with Hab : $(\mathsf{a} < \mathsf{b})$ and F : PartIR, the formalized counterpart of Theorem 2.4.12 looks like this.

Theorem Rolle : $\forall_{\mathsf{F':PartIR}}.$(Derivative_I Hab F F') $\rightarrow$
$(\forall_{\mathsf{Ha,Hb}}.(\mathsf{F\ a\ Ha}) = (\mathsf{F\ b\ Hb})) \rightarrow \forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \rightarrow$
$\exists_{\mathsf{x}:\mathbb{R}}.([\mathsf{a},\mathsf{b}]\ \mathsf{x}) \wedge \forall_{\mathsf{Hx}}.(\mathsf{AbsIR}\ (\mathsf{F'\ x\ Hx})) \leq \mathsf{e}.$

On a different note, a related problem arises when the notion of differentiability is considered. Once again, from the mathematical perspective, this is no problem: f is differentiable on I if it has a derivative on that interval, and since derivatives are unique this function can be denoted by f'. But how is f' precisely defined?

Trying to answer this question in Coq reveals some interesting issues. First, as was discussed in Chapter 3, the type of partial functions over a setoid has itself type **Type**; this is unfortunate, as it means that the following "definition" fails.

"Definition" Diffble_I (Hab:(a < b)) (F:PartIR) :=
    $\exists_{\mathsf{F'}:\mathsf{PartIR}}$.(Derivative_I Hab F F')

The problem is, even though the previous statement is possible to write down in Coq, there will be no way to extract the function F' from a proof of (Diffble_I Hab F). This arises because of the restrictions to the elimination rules for inductive types—with such a definition, the type of (Diffble_I Hab F) is **Set**, and it would be eliminated to create an object of type PartIR which itself has type **Type**.

Instead of being a drawback, this limitation actually turns out to be an advantage because it leads to a more careful analysis of the concept being formalized. In fact, the previously proposed definition is not totally satisfactory for reasons other than just typing constraints. Take the derivative function itself. It is reasonable to assume (and probably expected by most) that, if f is differentiable on an interval I and $J \subseteq I$, then the derivative of f on I will extend the derivative of f on J, in the sense that these two functions will coincide where they are both defined, and the first one should be defined whenever the second one is. But this is not true: since the quantifier ranges over *all* partial functions but the predicate in its scope only specifies their behaviour on the interval under consideration, there is no restriction on what a derivative of f on J would do outside J. In particular, it could very well *not* be a derivative of f on I.

But this situation is very unnatural. There is, however, a very nice and easy way around it, and that is also in line with the discussion following Definition 2.4.10: the *canonical* derivative of f on I is defined to be the derivative of f on I whose domain is *exactly* I. A function f is said to be differentiable on I iff it has a canonical derivative on I.

This notion clearly coincides with the previous one, since any function can be restricted to a subset of its domain. Also, it solves the typing problem above discussed: since the domain I of the would-be derivative is now known, there is no need to quantify over all partial functions, but only over partial functions with that domain. These are exactly[7] the setoid functions from I to $\mathbb{R}$, and "differentiable function" can now be defined.

Definition Diffble_I (Hab:(a < b)) (F:PartIR) :=
$\quad\exists_{\mathsf{f}':(\mathsf{CSetoid\_fun}\ \{\mathbb{R}|[\mathsf{a},\mathsf{b}]\}\ \mathbb{R})}$.(Derivative_I Hab F f').

The (canonical) derivative of f is just the witness extraction from a proof term of this type.

Using this predicate a second version of Rolle's Theorem can be written.

Theorem Rolle' : $\forall_{\mathsf{H}:(\mathsf{Diffble\_I\ Hab\ F})}$. let F':=(ProjS1 H) in
$\quad(\forall_{\mathsf{Ha},\mathsf{Hb}}.(\mathsf{F\ a\ Ha}) = (\mathsf{F\ b\ Hb}))\rightarrow\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e})\rightarrow$
$\qquad\exists_{\mathsf{x}:\mathbb{R}}.([\mathsf{a},\mathsf{b}]\ \mathsf{x}) \wedge \forall_{\mathsf{Hx}}.(\mathsf{AbsIR\ (F'\ x\ Hx)}) \leq \mathsf{e}.$

Interestingly, this statement looks much closer to the usual presentation in standard mathematics texts, but it is not so useful as the previous one in practice. The problem is, it can only be applied to goals involving the canonical derivative of F, whereas the previous version can be used in general.

The main corollary of Rolle's Theorem is the Law of the Mean, already stated as Corollary 2.4.13. Just like Rolle's Theorem, it can also be stated with or without making the derivative of f explicit.

Lemma Law_of_the_Mean_I : $\forall_{\mathsf{F}':\mathsf{PartIR}}.(\mathsf{Derivative\_I\ Hab\ F\ F'})\rightarrow\forall_{\mathsf{Ha},\mathsf{Hb}}.$
$\quad\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e})\rightarrow\exists_{\mathsf{x}:\mathbb{R}}.([\mathsf{a},\mathsf{b}]\ \mathsf{x}) \wedge \forall_{\mathsf{Hx}}.$
$\qquad((\mathsf{AbsIR\ ((F\ b\ Hb)}-(\mathsf{F\ a\ Ha}))-(\mathsf{F'\ x\ Hx})\times(\mathsf{b}-\mathsf{a})) \leq \mathsf{e}).$

Lemma Law_of_the_Mean'_I : $\forall_{\mathsf{HF}:(\mathsf{Diffble\_I\ Hab\ F})}.\forall_{\mathsf{Ha},\mathsf{Hb}}.$
$\quad\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e})\rightarrow\exists_{\mathsf{x}:\mathbb{R}}.([\mathsf{a},\mathsf{b}]\ \mathsf{x}) \wedge \forall_{\mathsf{Hx}}.$
$\qquad(\mathsf{AbsIR\ ((F\ b\ Hb)}-(\mathsf{F\ a\ Ha}))-((\mathsf{ProjS1\ HF})\ \mathsf{x\ Hx})\times(\mathsf{b}-\mathsf{a})) \leq \mathsf{e}.$

The proof of the Law of the Mean relies heavily on the automation tactics which were built in parallel with the formalization, as described in detail in Section 4.4, from page 79 onwards.

For practical applications, it is useful to generalize the Law of the Mean in two different ways. First, the restriction above, that a and b are the endpoints of a proper interval, can be too strong; one can imagine wanting to apply the Law of the Mean without knowing whether $a < b$! Suppose then that I : interval with pI : (proper I).

Theorem Law_of_the_Mean : $\forall_{\mathsf{F}':\mathsf{PartIR}}.(\mathsf{Derivative\ pI\ F\ F'})\rightarrow$
$\quad\forall_{\mathsf{a},\mathsf{b}:\mathbb{R}}.(\mathsf{I\ a})\rightarrow(\mathsf{I\ b})\rightarrow\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e})\rightarrow$
$\qquad\exists_{\mathsf{x}:\mathbb{R}}.([(\mathsf{Min\ a\ b}),(\mathsf{Max\ a\ b})]\ \mathsf{x}) \wedge \forall_{\mathsf{Ha},\mathsf{Hb},\mathsf{Hx}}.$
$\qquad\quad(\mathsf{AbsIR\ ((F\ b\ Hb)}-(\mathsf{F\ a\ Ha}))-(\mathsf{F'\ x\ Hx})\times(\mathsf{b}-\mathsf{a})) \leq \mathsf{e}.$

[7] More precisely, these are isomorphic to these functions; the isomorphism will be left implicit in this presentation.

A few remarks are due about this lemma.

First, it is important to notice that it makes indeed no requirements over the order relation between a and b, therefore solving the restriction above mentioned.

Second, there is no need to generalize Rolle's Theorem in a similar way: since this last lemma can be proved directly from the first version of the Law of the Mean, the proof does not require it; and as no restrictions are placed at all on $f(a)$ and $f(b)$, Rolle's Theorem is simply a special case of this lemma. It should also be pointed out that when Rolle's Theorem is explicitly called for, its formulation in terms of compact intervals is usually more practical.

For a similar reason, it is not very interesting to formulate this lemma in terms of a differentiable function f. Even in the presence of a goal which involves the derivative of f as obtained from a proof that f is differentiable, this lemma can be applied, and the remaining goal (to prove that f' is indeed a derivative of f) is trivial to prove.

Finally, as was also discussed in Section 2.4, the most useful formulation of the Law of the Mean is Equation (2.2). This is straightforward to formalize and prove given all the previous results. As before, $\mathsf{F} : \mathsf{PartIR}$ and $\mathsf{I} : \mathsf{interval}$ are implicit.

$$
\begin{array}{l}
\mathsf{Theorem\ Law\_of\_the\_Mean\_ineq} : \forall_{\mathsf{F}':\mathsf{PartIR}}.(\mathsf{Derivative\ pI\ F\ F}') \rightarrow \\
\quad \forall_{\mathsf{a,b}:\mathbb{R}}.(\mathsf{I\ a}) \rightarrow (\mathsf{I\ b}) \rightarrow \forall_{\mathsf{e}:\mathbb{R}}.\forall_{\mathsf{c}:\mathbb{R}}. \\
\qquad (\forall_{\mathsf{x}:\mathbb{R}}.([(\mathsf{Min\ a\ b}), (\mathsf{Max\ a\ b})]\ \mathsf{x}) \rightarrow \forall_{\mathsf{Hx}}.(\mathsf{AbsIR}\ (\mathsf{F}'\ \mathsf{x\ Hx})) \leq \mathsf{c}) \rightarrow \\
\qquad\quad \forall_{\mathsf{Ha,Hb}}.((\mathsf{F\ b\ Hb}) - (\mathsf{F\ a\ Ha})) \leq \mathsf{c} \times (\mathsf{AbsIR\ b} - \mathsf{a}).
\end{array}
$$

Taylor's Theorem

Taylor's Theorem presents some more subtle issues of a different kind. The theorem, stated as Theorem 2.4.16, gives a polynomial approximation of an $(n+1)$ times differentiable function in terms of its derivatives together with an estimate of the error.

The first step in its formalization is, therefore, defining the n^{th} order derivative of a function. This is an inductive definition, to which similar comments apply as those made about the definition of differentiable function.

$$
\begin{array}{l}
\mathsf{Fixpoint\ Derivative\_I\_n\ (Hab{:}(a < b))\ (n{:}}\mathbb{N}) := \lambda_{\mathsf{F,Fn:PartIR}}.\ \mathsf{match\ n\ with} \\
\quad \mid \mathsf{O} \quad \Rightarrow \mathsf{Feq\ I\ F\ Fn} \\
\quad \mid \mathsf{S\ p} \Rightarrow \exists_{\mathsf{f}':(\mathsf{CSetoid\_fun}\{\mathbb{R}|[\mathsf{a,b}]\}\ \mathbb{R})}. \\
\qquad\qquad\qquad (\mathsf{Derivative\_I\ Hab\ F\ f'}) \wedge (\mathsf{Derivative\_I\_n\ Hab\ p\ f'\ Fn}) \\
\quad \mathsf{end}.
\end{array}
$$

Unlike the first order case, n^{th} order differentiable functions are best defined directly through a similar recursive definition than via an existential quantifier.

Next, similar results are proved as those for the first order case: both relations are preserved by functional equality; n times differentiable functions are continuous; the same holds for their n^{th} order derivative; if f has an n^{th} order derivative then f is n times differentiable; and n^{th} order derivatives are unique. Furthermore, when $n = 1$ these concepts coincide with the previous notions of derivative and differentiable function.

Then, some more interesting (but nevertheless mathematically trivial) results are proved: if f is n times differentiable then it is also m times differentiable for every $m \leq n$; and it is possible to define a finite sequence $\{f^{(i)}\}_{i=0}^{n}$ of functions such that $f^{(0)} =_I f$ and $f^{(i+1)} =_I (f^{(i)})'$. From the last result an operator can be defined which will be fundamental to state Taylor's Theorem.

All these results are straightforwardly generalized to arbitrary intervals as usual.

The proof of Taylor's Theorem is not too different from the one in [10]. The real difficulty lies in writing down all the auxiliary notions needed to *state* the theorem.

The formalization proceeds in two steps. First, two distinct points a and b are chosen within an interval I where the function f is $n+1$ times differentiable. In this context, the family of i^{th}-order derivatives of f for $i \leq n+1$ can be defined as a Coq term $\mathsf{fi} : \Pi_{\mathsf{i}:\mathbb{N}}.(\mathsf{i} \leq \mathsf{n}) \rightarrow \mathsf{PartIR}$.

Next, the sequence of functions

$$\left\{\lambda_{\mathsf{x}}.\frac{f^{(i)}(a)}{i!}(x-a)^i\right\}_{i=0}^{n}$$

is defined as another operator funct_i. Here, a proof term (denoted by Ha) is needed to ensure fi can be applied to a.

Local funct_i := $\lambda_{\mathsf{i}:\mathbb{N}}.\lambda_{\mathsf{Hi}:(\mathsf{i}\leq\mathsf{n})}.(\lambda\mathsf{x}.(\mathsf{fi\ i\ Hi\ a\ Ha})/(\mathsf{fac\ i}))\times(\mathsf{FId}-\lambda\mathsf{x.a})^{\mathsf{i}}$.

Then, Taylor_seq is defined as the sum of all terms of this finite sequence. This is still a function, which can be applied to b (naturally there is a proof term Hb of this) and then subtracted from $f(b)$.

Definition Taylor_rem := (F b Hb)−(Taylor_seq b Hb).

Similarly, deriv_Sn is defined to represent the sequence

$$\left\{\lambda_{\mathsf{x}}.\frac{f^{(i)}(x)}{i!}(b-x)^i\right\}_{i=0}^{n+1}$$

whose last term appears in the statement of Taylor's Theorem, but which is needed as a whole in its proof.

Finally, the first formalized version of Theorem 2.4.16 reads as follows.

Lemma Taylor_lemma : $\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \rightarrow \exists_{\mathsf{c}:\mathbb{R}}.([(\mathsf{Min\ a\ b}),(\mathsf{Max\ a\ b})]\ \mathsf{c}) \wedge$
$\forall_{\mathsf{Hc}}.(\mathsf{AbsIR\ Taylor\_rem}-(\mathsf{deriv\_Sn\ (S\ n)\ c\ Hc})\times(\mathsf{b}-\mathsf{a})) \leq \mathsf{e}.$

The generalization proceeds by choosing two arbitrary points in I and considering three cases:

- $a < b$: then Taylor's Theorem can be trivially reduced to the previous lemma, with $\min(a, b) = a$ and $\max(a, b) = b$;
- $a > b$: similarly, but now $\min(a, b) = b$ and $\max(a, b) = a$;
- else, $|a - b|$ can be taken to be "sufficiently small". This trivial step in Bishop's informal proof turns out to be quite tough to formalize, as a precise (positive) upper bound on $|a - b|$ has to be made explicit.

The final version of the Theorem looks like this.

Theorem Taylor' : $\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \rightarrow \exists_{\mathsf{c}:\mathbb{R}}.([(\mathsf{Min\ a\ b}),(\mathsf{Max\ a\ b})]\ \mathsf{c}) \wedge$
$\forall_{\mathsf{Hc}}.(\mathsf{AbsIR\ Taylor\_rem}-(\mathsf{deriv\_Sn\ (S\ n)\ c\ Hc})\times(\mathsf{b}-\mathsf{a})) \leq \mathsf{e}.$

This is strikingly similar to the previous one. The differences lie in the context: now no order is assumed between a and b; and f is assumed to be $n + 1$ times differentiable on I, rather than on the (potentially) smaller interval $[\min(a, b), \max(a, b)]$.

In the case where an explicit sequence of derivatives of order up to $n + 1$ of f is known, Taylor_rem and deriv_Sn can be defined directly in terms of this sequence and a similar result can be proved. This turns out to be very helpful when working with functions defined by series.

5.4 Integral Calculus

Integration turned out to be by far the most difficult process to formalize following Bishop's work. There were several reasons for this.

- The need for heavy computation involving sums—the results proved so far had already required several computations and bounds, but these usually involved little more than properties of the absolute value and algebraic identities. As will be seen below, the situation turned out to be quite different when dealing with integrals.

- The need for technical lemmas, including very specific identities between sums, numerous results about proof irrelevance and formalizing fuzzy concepts like "sufficiently close approximation". Once again, all these issues have already appeared in this chapter, but at this stage they become much more prominent.

This section describes in detail the process of formalizing the definition of the integral (as presented in page 27). Special attention will be paid to the proof of one specific lemma, which was accidentally incorrect in the reference book, and which illustrates quite well the kind of technicalities needed at the level of formalization—as well as some examples of the kind of proof steps that don't seem likely to be automated in the near future.

Partitions

Partitions were introduced as Definition 2.4.18. Like totally bounded sets, they are an instance of a concept where finite sets play a fundamental role (although this time there is no existential quantifier lurking around), and therefore formalizing them poses similar questions to those addressed at the beginning of Section 5.2.

In this situation, however, it turns out that lists are not the best way to represent finite sets. There is a good reason for this: when dealing with totally bounded sets, all that was needed was to show that some point in the finite set satisfied some property; but partitions are finite sets all of whose elements must constantly be accessed, and this is much easier to do by means of functional application. A partition with n points is therefore defined as a function from the finite set $\{0,\ldots,n-1\}$ to $\mathbb{R}$ satisfying some extra properties, formalized as the following record type, where $\mathsf{a},\mathsf{b}:\mathbb{R}$ are implicit.

$$
\begin{array}{l}
\mathsf{Record\ Partition\ (Hab{:}(a \leq b))\ (lng{:}\mathbb{N})} : \mathbf{Set}{:=}\\
\quad \{\mathsf{Pts} \;\; :> \Pi_{\mathsf{i}:\mathbb{N}}.(\mathsf{i} \leq \mathsf{lng}) \to \mathbb{R};\\
\quad\;\; \mathsf{pr\_irr} : \;\; \forall_{\mathsf{i},\mathsf{j}:\mathbb{N}}.\mathsf{i} = \mathsf{j} \to \forall_{\mathsf{Hi},\mathsf{Hj}}.(\mathsf{Pts\ i\ Hi}) = (\mathsf{Pts\ j\ Hj});\\
\quad\;\; \mathsf{incr} \;\; : \;\; \forall_{\mathsf{i}:\mathbb{N}}.\forall_{\mathsf{H},\mathsf{H'}}.(\mathsf{Pts\ i\ H}) \leq (\mathsf{Pts\ (S\ i)\ H'});\\
\quad\;\; \mathsf{start} \; : \;\; \forall_{\mathsf{H}}.(\mathsf{Pts}\ 0\ \mathsf{H}) = \mathsf{a};\\
\quad\;\; \mathsf{finish} : \;\; \forall_{\mathsf{H}}.(\mathsf{Pts\ lng\ H}) = \mathsf{b}\}.
\end{array}
$$

The correspondence with Definition 2.4.18 should be straightforward. The functional part of the partition is the first component Pts of the record type; this is declared as a coercion, so that instead of the cumbersome (Pts P i Hi) the more intuitive (P i Hi) can be written for the i^{th} element of the partition (where Hi is an adequate proof term), corresponding to the informal P_i. The other four components of the record state the logical properties which must be satisfied by this function: it must not depend on its proof argument (pr_irr), be increasing (incr), begin at a (start) and end at b (finish).

Notice that, in principle, one could define a partition simply as a function with type $\mathbb{N}\to\mathbb{R}$ and require, besides the other properties, that this function map all natural numbers greater than n onto b. Although this yields nicer notation, allowing to dispense with the proof terms when referring to points of the partition, it is felt not to be as close to mathematical usage as the previous approach. Another disadvantage is that it requires many extra cases when doing proofs about properties of partitions; and in particular the construction of separated partitions (see below) becomes much more complicated.

The two first parameters of type of partitions of $[a, b]$ with n points are implicit; thus this type is written as (Partition Hab n). Also, everywhere a partition is taken as argument, all of its four parameters can be left implicit.

Interestingly, to define the mesh of a partition it is useful to go back to using lists. This is not totally surprising, as once again one faces a situation when one is interested only in one of the elements of a finite set rather than in arbitrarily accessing any of them. Thus, the definition of mesh proceeds in two stages: first, given a partition P with $n + 1$ points, an auxiliary list containing the values $P_{i+1} - P_i$, for $i < n$, is built; the mesh of P is defined to be the maximum of this list, which can be obtained by applying the maximum operator on lists defined earlier. (The minimum of this list will also be of interest for technical reasons.) The formalized versions of these definitions are not presented here, as they were constructed interactively: the need to first destruct the type constructor for Partition renders the corresponding Coq terms large and not easily readable.

Even partitions (see page 26) are straightforwardly defined by specifying their functional part[8]

$$\lambda_{\mathsf{i}:\mathbb{N}}.\lambda_{\mathsf{Hi}:(\mathsf{i}\leq\mathsf{n})}.\mathsf{a}+(\mathsf{nring}\ \mathsf{i})\times((\mathsf{b}-\mathsf{a})/(\mathsf{nring}\ \mathsf{n})).$$

and proving that this indeed defines a partition.

Throughout the remainder of this subsection, a, b, Hab : $(\mathsf{a} \leq \mathsf{b})$, n : $\mathbb{N}$ and a specific partition P : (Partition Hab n) will be assumed fixed; they will parameterize all definitions and lemmas, but will always be omitted. Furthermore, in most of the definitions and lemmas below P itself can be derived from other proof terms; whenever this is the case, it will be left as an implicit argument.

[8]For any ring, (nring i) is the injection of the natural number i in that ring, defined recursively by (nring 0) = 0 and (nring (S p)) = (nring p) + 1.

A partition Q is said to be a *refinement* of P iff P is a subsequence of Q.

$\mathsf{Definition\ Refinement\ (m{:}\mathbb{N})\ (Q{:}(Partition\ m)) :=}$
$\exists_{\mathsf{f}:\mathbb{N}\to\mathbb{N}}.\ \mathsf{(f\ 0){=}0} \wedge \mathsf{(f\ n){=}m} \wedge (\forall_{\mathsf{i,j}:\mathbb{N}}.\mathsf{i} < \mathsf{j} \to \mathsf{(f\ i)} < \mathsf{(f\ j)}) \wedge$
$\forall_{\mathsf{i}:\mathbb{N}}.\forall_{\mathsf{H}}.\ \exists_{\mathsf{H'}}.\mathsf{(P\ i\ H) = (Q\ (f\ i)\ H')}.$

Useful results about refinements of P are that their mesh is not greater than P's and that any two consecutive points of a refinement are between two consecutive points of P.

$\mathsf{Lemma\ Mesh\_leEq} : \forall_{\mathsf{m}:\mathbb{N}}.\forall_{\mathsf{Q:(Partition\ m)}}.$
$\mathsf{(Refinement\ Q)} \to \mathsf{(Mesh\ Q)} \leq \mathsf{(Mesh\ P)}.$

$\mathsf{Lemma\ Refinement\_prop} : \forall_{\mathsf{m}:\mathbb{N}}.\forall_{\mathsf{Q:(Partition\ m)}}.\mathsf{(Refinement\ Q)} \to$
$\forall_{\mathsf{i}:\mathbb{N}}.\forall_{\mathsf{Hi,Hi'}}.\exists_{\mathsf{j}:\mathbb{N}}.\exists_{\mathsf{Hj,Hj'}}.\mathsf{(P\ j\ Hj)} \leq \mathsf{(Q\ i\ Hi)} \wedge \mathsf{(Q\ (S\ i)\ Hi')} \leq \mathsf{(P\ (S\ j)\ Hj')}.$

Given a continuous function f, Bishop loosely defines $S(f, P)$ as an arbitrary sum of the type

$$\sum_{i=0}^{n-1} f(x_i)(P_{i+1} - P_i),$$

where $x_i \in [P_i, P_{i+1}]$. This informal definition cannot be directly phrased in Coq terms (or at least, not in such a way that it can easily be used later); its formalization therefore proceeds in two steps: first, one defines a *choice of points respecting* P to be a set $\{x_0, \ldots, x_{n-1}\}$ of points such that, for every i, $x_i \in [P_i, P_{i+1}]$. Once again, x will be formalized as a function type rather than as a list for similar reasons as those presented above.

$\mathsf{Definition\ Points\_in\_Partition\ (x{:}\Pi_{i:\mathbb{N}}.(i < n) \to \mathbb{R}) :=}$
$\forall_{\mathsf{i}:\mathbb{N}}.\forall_{\mathsf{H}}.\mathsf{([(P\ i\ H'),(P\ (S\ i)\ H'')]\ (x\ i\ H))}.$

Here, $\mathsf{H'}$ and $\mathsf{H''}$ are proof terms built from H.

Given a function F defined on $[a, b]$ and such a choice of points x, a *specific* instance of one of Bishop's $S(f, P)$ numbers can be defined.

$\mathsf{Definition\ Partition\_Sum}$
$\mathsf{(x{:}\Pi_{i:\mathbb{N}}.(i < n) \to \mathbb{R})\ (F{:}PartIR)\ (H{:}(Points\_in\_Partition\ x)) :=}$
$\mathsf{(Sumx\ \lambda_{i:\mathbb{N}}.\lambda_{Hi}.(F\ (x\ i\ Hi)\ Hi_1)) \times ((P\ (S\ i)\ Hi_2) - (P\ i\ Hi_3))}.$

The omitted proof terms are constructed from Hi.

In the previous definition, x can be obtained from the proof term H, and will therefore be left implicit.

From this point onwards, the informal statement "for all $S(f, P)$..." should always be read as corresponding to "for all $\mathsf{x} : \Pi_{\mathsf{i}:\mathbb{N}}.(\mathsf{i} < \mathsf{n}) \to \mathbb{R}$ and $\mathsf{H} : \mathsf{(Points\_in\_Partition\ x)}$, $\mathsf{(Partition\_Sum\ F\ H)}$ satisfies...".

Just as given any natural number m there is a canonical partition of any compact interval (the even partition with m points), there is a canonical choice of points respecting any given partition: simply choose the left end-point of each subinterval or, equivalently, take the first $n-1$ points of the partition.

Definition Partition_imp_points (m:$\mathbb{N}$) (Q:(Partition m)) : $\Pi_{\mathsf{i}:\mathbb{N}}$.(i < m)$\to\mathbb{R}$:=
 $\lambda_{\mathsf{i}:\mathbb{N}}.\lambda_{\mathsf{H}}$.i (Q i H')

Lemma Partition_imp_points_wd : $\forall_{\mathsf{m}:\mathbb{N}}.\forall_{\mathsf{Q}:(\mathsf{Partition\ m})}$.
 (Points_in_Partition Q (Partition_imp_points Q)).

The Integral

During this subsection, the interval $[a, b]$ (together with the relevant proof term Hab : (a $\leq$ b)) will be fixed; f is a continuous function with modulus of continuity ω in $[a, b]$. Together with a, b and Hab, both F and the proof contF of its continuity will be left implicit throughout. Afterwards, only the last of these will have to be explicitly given, as it contains enough information to construct the other four.

Coupling the last definitions above with the definition of even partition, the following sequence (Bishop's $\{S(f, a, b, n)\}_{n=1}^{\infty}$) is defined[9].

Definition Even_Partition_Sum (m:$\mathbb{N}$) :=
 (Partition_Sum F (Partition_imp_Points_wd (Even_Partition m))).

Definition integral_seq : $\mathbb{N}\to\mathbb{R}$:= $\lambda_{\mathsf{n}:\mathbb{N}}$.(Even_Partition_Sum (S n)).

In order to prove that the sequence of sums previously defined is a Cauchy sequence, the two following lemmas are needed.

LEMMA 5.4.1 If P and Q are partitions of $[a, b]$ with $\text{mesh}(P) \leq \omega(\varepsilon)$ and Q is a refinement of P, then, for any sums $S(f, P)$ and $S(f, Q)$, the following inequality holds:

$$|S(f, P) - S(f, Q)| \leq \varepsilon(b - a).$$

LEMMA 5.4.2 If P and R are partitions such that $\text{mesh}(P) \leq \omega(\varepsilon)$ and $\text{mesh}(R) \leq \omega(\varepsilon')$, and if there exists a partition Q which is simultaneously a refinement of P and of R, then for any sums $S(f, P)$ and $S(f, R)$ the following inequality holds:

$$|S(f, P) - S(f, R)| \leq (\varepsilon + \varepsilon')(b - a).$$

[9]See the remark on implicit arguments on page 115.

The proof of the first result presents no problems other than technical issues. It hangs mainly on the following fact: from the definition of refinement, given i it is possible to find j and j' such that $P_i = Q_j$ and $P_{i+1} = Q_{j'}$; this allows $S(f,P)$ to be written in terms of points of Q, and use of the modulus of continuity establishes the result. However, heavy manipulation of double sums is required which involves a lot more than just trivial computation: intuitively obvious results, like how to exchange the order of summation, are extremely non trivial to formalize.

The formalized counterpart of this lemma, where d is defined to be $\omega(\varepsilon)$ and fP and fQ are adequate choices of points (with corresponding proof terms HfP and HfQ), reads as follows.

$\mathsf{first\_refinement\_lemma} : \forall_{\mathsf{m,n}:\mathbb{N}}.\forall_{\mathsf{P:(Partition\ n)}}.\forall_{\mathsf{Q:(Partition\ m)}}.$
$\quad (\mathsf{Refinement\ P\ Q}) \to \forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \to (\mathsf{Mesh\ P}) \leq \mathsf{d} \to \forall_{\mathsf{fP,fQ}}.\forall_{\mathsf{HfP,HfQ}}.$
$\qquad (\mathsf{AbsIR\ (Partition\_Sum\ HfP\ F)}-(\mathsf{Partition\_Sum\ HfQ\ F})) \leq \mathsf{e}\times(\mathsf{b}-\mathsf{a}).$

The proof of the second lemma is quite simpler, as it just amounts to two applications of the first one: $|S(f,P) - S(f,R)|$ is obviously equal to $|(S(f,P) - S(f,Q)) + (S(f,Q) - S(f,R))|$, and the desired result follows from the triangle inequality. Similar notations as previously apply to its formalization.

$\mathsf{second\_refinement\_lemma} : \forall_{\mathsf{j,n,k}:\mathbb{N}}.\forall_{\mathsf{P:(Partition\ j)}}.\forall_{\mathsf{Q:(Partition\ n)}}.\forall_{\mathsf{R:(Partition\ k)}}.$
$\quad (\mathsf{Refinement\ P\ Q}) \to (\mathsf{Refinement\ R\ Q}) \to \forall_{\mathsf{e,e'}:\mathbb{R}}.(0 < \mathsf{e}) \to (0 < \mathsf{e'}) \to$
$\qquad (\mathsf{Mesh\ P}) \leq \mathsf{d} \to (\mathsf{Mesh\ R}) \leq \mathsf{d'} \to \forall_{\mathsf{fP,fR}}.\forall_{\mathsf{HfP,HfR}}.$
$\qquad\quad (\mathsf{AbsIR\ (Partition\_Sum\ HfP\ F)}-(\mathsf{Partition\_Sum\ HfR\ F})) \leq$
$\qquad\qquad \mathsf{e}\times(\mathsf{b}-\mathsf{a})+\mathsf{e'}\times(\mathsf{b}-\mathsf{a}).$

It can now be proved that integral_seq converges: since even partitions always have a common refinement (given even partitions of $[a,b]$ with n and m points, the even partition of the same interval with $m \times n$ points is such a refinement), the Cauchy property for integral_seq follows trivially from Lemma 5.4.2. The integral of f in $[a,b]$ is defined as its limit.

Lemma Cauchy_integral_Seq : (Cauchy_prop integral_seq).

Definition integral := (Lim integral_seq).

The full type of integral is

$$\Pi_{\mathsf{a,b}:\mathbb{R}}.\Pi_{\mathsf{Hab}:(\mathsf{a}\leq\mathsf{b})}.\Pi_{\mathsf{F:PartIR}}.(\mathsf{Continuous\_I\ Hab\ F}) \to \mathbb{R},$$

and the first four arguments will be made implicit[10].

[10] This makes the notation somewhat strange-looking, since one will write (Integral H) where H is a proof of continuity, and both the function and the domain of integration are not mentioned. However, it also makes the notation much lighter, which is very helpful after one gets used to it.

Linearity and monotonicity of the integral are proved by unfolding the definition of integral and appealing to the corresponding properties of limits of Cauchy sequences and of sums; also relevant for the sequel are the facts that $\int_a^b 0 = 0$ and $\int_a^b 1 = b - a$. The absolute value of the integral of f in $[a, b]$ is bounded by $\|f\|_{[a,b]} \times (b - a)$.

The formalization of all these results is in C-CoRN.

Furthermore, the integral is strongly extensional in all its arguments: if $\int_a^b f \# \int_a^b g$ then an x can be found such that $f(x) \# g(x)$, and if $\int_a^b f \# \int_c^d f$ then either $a \# c$ or $b \# d$.

$\mathsf{Lemma\ integral\_strext} : \forall_{\mathsf{a,b}:\mathbb{R}}.\forall_{\mathsf{Hab:(a\leq b)}}.\forall_{\mathsf{F,G:PartIR}}.\forall_{\mathsf{contF:(Continuous\_I\ Hab\ F)}}.$
$\quad\forall_{\mathsf{contG:(Continuous\_I\ Hab\ G)}}.(\mathsf{integral\ contF}) \# (\mathsf{integral\ contG}) \rightarrow$
$\qquad\exists_{\mathsf{x}:\mathbb{R}}.([\mathsf{a,b}]\ \mathsf{x}) \wedge \forall_{\mathsf{Hx,Hx'}}.(\mathsf{F\ x\ Hx}) \# (\mathsf{G\ x\ Hx'}).$

$\mathsf{Lemma\ integral\_strext'} : \forall_{\mathsf{a,b,c,d}:\mathbb{R}}.\forall_{\mathsf{Hab:(a\leq b)}}.\forall_{\mathsf{Hcd:(c\leq d)}}.\forall_{\mathsf{F:PartIR}}.$
$\quad\forall_{\mathsf{contFab:(Continuous\_I\ Hab\ F)}}.\forall_{\mathsf{contFcd:(Continuous\_I\ Hcd\ F)}}.$
$\qquad(\mathsf{integral\ contFab}) \# (\mathsf{integral\ contFcd}) \rightarrow (\mathsf{a} \# \mathsf{c}) \vee (\mathsf{b} \# \mathsf{d}).$

The next step is to show that

$$\int_a^b f = \int_a^c f + \int_c^b f \tag{5.1}$$

whenever $a \leq c \leq b$. The motivation for this is as follows: removing the order restrictions on a, b and c, a general relation is obtained which can be used as a *definition* of integral in the general case.

However, the proof of this is both far from trivial, and significantly more complicated than the informal presentation of Bishop; the next paragraph is dedicated to its presentation.

The Generalized Integral

In order to show that Equation (5.1) holds, the following theorem is needed.

THEOREM 5.4.3 Let f be a continuous function on a compact interval $[a, b]$ with modulus of continuity ω. If P is any partition of $[a, b]$, if $\varepsilon > 0$, and if $\mathrm{mesh}(P) \leq \omega(\varepsilon)$, then, for any sum $S(f, P)$ the following relation holds:

$$\left| S(f, P) - \int_a^b f \right| \leq \varepsilon(b - a). \tag{5.2}$$

At first glance, the desired result can easily be obtained by applying Lemma 5.4.2 to $S(f, P)$ and some ($\mathsf{integral\_seq\ n}$) with n large enough, using properties of inequalities and limits. All that remains is to prove that the two partitions in question share a common refinement, which was stated without proof in [10].

Unfortunately, though classically this is a trivial statement, constructively it does not hold[11]! The reason for that is that in a partition points must be ordered, and the order relation on the real numbers is not decidable.

Notice that Theorem 5.4.3 holds if only even partitions are considered, since these always have a common refinement. However, the full power of the theorem is needed to prove Equation (5.1): in this situation, it is necessary to define a partition of $[a, b]$ given partitions of $[a, c]$ and $[c, b]$, and if the ratio between the lengths of these intervals is irrational even partitions will not suffice.

In [11] this problem was solved in the following way: first, two partitions P and Q with respectively n and m points are defined to be *separated* iff for all i and j in the appropriate ranges $P_i < P_{i+1}$ and $Q_j < Q_{j+1}$; furthermore, if $0 < i < n$ and $0 < j < m$ then $P_i \# Q_j$.

Two separated partitions always have a common refinement. This is a consequence of co-transitivity of the order: given i and j, it can be decided whether $Q_j < P_i$ or $P_i < Q_{j+1}$, which allows the points of both partitions to be ordered. Formalizing this, though not complex, is still a very long and tedious process, requiring several proofs by induction and formalizing many intuitive, yet non trivial, auxiliary results.

The general form of theorem 5.4.3 is then proved by taking close enough approximations of P and R that are separated.

Of course, this is enough for the informal presentation, but as was already mentioned earlier, the concept of "close enough approximations" is all but easy to formalize. In other words, to prove this result in Coq it is necessary to actually construct these approximations. The next few paragraphs explain how this can be done.

At this stage of the formalization, one really becomes aware of the fundamental difference between informal (no matter how precise) and formal mathematics. In [11], the authors never need to go to this level of detail, which is fundamental for the formalization of this proof.

The notion of separation is defined in two steps. A partition P is said to be (simply) separated iff $P_i < P_{i+1}$ for all i.

Definition _Separated (n:$\mathbb{N}$) (P:(Partition n)) :=
  $\forall_{\mathsf{i}:\mathbb{N}}.\forall_{\mathsf{Hi},\mathsf{Hi}'}$.(P i Hi) < (P (S i) Hi').

[11]Actually, this is the only essential mistake in Bishop's book that was found while formalized; and it had been (almost) corrected in the second edition [11], as discussed below.

Two partitions P and Q are said to be (mutually) separated iff each of them is separated and if $P_i \# Q_j$ whenever $0 < i < n$ and $0 < j < m$.

Definition Separated (n m:$\mathbb{N}$) (P:(Partition n)) (Q:(Partition m)) :=
 (_Separated P) $\wedge$ (_Separated Q) $\wedge$
 $\forall_{\mathsf{i,j:\mathbb{N}}}.(0 < \mathsf{i}) \to (0 < \mathsf{j}) \to (\mathsf{i} < \mathsf{n}) \to (\mathsf{j} < \mathsf{m}) \to \forall_{\mathsf{Hi,Hj}}.(\mathsf{P\ i\ Hi}) \# (\mathsf{Q\ j\ Hj})$.

The construction of separated approximations of two partitions is then done in two steps. First, given a partition P and positive real numbers α and ξ, an algorithm is developed to get a separated partition P' with the following properties:

- $\mathrm{mesh}(P') \leq \mathrm{mesh}(P) + \xi$;
- for every sum $S(f, P)$ there is a corresponding sum $S(f, P')$ such that $|S(f, P) - S(f, P')| < \alpha$.

To do this, one takes δ to be $\min(\xi, \frac{\alpha}{n \cdot M})$, where n is the number of points in P and M is the norm of f in $[a, b]$. It is clear that δ is positive, which means that for every real number x either $x > 0$ or $x < \frac{\delta}{2}$. This allows one to recursively define the following sequence of partitions:

- $P^0 = P$;
- P^{i+1} is obtained from P in the following way: for every pair P^i_j, P^i_{j+1} of consecutive points in P^i, test whether $P^i_{j+1} - P^i_j > 0$ or $P^i_{j+1} - P^i_j < \frac{\delta}{2}$. If there is a j for which the second case holds, choose the least such j and define $P^{i+1}_m = P^i_m$ for $m \leq j$ and $P^{i+1}_m = P^i_{m-1}$ for $m > j$ (that is, obtain P^{i+1} by removing the $(j+1)^{\text{th}}$ point in P^i)[12]; else $P^{i+1} = P^i$.

This construction always gets to a fixed point; this is a partition P' satisfying both desired conditions (the first is trivial; for the second, take any choice of points respecting P and simply remove the points that were removed in the construction of P'; continuity of f yields the desired inequality).

Now, given P and R, two separated partitions P' and R' can be found by the above construction; then, points in P' can be shifted by a similar (though even trickier) construction to get a partition P'' which is also separated from R' and for which the previous two properties hold.

At this point, there turns out to be yet another small detail which has to be corrected in the statement of the theorem. It was assumed that $\mathrm{mesh}(P) \leq \omega(\varepsilon)$; however, although the approximations P' and P'' can have a mesh as

[12]Some care needs to be taken if $j + 1$ is the length of P^i, but that level of detail is unnecessary at this stage.

close to that of P as desired, this mesh cannot actually be guaranteed to be equal to that of P (to see this, consider the case when P is an even partition; then any shifting of its points will necessarily increase its mesh).

This invalidates the reasoning through approximations, as if $\text{mesh}(P') > \omega(\varepsilon)$ no bound for the sum can any longer be established. Still, this problem can be solved simply by requiring, in the statement of the theorem, that $\text{mesh}(P) < \omega(\varepsilon)$. Now the approximations can be built in such a way that this inequality still holds (by taking $\alpha = \frac{1}{2}[\omega(\varepsilon) - \text{mesh}(P)]$), and Lemma 5.4.2 can be applied. This allows the following generalization of that same result, where the common refinement has been removed.

$\mathsf{refinement\_lemma} : \forall_{\mathsf{n,m}:\mathbb{N}}.\forall_{\mathsf{P}:(\mathsf{Partition\ n})}.\forall_{\mathsf{R}:(\mathsf{Partition\ m})}.$
$\quad\forall_{\mathsf{e,e'}:\mathbb{R}}.(0 < \mathsf{e}) \to (0 < \mathsf{e'}) \to$
$\qquad(\mathsf{Mesh\ P}) \leq \mathsf{d} \to (\mathsf{Mesh\ R}) \leq \mathsf{d'} \to \forall_{\mathsf{fP,fR}}.\forall_{\mathsf{HfP,HfR}}.$
$\qquad\quad(\mathsf{AbsIR\ (Partition\_Sum\ HfP\ F) {-} (Partition\_Sum\ HfR\ F)}) \leq$
$\qquad\qquad\mathsf{e} \times (\mathsf{b} - \mathsf{a}) + \mathsf{e'} \times (\mathsf{b} - \mathsf{a}).$

The formalized version of Theorem 5.4.3 then reads as follows.

$\mathsf{partition\_Sum\_conv\_integral} : \forall_{\mathsf{n}:\mathbb{N}}.\forall_{\mathsf{P}:(\mathsf{Partition\ n})}.\forall_{\mathsf{e}:\mathbb{R}}.(0 < \mathsf{e}) \to$
$\quad((\mathsf{Mesh\ P}) < \mathsf{d}) \to \forall_{\mathsf{fP}:\Pi_{\mathsf{i}:\mathbb{N}}.(\mathsf{i}<\mathsf{n}) \to \mathbb{R}}.(\mathsf{Points\_in\_Partition\ P\ fP}) \to$
$\qquad(\mathsf{AbsIR\ (Partition\_Sum\ fP\ F) {-} (integral\ F)}) \leq \mathsf{e} \times \mathsf{b} - \mathsf{a}.$

Equation (5.1) can now be shown to hold using properties of limits, closely following Bishop's proof, and appealing to Equation (5.2). As discussed above, this cannot in general be done only using even partitions, and therefore the full power of Theorem 5.4.3 is needed.

It has been suggested that Theorem 5.4.3 could have been avoided in its general form if one restricted one's attention to partitions where all points are rational and used approximations to deal with the general case. In practice, though, this is at least as difficult as proving this general form of the theorem, since it also requires specification of notions such as "close enough (rational) approximation"; and there is another, trickier, issue involved: if (one of) the endpoints of the interval in question are (is) irrational, then partitions of that interval will be approximated by partitions of a *different* interval, and it becomes much more complex to even state the approximation results that should hold.

In the general case, the integral is defined by the relation

$$\int_a^b f = \int_{\min(a,b)}^b f - \int_{\min(a,b)}^a f,$$

where it is assumed that f is continuous in $[\min(a,b), \max(a,b)]$. In the formalization, $\mathsf{Hab'} : (\mathsf{Min\ a\ b}) \leq \mathsf{a}$ and $\mathsf{Hab''} : (\mathsf{Min\ a\ b}) \leq \mathsf{b}$ are defined terms.

Lemma Integral_inc1 : (Continuous_I Hab' F).

Lemma Integral_inc2 : (Continuous_I Hab" F).

Definition Integral := (integral Integral_inc2)−(integral Integral_inc1).

Notice that the endpoints of the intervals as well as the function being integrated are again implicit.

The arguments to Integral are slightly different than those to integral: two real numbers a and b, a proof of $\min(a, b) \leq \max(a, b)$, a partial function f and a proof that f is continuous in the interval $[\min(a, b), \max(a, b)]$. As before, all arguments but the last are implicit.

This definition is consistent with the previous, as both coincide when $a \leq b$; and Equation (5.1) can now be seen to hold for arbitrary a, b and c provided that f is continuous on all necessary intervals.

It is easy to prove that this new integral inherits all the properties of the old one.

As a final step, a primitive operator Fprim is defined. This operator takes as arguments a function f, an interval I, a point a of I and a proof that f is continuous on I and yields the primitive of f defined by $\lambda_{\mathsf{x}}. \int_a^x f$.

The definition goes in three steps. First, it is shown that if f is a continuous function on I, then it is continuous on every subinterval of I of the form $[\min(a, x), \max(a, x)]$, where x is arbitrary (this is an almost trivial fact; the relevant step here is that it is folded into a lemma, and can thus be used in further definitions). Here, (Min_leEq_Max a x) is a canonical proof of $\min(a, x) \leq \max(a, x)$.

Lemma prim_lemma : $\forall_{\mathsf{x}:\mathbb{R}}$.(I x)$\rightarrow$(Continuous_I (Min_leEq_Max a x) F).

Next, the operation $\lambda_{\mathsf{x}}. \int_a^x f$ is shown to be strongly extensional.

Lemma Fprim_strext : $\forall_{\mathsf{x,y}:\mathbb{R}}.\forall_{\mathsf{Hx,Hy}}$.
(Integral (prim_lemma x Hx)) # (Integral (prim_lemma y Hy))$\rightarrow$(x # y).

And finally, a partial function is defined using this last fact.

Definition Fprim := (Build_PartFunct $\mathbb{R}$ I (iprop_wd I)
$\lambda_{\mathsf{x}:\mathbb{R}}.\lambda_{\mathsf{Hx:(I\ x)}}$.(Integral (prim_lemma x Hx)) Fprim_strext).

This is a continuous function.

Now Theorem 2.4.19 and Corollary 2.4.20 can easily be proved.

Theorem FTC1 : $\forall_{\mathsf{J:interval}}.\forall_{\mathsf{F:PartIR}}.\forall_{\mathsf{contF:(Continuous\ J\ F)}}.\forall_{\mathsf{x_0}:\mathbb{R}}.\forall_{\mathsf{Hx_0:(J\ x_0)}}$.
$\forall_{\mathsf{pJ:(proper\ J)}}$.(Derivative pJ (Fprim contF $\mathsf{x_0}$ $\mathsf{Hx_0}$) F)

Theorem FTC2 : $\forall_{\mathsf{J:interval}}.\forall_{\mathsf{F:PartIR}}.\forall_{\mathsf{contF:(Continuous\ J\ F)}}.\forall_{\mathsf{x_0}:\mathbb{R}}.\forall_{\mathsf{Hx_0:(J\ x_0)}}$.

$\forall_{\mathsf{pJ:(proper\ J)}}.\forall_{\mathsf{G_0:PartIR}}.(\mathsf{Derivative\ pJ\ G_0\ F}) \rightarrow$
$\exists_{\mathsf{c}:\mathbb{R}}.(\mathsf{Feq\ J\ (Fprim\ contF\ x_0\ Hx_0)} - \mathsf{G_0}\ \lambda \mathsf{x.c})$

Theorem FTC3 : $\forall_{\mathsf{J:interval}}.\forall_{\mathsf{F:PartIR}}.\forall_{\mathsf{contF:(Continuous\ J\ F)}}.\forall_{\mathsf{x_0}:\mathbb{R}}.\forall_{\mathsf{Hx_0:(J\ x_0)}}.$
$\forall_{\mathsf{pJ:(proper\ J)}}.\forall_{\mathsf{G_0:PartIR}}.(\mathsf{Derivative\ pJ\ G_0\ F}) \rightarrow \forall_{\mathsf{a,b}:\mathbb{R}}.(\mathsf{J\ a}) \rightarrow (\mathsf{J\ b}) \rightarrow$
$\forall_{\mathsf{H:(Continuous\_I\ (Min\_leEq\_Max\ a\ b)\ F)}}.$
$(\mathsf{Integral\ H}) = (\mathsf{G_0\ b\ Hb'}) - (\mathsf{G_0\ a\ Ha'}).$

where Ha' and Hb' in the last theorem are proof terms defined from H.

The proof of these results follows Bishop's without any detour.

5.5 Transcendental Functions

The last section of [10, Chapter 2] deals with the so-called elementary transcendental functions: the exponential, the trigonometric functions (sine and cosine) and their inverses.

The process of defining these functions, as described in the last section of Section 2.4, follows a common strategy. Exponential, sine and cosine are defined as the (unique) solution to some differential equation motivated by the analysis of their expected properties. It is then proved that they do satisfy these properties; and their inverses are defined as indefinite integrals motivated by the chain rule for differentiation.

Beyond Taylor's Theorem

The theory developed at this stage is not powerful enough to allow the formalization of this section to begin immediately. Rather, a number of results already proved have to be strengthened or extended. These will now be described in general terms. Since their formalization poses no new problems, no actual Coq code will be presented.

The first step is to prove the corollaries to the Fundamental Theorem of Calculus presented previously as Corollaries 2.4.21 (integrals and limits commute) and 2.4.22 (derivatives and limits commute).

Corollary 2.4.21 is stated and proved first for compact intervals, then for arbitrary intervals. The proof of the first case follows Bishop's closely; the generalization is quite straightforward.

As for Corollary 2.4.22, it can be proved directly in the general case, as the proof in [10] does not make any use of the properties of compact intervals: it relies only on Theorem 2.4.19 and Corollary 2.4.21, both of which are valid in general.

The formalized versions of these results are direct translations of the original statements.

On another level, an operator Taylor_Series is defined, assigning to any infinitely differentiable function on an interval its Taylor series around a point of that interval. Corollary 2.4.17 (the Taylor series converges to the original function) can then be stated and proved.

Coupling the two last results yields a very nice and powerful result, which states that two functions satisfying the hypotheses of Corollary 2.4.17 whose derivatives of order 1 and greater all coincide must also be identical (see Corollary 2.4.23). This will turn out to be the key result in this section.

Also, it is proved that two functions which coincide on one point and have the same derivative on a given interval are equal on that interval. (This is a straightforward consequence of the Fundamental Theorem of Calculus, Theorem 2.4.19.)

Finally, the formula for the derivative of a function defined by a power series is proved: the sequence of derivatives of (FPowerSeries′ x_0 a) on the real line is the sequence (FPowerSeries′ x_0 $\lambda_{n:\mathbb{N}}$.(a (S n))).

Exponential and Logarithm

The exponential function is defined by applying the FPowerSeries′ constructor[13] to the constant sequence with value 1. This defines a function over the whole real line; therefore, every real number is in its domain.

Definition Exp_ps := (FPowerSeries' 0 $\lambda_{n:\mathbb{N}}.1$).

Lemma Exp_conv : (fun_series_convergent_IR realline Exp_ps).

Definition Expon := (Fun_Series_Sum Exp_conv).

Lemma Exp_domain : $\forall_{x:\mathbb{R}}$.(dom Expon x).

Using this, a total function Exp : (CSetoid_un_op $\mathbb{R}$) can be defined.

This last definition is extremely useful, since it allows one to write (Exp x) for $\exp(x)$ without further ado[14].

Since Exp is defined in terms of a partial function, it is trivially (strongly) extensional. Furthermore, from its definition it is straightforward to prove that $\exp(0) = 1$ and $\exp(1) = e$, where e in the last expression refers to the constant defined on page 98. Also the exponential is its own derivative.

[13] See definition on page 106.

[14] The notation $\exp(x)$ will be used in the informal discussion instead of the more usual e^x to emphasize the fact that (Exp x) represents the application of the exponential function to the real number x rather than the real number e raised to the power x.

Lemma Exp_strext : $\forall_{x,y:\mathbb{R}}$.(Exp x) # (Exp y) $\rightarrow$ (x # y).

Lemma Exp_zero : (Exp 0) = 1.

Lemma Exp_one : (Exp 1) = E.

Lemma Derivative_Exp : $\forall_{\mathsf{H}:(\mathsf{proper\ realline})}$.(Derivative H Expon Expon).

The exponential is the only function that is its own derivative and has value 1 at the origin. The proof, which is skipped in [10], is a quite straightforward consequence of the lemmas on Taylor series. Here, I : $\top$ is the canonical proof of the true proposition.

Lemma Exp_unique : $\forall_{\mathsf{F}:\mathsf{PartIR}}$.(Derivative (I := realline) I F F) $\rightarrow$
($\forall_{\mathsf{H1}}$.(F 0 H1) = 1) $\rightarrow$ (Feq realline Expon F).

A direct consequence is the relation $\exp(x+y) = \exp(x) \times \exp(y)$, which is proved closely following Bishop.

Bishop ends his analysis of the exponential function by proving that $\exp(-x) = \exp(x)^{-1}$ and thereby concluding that exp is always positive.

In order to study the usability of this formalization, the study of the properties of the exponential was taken quite a bit further. The results in the C-CoRN library include not only the above presented rules for the exponential of the addition and subtraction of two real numbers, but also that the exponential is monotonous and injective. These were all trivial to state and prove.

The next step is to build the logarithm function. Bishop defines it to be the indefinite integral of the function $\lambda_{\mathsf{x}}.\frac{1}{x}$ from 1. To do this, one first needs to prove that the latter is a continuous function on the positive reals, which is completely trivial.

Lemma log_defn_lemma : (Continuous (openl 0) 1/FId).

Definition Logarithm := (Fprim log_defn_lemma 1 (pos_one $\mathbb{R}$)).

Definition Log := $\lambda_{\mathsf{x}:\mathbb{R}}.\lambda_{\mathsf{Hx}:(0<\mathsf{x})}$.(Logarithm x Hx).

Notice the introduction of the notation (Log x Hx); the main reason for doing this is to separate clearly in the statements (and proofs) of lemmas the functional properties (which hold for Logarithm) from the algebraic properties (referring always to a specific (Log x Hx)); this is similar to what happens with the exponential function. The types of these terms reflect this: Logarithm has type PartIR, while Log has type $\Pi_{\mathsf{x}:\mathbb{R}}.(0 < \mathsf{x}) \rightarrow \mathbb{R}$.

From its definition it is also trivial to prove that the logarithm is (strongly) extensional; that its derivative is (of course) the function $\lambda_{\mathsf{x}}.\frac{1}{x}$; and that $\log(1) = 0$. All these proofs can be done in Coq in less than five lines.

Lemma Log_strext : $\forall_{x,y:\mathbb{R}}.\forall_{Hx,Hy}.$(Log x Hx) # (Log y Hy) $\rightarrow$ (x # y).

Lemma Derivative_Log : $\forall_{H:(\text{proper (openl 0)})}.$(Derivative H Logarithm 1/FId).

Lemma Log_one : $\forall_{H}.$(Log 1 H) = 0.

Bishop's proof that $\log(xy) = \log(x) + \log(y)$, appealing to the chain rule for the derivative and to the Fundamental Theorem of Calculus, also translates directly.

The logarithm function is then proved to be the inverse of the exponential, in the sense of the following two lemmas.

Lemma Log_Exp_inv : (Feq realline Logarithm $\circ$ Expon FId).

Lemma Exp_Log_inv : (Feq (openl 0) Expon $\circ$ Logarithm FId).

Once again, both proofs are direct transcriptions of Bishop's.

As for the exponential, a more extensive analysis of the properties of the logarithm function is done. Thus, the results in the C-CoRN library presently include not only the above, but also the rule $\log(x/y) = \log(x) - \log(y)$ (assuming all three logarithms are defined) and several order properties, of which a sample is presented.

Lemma Log_E : $\forall_{He}.$(Log E He) = 1.

Lemma Log_div : $\forall_{x,y:\mathbb{R}}.\forall_{Hx,Hy,Hy',Hxy}.$
(Log (x/y//Hy') Hxy) = (Log x Hx) $-$ (Log y Hy).

Lemma Log_cancel_less : $\forall_{x,y:\mathbb{R}}.\forall_{Hx,Hy}.$(Log x Hx) < (Log y Hy) $\rightarrow$ (x < y).

Lemma Log_resp_less : $\forall_{x,y:\mathbb{R}}.\forall_{Hx,Hy}.$(x < y) $\rightarrow$ (Log x Hx) < (Log y Hy).

Lemma Log_less_Zero : $\forall_{x:\mathbb{R}}.\forall_{Hx}.$(x < 1) $\rightarrow$ (Log x Hx) < 0.

Lemma Zero_less_Log : $\forall_{x:\mathbb{R}}.\forall_{Hx}.$(1 < x) $\rightarrow$ 0 < (Log x Hx).

The formalized versions of these results are clearly very similar to their informal counterparts. This is felt to be the best evidence of the usability of this formalization.

Powers with Real Exponents

In order to test the usability of the formalization so far, it was decided to extend it with the generalization of exponentiation of positive numbers to an arbitrary real power and prove the main properties of this operation.

In mathematics, this operation is motivated by the relation $x^y = \exp(y \times \log(x))$, valid whenever x is positive and y an integer. This relation is then used as a definition for all real y, yielding an extension of the usual exponentiation function.

Definition power $:= \lambda_{x,y:\mathbb{R}}.\lambda_{Hx:0<x}.(\mathsf{Exp}\ y\times(\mathsf{Log}\ x\ Hx))$.

Following the Coq notation closely, the term (power x y Hx) will be written down as $(x \uparrow y//Hx)$.

The power operation is (strongly) extensional; it commutes with the algebraic operations in the usual way.

Lemma power_strext : $\forall_{x,x',y,y':\mathbb{R}}.\forall_{Hx,Hx'}.$
$(x \uparrow y//Hx) \# (x' \uparrow y'//Hx') \to (x \# x') \vee (y \# y')$.

Lemma power_plus : $\forall_{x,y,z:\mathbb{R}}.\forall_{Hx}.$
$(x \uparrow y + z//Hx) = (x \uparrow y//Hx) \times (x \uparrow z//Hx)$.

Lemma power_minus : $\forall_{x,y,z:\mathbb{R}}.\forall_{Hx,Hxz}.$
$(x \uparrow y - z//Hx) = (x \uparrow y//Hx)/(x \uparrow z//Hx)//Hxz$.

Lemma mult_power : $\forall_{x,y,z:\mathbb{R}}.\forall_{Hx,Hy,Hxy}.$
$((x \times y) \uparrow z//Hxy) = (x \uparrow z//Hx) \times (y \uparrow z//Hy)$.

Lemma div_power : $\forall_{x,y,z:\mathbb{R}}.\forall_{Hx,Hy,Hy',Hxy,Hyz}.$
$((x/y//Hy') \uparrow z//Hxy) = (x \uparrow z//Hx)/(y \uparrow z//Hy)//Hyz$.

Lemma power_mult : $\forall_{x,y,z:\mathbb{R}}.\forall_{Hx,Hxy}.$
$(x \uparrow (y \times z)//Hx) = ((x \uparrow y//Hx) \uparrow z//Hxy)$.

The presence of the proof terms, although making the notation a bit heavier than one is accustomed to, guarantees that all terms in these lemmas are defined whenever any of them is applied.

A next step is to prove that this operation extends the exponentiation with natural and integer exponents; generalizes n^{th} roots; and coincides with Exp when the basis is E.

Lemma power_nat : $\forall_{x:\mathbb{R}}.\forall_{n:\mathbb{N}}.\forall_{Hx}.(x \uparrow (\mathsf{nring}\ n)//Hx) = x^n$.

Lemma power_zero : $\forall_{x:\mathbb{R}}.\forall_{Hx}.(x \uparrow 0//Hx) = 1$.

Lemma power_one : $\forall_{x:\mathbb{R}}.\forall_{Hx}.(x \uparrow 1//Hx) = x$.

Lemma power_int : $\forall_{x:\mathbb{R}}.\forall_{z:\mathbb{Z}}.\forall_{Hx,Hx'}.(x \uparrow (\mathsf{zring}\ z)//Hx) = (x//Hx')^z$.

Lemma power_div : $\forall_{x:\mathbb{R}}.\forall_{p,q:\mathbb{N}}.\forall_{Hx,Hx',Hq,Hq'}.$
$(x \uparrow ((\mathsf{nring}\ p)/(\mathsf{nring}\ q)//Hq)//Hx) = (\mathsf{NRoot}\ x\ q\ Hx'\ Hq')^p$.

Lemma Exp_power : $\forall_{x:\mathbb{R}}.\forall_{He}.(E \uparrow x//He) = (\mathsf{Exp}\ x)$.

Finally, x^y is always positive.

Lemma power_pos : $\forall_{x,y:\mathbb{R}}.\forall_{Hx}.0 < (x \uparrow y//Hx)$.

This operation can easily be lifted to a functional operator that given two functions f and g yields $\lambda_x.f(x)^{g(x)}$, defined wherever g is defined and f is positive.

Definition FPower (F G:PartIR) := Expon $\circ$ (G$\times$(Logarithm $\circ$ F)).

Lemma FPower_domain : $\forall_{\mathsf{F,G:PartIR}}.\forall_{\mathsf{x}:\mathbb{R}}.$(dom F x)$\rightarrow$(dom G x)$\rightarrow$
($\forall_{\mathsf{Hx}}$.0 $<$ (F x Hx))$\rightarrow$(dom (FPower F G) x).

The term (FPower F G) is denoted by F $\uparrow$ G.
The rule for the derivative of this operation is

$$(f^g)'(x) = g(x) \times \left[\left(f(x)^{g(x)-1}\right) \times f'(x)\right] + f(x)^{g(x)} \times g'(x) \times \log(f(x)).$$

Formally proving this is a real test to the capabilities of the Coq system; but the automation tactics currently available really suffice to make the proof run quite smoothly and for the most part with only limited input from the user.

Lemma Derivative_power : $\forall_{\mathsf{J:interval}}.\forall_{\mathsf{pJ:(proper\ J)}}.\forall_{\mathsf{F,F',G,G':PartIR}}.$
(positive_fun J F)$\rightarrow$(Derivative pJ F F$'$)$\rightarrow$(Derivative pJ G G$'$)$\rightarrow$
(Derivative pJ F$\uparrow$G
(G$\times$((F$\uparrow$(G$-\lambda$x.1))$\times$F'))+(F$\uparrow$G$\times$(G'$\times$(Logarithm $\circ$ F)))).

Basic Trigonometric Functions

Bishop's treatment of trigonometric functions was also directly translated into Coq. Sine and cosine are both defined as power series. The actual Coq code is omitted, since it directly parallels the definition of the exponential function (except in the sequences one begins from); thus two partial functions Sine and Cosine are defined, with domain $\mathbb{R}$. These are made into setoid functions Sin and Cos.

The Tang function is defined to be the quotient of these two; as was done for the logarithm, a constant Tan is introduced to distinguish results about the function or about its output.

Definition Tang := Sine/Cosine.

Definition Tan := $\lambda_{\mathsf{x}:\mathbb{R}}.\lambda_{\mathsf{Hx}}.$(Tang x Hx).

The first step is to prove that the values of these functions at the origin are as expected. This is done by simply unfolding their definition and computing the sum of the resulting series.

Lemma Sin_zero : (Sin 0) $= 0$.

Lemma Cos_zero : (Cos 0) $= 1$.

Lemma Tan_zero : $\forall_{\mathsf{H}}.$(Tan 0 H) $= 0$.

The derivatives of Sine and Cosine are then proved, using one of the above discussed corollaries.

Proving the rule $\sin(x+y) = \sin(x)\cos(y) + \cos(x)\sin(y)$ is more complicated, as it requires rewriting both sides of this equation (seen as functions of x) into their Taylor series representation and showing these to coincide, in order to apply Corollary 2.4.23. Although this requires some work (namely several proofs by induction with an unusual induction hypothesis, since the derivatives of these functions form a sequence with period 4), there is nothing essentially new in the formalized proof—which once again directly translates Bishop's.

Another key step (which Bishop does not mention) is verifying that the side condition in Corollary 2.4.23 holds, i.e., that

$$\frac{r^n\left(\lambda_{\mathsf{x}}.\sin(x+y)\right)^{(n)}}{n!} \longrightarrow 0 \text{ for every positive } r.$$

This is actually a bit more difficult to prove than the previous condition, but still no real challenge.

Once this result has been formalized, one does not need to look inside the actual definitions of Sin and Cos any more in order to prove a variety of algebraic properties, including the Fundamental Formula of Trigonometry ($\cos^2(x)+\sin^2(x) = 1$). The following are a few of these; they are all trivially obtained: none of them requires more than a couple of lines long proof, and many can be proved automatically from previous ones.

Lemma Sin_plus : $\forall_{\mathsf{x,y}:\mathbb{R}}.(\mathsf{Sin\ x{+}y}) = (\mathsf{Sin\ x})\times(\mathsf{Cos\ y}){+}(\mathsf{Cos\ x})\times(\mathsf{Sin\ y})$.

Lemma Tan_plus : $\forall_{\mathsf{x,y}:\mathbb{R}}.\forall_{\mathsf{Hx,Hy,Hxy,H}}.(\mathsf{Tan\ x{+}y\ Hxy}) =$
$((\mathsf{Tan\ x\ Hx}){+}(\mathsf{Tan\ y\ Hy}))/(1{-}(\mathsf{Tan\ x\ Hx})\times(\mathsf{Tan\ y\ Hy}))//\mathsf{H}$.

Lemma Sin_inv : $\forall_{\mathsf{x}:\mathbb{R}}.(\mathsf{Sin}\ {-}\mathsf{x}) = -(\mathsf{Sin\ x})$.

Theorem FFT : $\forall_{\mathsf{x}:\mathbb{R}}.(\mathsf{Cos\ x})^2{+}(\mathsf{Sin\ x})^2 = 1$.

Lemma FFT' : $\forall_{\mathsf{x}:\mathbb{R}}.\forall_{\mathsf{Hx,H}}.(1{+}(\mathsf{Tan\ x\ Hx})^2) = 1/(\mathsf{Cos\ x})^2//\mathsf{H}$.

The next step is to define π as twice the first positive zero of the cosine function. The proof in [10] is too long to be presented here; but its general lines will be sketched.

First, one defines an auxiliary sequence by

$$\begin{aligned} x_1 &= 1 \\ x_{n+1} &= x_n + \cos(x_n) \end{aligned} \tag{5.3}$$

Then, the following facts are proved:

- if $0 \leq t \leq x_n$ for some n, then $\cos(t) > 0$;
- $\{x_n\}$ is a monotone increasing sequence;
- for all n, $\sin(x_n) \geq 0$;
- if $1 \leq t \leq x_n$ for some n, then $\sin(t) \geq \sin(1)$;
- from these, one can conclude that $\{x_n\}$ is a Cauchy sequence and define π to be twice its limit;
- taking limits in (5.3), one concludes that $\cos\left(\frac{\pi}{2}\right) = 0$;
- from the first fact above one also concludes that $\cos(x) > 0$ for $0 \leq x < \frac{\pi}{2}$.

The details of the proof can be found in the reference book.

It is interesting to point out that once again the formalization really follows this proof very closely, the only difference lying in the definition of x: since Coq's natural numbers begin at 0, it is easier to begin with $x_0 = 0$, which is consistent with the previous definition since $\cos(0) = 1$.

The steps of Bishop's one-and-a-half page long proof can be seen in the sequence of lemmas formalized.

Fixpoint pi_seq (n:$\mathbb{N}$) : $\mathbb{R}$:= match n with
  | O $\Rightarrow 0$
  | S p $\Rightarrow$ (pi_seq p)+(Cos (pi_seq p))
  end.

Lemma cos_pi_seq_pos : $\forall_{n:\mathbb{N}}.\forall_{t:\mathbb{R}}.$(0 $\leq$ t)$\rightarrow$(t $\leq$ (pi_seq n))$\rightarrow$0 $<$ (Cos t).

Lemma pi_seq_incr : $\forall_{n:\mathbb{N}}.$(pi_seq n) $<$ (pi_seq (S n)).

Lemma sin_pi_seq_nonneg : $\forall_{n:\mathbb{N}}.0 \leq$ (Sin (pi_seq n)).

Lemma sin_pi_seq_gt_one : $\forall_{n:\mathbb{N}}.\forall_{t:\mathbb{R}}.$
  1 $\leq$ t$\rightarrow$t $\leq$ (pi_seq (S n))$\rightarrow$(Sin 1) $\leq$ (Sin t).

Lemma pi_seq_Cauchy : (Cauchy_prop pi_seq).

Definition pi := $2\times$(Lim (Build_CauchySeq pi_seq pi_seq_Cauchy)).

Lemma Cos_HalfPi : (Cos pi/2) = 0.

Lemma pos_cos : $\forall_{x:\mathbb{R}}.$(0 $\leq$ x)$\rightarrow$(x $<$ pi/2)$\rightarrow$(0 $<$ (Cos x)).

Although it is obvious to the human reader, Coq needs to be convinced that π is a positive number. The C-CoRN library includes not only this, but also all relations in the chain

$$-\pi < -\frac{\pi}{2} < -\frac{\pi}{4} < 0 < \frac{\pi}{4} < \frac{\pi}{2} < \pi,$$

which are required again and again throughout the proofs of results about trigonometric functions. The following are a sample of these.

Lemma pos_HalfPi : $0 <$ pi/2.

Lemma neg_invHalfPi : $-$(pi/2) < 0.

Lemma invHalfPi_less_HalfPi : $-$(pi/2) $<$ pi/2.

These lemmas were added to a hints database `piorder`, and a tactic `PiSolve` was written that first converts inequalities into strict inequalities, and then calls `auto` with that database. This very simple tactic turned out to be extremely helpful in proving side conditions during these proofs.

A number of other properties of trigonometric functions can now be proved: Sine and Cosine have period 2π, whereas Tang has period π; several interesting values of these functions are computed (e.g., $\sin(\pi/4) = \sqrt{2}/2$).

Lemma Sin_QuarterPi : $\forall_{\mathsf{Hpos,H}}$.(Sin pi$/4$) $= 1/$(sqrt 2 Hpos)$//$H.

Lemma Tan_QuarterPi : $\forall_{\mathsf{H}}$.(Tan pi$/4$ H) $= 1$.

Lemma Sin_periodic : $\forall_{\mathsf{x}:\mathbb{R}}$.(Sin x$+2\times$pi) $=$ (Sin x).

Lemma Tan_periodic : $\forall_{\mathsf{x}:\mathbb{R}}.\forall_{\mathsf{Hx,Hx'}}$.(Tan x$+$pi Hx') $=$ (Tan x Hx).

Furthermore, monotonicity properties are shown for all three functions and the derivative of the tangent is proved. In the last two lemmas, the term H : (proper (olor $-$ pi/2 pi/2)) is universally quantified.

Lemma Sin_pos : $\forall_{\mathsf{x}:\mathbb{R}}.0 <$ x$\rightarrow$x $<$ pi$\rightarrow 0 <$ (Sin x).

Lemma Sin_resp_less : $\forall_{\mathsf{x,y}:\mathbb{R}}$.
$-$(pi/2) $\leq$ x$\rightarrow$x $<$ y$\rightarrow$y $\leq$ pi/2$\rightarrow$(Sin x) $<$ (Sin y).

Lemma Derivative_Tan_1 : (Derivative H Tang $1/$(Cosine^2)).

Lemma Derivative_Tan_2 : (Derivative H Tang λx.$1 +$ Tang^2).

Inverse Trigonometric Functions

Finally, the inverse trigonometric functions ArcSin, ArcCos and ArcTan are formalized.

Bishop's work only includes a brief description of arcsin, but the relation $\cos(x) = \sin(\pi/2 - x)$ suggests the definition $\operatorname{arccos}(x) = \pi/2 - \arcsin(x)$; arctan is defined through an analogous process to arcsin.

The whole formalization is very similar to that of the logarithm function. First, it is proved that the function $\lambda_{\mathsf{x}}.\left(1 - x^{-1/2}\right)$ is continuous on the interval $(-1, 1)$, and this is used to define ArcSin. This is slightly less easy than in the previous case because of all the compositions occurring in the definition of FPower, which generate side conditions that must be verified.

Lemma ArcSin_def_lemma : (Continuous (olor -1 1) (λx.$1-$FId^2)$\uparrow$(λx.$-1/2$)).

Lemma ArcSin_def_zero : ((olor -1 1) 0).

Definition ArcSin := (Fprim ArcSin_def_lemma 0 ArcSin_def_zero).

The domain of ArcSin is trivial to characterize; and Theorem 2.4.19 directly yields its derivative.

Lemma ArcSin_domain : $\forall_{x:\mathbb{R}}.-1 < x \rightarrow x < 1 \rightarrow$(dom ArcSin x).

Lemma Derivative_ArcSin : $\forall_{H:(\text{proper (olor } -1\ 1))}$.
(Derivative H ArcSin (λx.$1-$FId^2)$\uparrow$(λx.$-1/2$)).

From this, it can now be proved that this function is inverse to Sine. Once again, this requires verifying a number of side conditions due to the presence of composed functions (the proof proceeds by showing that both functions have the same derivative).

Lemma ArcSin_Sin_inv : (Feq (olor $-$pi$/2$ pi$/2$) ArcSin $\circ$ Sine FId).

Lemma ArcSin_Sin : $\forall_{x:\mathbb{R}}.-\text{pi}/2 < x \rightarrow x < \text{pi}/2 \rightarrow \forall_H$.(ArcSin (Sin x) H) = x.

At this stage, Bishop once again skips a huge step with a short remark[15]: in order to prove that the inverse composition is also the identity, it is necessary to show that ArcSin totally maps the interval $(-1, 1)$ into the interval $(-\pi/2, \pi/2)$. A formal proof of this requires some version of the Intermediate Value Theorem, which was therefore formalized but which will not be shown here[16].

Lemma ArcSin_range : $\forall_{x:\mathbb{R}}.\forall_{Hx}$.
$-$pi$/2$ <(ArcSin x Hx) $\wedge$ (ArcSin x Hx) < pi$/2$).

Lemma Sin_ArcSin : $\forall_{x:\mathbb{R}}.\forall_{Hx}$.x = (Sin (ArcSin x Hx)).

Lemma Sin_ArcSin_inv : (Feq (olor -1 1) Sine $\circ$ ArcSin FId).

As a curiosity, one shows that ArcSin is an increasing function.

Lemma ArcSin_resp_leEq : $\forall_{x,y:\mathbb{R}}.-1 < x \rightarrow x \leq y \rightarrow y < 1 \rightarrow$
$\forall_{Hx,Hy}$.(ArcSin x Hx) $\leq$ (ArcSin y Hy).

The definition of ArcCos, as was said, is slightly different.

[15]This was corrected in the second edition [11].

[16]Several constructive substitutes of the Intermediate Value Theorem will be discussed in Section 6.3.

Definition ArcCos := λx.pi/2 − ArcSin.

All properties of ArcSin can now be used to state similar ones for ArcCos. Finally, the formalization of the inverse tangent function ArcTan is analogous *mutatis mutandis* to that of ArcSin.

5.6 Conclusions

By the time the work described in this chapter had been completed, the original FTA-library had grown to more than twice its original dimensions, as measured by the total size of the input files (see Figure 5.2). Instead of the FTA-library's 1400 lemmas, C-CoRN now included about 3100 formalized lemmas and over 800 definitions.

Description	Size (Kb)	% of total
Algebraic Hierarchy (incl. tactics)	533	26.4
Real Numbers (incl. Models)	470	23.3
FTA (incl. Complex Numbers)	175	8.7
Real Analysis (incl. Transc. Fns.)	842	41.6
Total	2020	100

Figure 5.2: Contents and size of C-CoRN (input files)

Size, however, isn't all. Although the simple fact that it *was* possible to formalize a whole chapter of Bishop's book [10] can already be seen as a sign of success by itself, there are other more interesting aspects which can be observed from the formalization process itself.

One of the positive aspects of C-CoRN is its readability. As was hopefully made clear by the frequent display of actual (correct) Coq code throughout this whole chapter, the correspondence between the formalized statements and their original formulation in [10] is almost always straightforward.

Another point which cannot be overemphasized is the clear parallel between most of Bishop's informal proofs and the corresponding formalization. Except for some specific key areas, most of the work consisted of simply rewriting the original proofs in Coq input. The exceptions arose in areas which are universally acknowledged to be difficult to formalize (such as finite sets and multiply indexed sums) or had to do with mathematically fuzzy concepts (like that of "close enough approximation") or heavy overloading (as in the notation f' for the derivative of f).

It is interesting to remark that in informal discussions with researchers who formalized similar results in other theorem proving systems (see Section 2.1) it turned out that double sums had been recognized as a major difficulty by people working with systems as diverse as Isabelle or PVS.

Yet another positive observation is that the places where the formalization becomes significantly more difficult than the informal development tend to occur at the lower stages of the theory. The whole section on elementary transcendental functions was surprisingly easy to formalize; and, as shown in Section 5.2, most of the problems in the area of continuous and differentiable functions arose before these concepts had even been introduced.

All in all, the objectives of this work are felt to have been successfully achieved. It was shown how a whole chapter of a mathematics book could be formalized in a way that stays close to the original presentation, and there are reasons to believe that this formalization can be used as a basis for further work.

Chapter 6

Program Extraction

Earlier on, in Section 2.3, several reasons were given for developing a library of constructive mathematics in Coq. One of the issues briefly addressed was that of applications. This chapter focuses one such application: program extraction.

Program extraction is a natural subject to address when one is working in formalizing constructive mathematics.

On the one hand, constructive mathematics has intrinsic computational meaning, as will be explained below; that is, every constructive proof of a mathematical statement contains an implicit algorithm. In particular, a proof of an existential statement implicitly provides a mechanism to actually obtain a witness.

On the other hand, *formalized* constructive mathematics is by its own nature spelled out in all detail, meaning that turning this implicit algorithm into an explicit one (in some programming language) can be done mechanically once the necessary translation has been defined.

Finally, the C-CoRN library is one of the largest formalizations of constructive mathematics, at least to the author's knowledge, and thus provides a unique opportunity to experiment with program extraction, which is available in Coq, on a much broader scale than ever before. This chapter is devoted to a description and analysis of such an experiment.

Section 6.1 begins by formally introducing the theory behind program extraction, namely the notion of "computational content" of a proof, and the different ways this has been implemented in different theorem provers, with a focus on how it is done in Coq. This section also includes an overview of previous work in the area.

The C-CoRN library, in particular the part corresponding to the original FTA-library, is then analyzed in Section 6.2. The reasons for deciding to choose the Fundamental Theorem of Algebra as a test case will be presented, as well as the immediate difficulties encountered. This section describes in detail how the *size* of the extracted program was brought down from (in practice) infinity to a breathtaking 15Mb, and later to a more modest 200kb—the stage where for the first time it could actually be examined and a little understood.

However extracting a computer program isn't much use unless it can actually be executed, and this problem will be in turn addressed in Section 6.3. There, some startling results are presented which show that very simple and apparently innocuous changes in the formalization can bring about astronomical differences in performance. Instead of focusing on the proof of the FTA, two smaller examples are studied here: computing the real number e and the square root function.

There is much to be learned from these two sections, and a more theoretical analysis is the subject of Section 6.4. Using the knowledge gained from the experiments previously described, this section looks at how further formalization should proceed so that program extraction will really come for free. Among the issues discussed here, special attention will be paid to the use of the sorts (type universes) in Coq.

Section 6.5 looks at the whole picture and tries to give an answer to the fundamental issue: is program extraction a useful tool or merely an interesting toy?

Most of the material in this chapter can be explained without dwelling in the actual Coq code. For this reason, the presentation will be kept on the mathematical level, insofar this is possible, and Coq code only presented where necessary.

It is important to stress once again that the motivation behind this work was to understand the possibilities of program extraction. In this sense, everything program generated by extraction is obtained *for free* from the formalization; therefore, all results described in this chapter are considered to be positive.

6.1 Overview

The notion of computational content of a proof has been around since the late 19^{th} century, being mentioned in the work of Kronecker. Brouwer, among others, began to wonder about the intrinsic meaning of the logical connectives, and how one should interpret a proof of a logical statement. This eventually led to what is known as the Brouwer–Heyting–Kolmogorov (BHK) interpretation of the logical connectives, presented in [10, 40, 45, 48].

Kleene made this more precise through the notion of realizability, presented in [63]. Instead of *interpreting* proofs in the BHK style, proofs are *defined* as formal objects: a proof of $A \to B$ is a function mapping proofs of A into proofs of B, a proof of $\exists x.P(x)$ consists of an object t and a proof of $P(t)$, and so on.

Realizability theory already has a flavour of what program extraction is all about, since it speaks about datatypes (pairs,...), objects and functions. But this correspondence was made even more explicit via the Curry–Howard isomorphism [4, 24, 42]. Through it, the same objects in a particular type theory can be read both as formal proofs in a specific logical system and as programs in a well-defined programming language.

However, the Curry–Howard isomorphism is a theoretical result; in practice, some work needs to be done to actually obtain a working program. The main reason for this is that not all of the information contained in a proof is relevant for computation, as two examples clearly show. On the one hand, the "interesting" part of a proof of $\exists x.P(x)$ is typically the witness t which must be present. On the other hand, since $\neg A$ corresponds to $A \to \bot$, and there are no proofs of $\bot$ (in other words, it is an empty datatype), proofs of $\neg A$ are really functions from A to the empty type, that is, they simply spell out the fact that A is also empty. Therefore, no loss of computational information occurs if these proofs are left out.

This is made more precise with the notion of "non-computational" part of a proof introduced by Goad in [36].

Another important point is that computer languages are typically good either for developing proofs or for executing programs, but seldom if ever for both. Therefore, in practice it is also necessary to have a tool that translates the prover's language into an executable one.

Such tools usually kill two birds with one stone, combining the translation with some optimization on the resulting program by removing irrelevant information. This whole mechanism—translating and optimizing—is what is meant by program extraction.

Before presenting the extraction mechanism of Coq in more detail, it is interesting to briefly survey the different ways in which such a tool can be designed.

A priori vs. *a posteriori* methods

Extraction techniques can be divided into two categories according to the time at which the identification of the irrelevant code is made. When some method relies on the knowledge that some terms will never be extracted, regardless of the concrete proof being manipulated, it is called an *a priori* method; when the parts of the proof to be left out in the extracted program are determined at extraction time by actually examining the proof term, the method is an *a posteriori* one.

Rather than being a formal distinction, this categorization really distinguishes two very different approaches which rely on totally unrelated methods of working.

Most of the *a priori* methods of extraction rely heavily on typing. In the type system of the proof assistant, some types are assigned computational content, where others are deemed irrelevant. Then, a translation operator is defined, which maps terms in the prover's language to programs in a given programming language.

This operator has to satisfy some nice properties. On the one hand, terms without computational content should not be extracted (since they represent parts of the proof that are irrelevant for the algorithm being computed). On the other hand, this operator should be proved correct, not only in the sense that the extracted program does compute the algorithm in the proof, but also by showing that, whenever given a correct term in the type theory, the operator will return a correct program in the output language.

The first extraction method for Coq, due to Paulin, was an *a priori* one, described in [54]; this was generalized by Letouzey [47] to the present-day extraction mechanism of Coq (see [17]). A similar mechanism is available in Nuprl [44].

A different way of working is to look at the concrete proof term and analyze its structure, trying to identify parts of the code which are not needed and removing them (pruning). Of course, this has to be done in such a way that the resulting program is still a valid program which is equivalent to the original one. Techniques which follow this approach are known as *a posteriori* methods.

Such methods look at the extracted program. Pruning techniques for simply typed λ-calculi have been discussed by Berardi in [8]; these were generalized to the F_2 system (polymorphic second order λ-calculus) by Boerio [9]. Prost [59] has proposed a more general method, in the setting of the Calculus of Constructions, based on marking computationally irrelevant subterms of the proof term, and has shown how his approach was a generalization of both Berardi and Boerio's methods and of Paulin's extraction mentioned above.

Both approaches have advantages and disadvantages. Intuitively, *a posteriori* methods will in general yield more efficient programs (i.e., programs containing less irrelevant code), since they can profit from the analysis of the actual proof term they act on. But *a priori* methods tend to be much faster, being linear in the length of the input instead of worst-case exponential. Also, as will be discussed later in this chapter, the sheer size of the proof term can make it impossible to analyze it in practice, and *a priori* methods suffer less from this.

The best method, of course, is to use some *a priori* method to eliminate most of the redundant code before extracting, and apply *a posteriori* techniques to the obtained program. Simple as it sounds, though, no such method seems to have been implemented to date.

Internal vs. external extraction

Another important aspect is the target language of the extraction. As was mentioned above, typically one wants to extract a program in an efficient programming language; but in some situations it is also useful to obtain again a term in the language of the proof assistant. In this situation, one speaks about *internal* extraction, as opposed to *external* extraction.

Internal extraction is very interesting from a theoretical perspective. The pruning techniques mentioned in [8, 9, 59] are all examples of internal extraction; correctness can then be expressed and analyzed in the meta-level of the type theory. Another way to look at it is to see internal extraction as an additional set of reduction rules, such that the original proof terms reduce to the corresponding programs, as described in [28].

However, when the intended use of the extracted program is to have something to compute with, it is more interesting to produce it in a traditional programming language. Besides being in general more efficient than computation within proof assistants, these often have one very important advantage: their type systems are often more permissive. For example, the ML type system allows the user to write recursive functions which might not terminate, unlike that of Coq. This means that, when extracting from Coq to ML, one can forget termination proofs and obtain a shorter program. The correctness of the extraction mechanism ensures that the result will still be a terminating program.

The standard extraction mechanisms of Nuprl [44] and Coq [47] are both examples of external extraction.

The Coq extraction

As mentioned in the previous sections, the extraction tool of Coq is an *a priori* external extraction mechanism.

A detailed description of this mechanism can be found in [17] or [47]. In this section, an informal overview will be given, focusing on the issues which are relevant for the discussion in the remainder of the chapter.

The extraction mechanism of Coq is based on the typing. As explained in Appendix A, there are three universes for types in the Calculus of Constructions: **Set**, **Prop** and **Type** (the latter being in fact an infinite family of types).

The sort **Prop** is used for the types of propositions; in other words, types $\mathsf{A} : \mathbf{Prop}$ are intended to represent properties with no computational content.

The sorts **Set** and **Type** are used for the datatypes. Intuitively, **Type** is used for "large" types, whereas **Set** is meant for "small" types; but this distinction has to be taken with a grain of salt. Until Coq version 7.4, **Set** was an impredicative sort: one could define new types in **Set** by quantifying over all terms of type **Set**, with some unexpected consequences which will be discussed later. This was changed in version 8 of the system, so that now one can think of the distinction between **Set** and **Type** as similar to that between sets and classes in set theory.

With this in mind, the extraction mechanism is very simple to describe: all types of sort **Prop** are forgotten, as well as their inhabitants; and all the rest is kept[1].

The correctness of this mechanism (in itself a far from trivial task) will be assumed from this point onwards. Only one key issue needs to be explained.

Since **Prop** terms are forgotten by the extraction mechanism, it is important to ensure that they really do not influence the outcome of any computation. This is achieved by restricting the elimination rules for inductive types: one is *not* allowed to define a computational object (that is, an object whose type lives in **Set** or **Type**) by analysis of an object with a non-trivial type with sort **Prop**. In other words, if $\mathsf{A} : \mathbf{Prop}$ is an inductive type with more than one constructor, or with a constructor with a computational parameter, and $\mathsf{x} : \mathsf{A}$ is in the context, one is not allowed to do case analysis on x if the current goal lives in **Set** or **Type**.

The two exceptions arise when A has zero or one constructors. A type with no constructors is isomorphic to $\perp$, so having a term of that type in the context is equivalent to having an inconsistent context; from the extracted program's point of view this corresponds to an unreachable part of the code, and the corresponding elimination principle is extracted as an exception. A type with one constructor raises no problems provided that that constructor takes no computational arguments, since elimination will then just replace the logical context with an equivalent one, but will not give rise to branches in the computation[2].

[1]Of course this is a simplified view, but from the user's perspective it is a detailed enough description of what is happening; in particular it suffices for the discussion in this chapter. More detailed information, particularly on how inductive types are dealt with, can be found in the references.

[2]This is not true if the constructor takes a computational argument, since elimination would then make a computational term "magically" appear.

6.2 The problem of *extracting*

Since Coq comes equipped with an extraction mechanism, it was quite natural to try it on the formalized proof of the Fundamental Theorem of Algebra. Unfortunately, early attempts at this invariably resulted in failure, with the computer running out of memory or crashing altogether; as a result, the FTA-library became widely considered as "too big" to be tackled by the extraction mechanism altogether.

It is now time to recall the discussion about the use of sorts in Section 2.2. There, it was mentioned that the logic had to be moved to **Set** in order to complete the formalization without resort to axioms. Keeping in mind what was said in the previous section, this can now be explained in more detail.

Logical connectives are defined inductively in Coq and, as was seen, elimination of inductive types in **Prop** is where the restrictions above discussed apply.

More specifically, the troubling connectives are disjunction and existential quantification. Negation is defined as an abbreviation; falsity is an empty type (which, as was already said, can be eliminated); and conjunction, though inductive, has only one constructor with two arguments whose type lives in **Prop**.

Inductive $\perp$:= .

Definition $\neg$ (A:**Prop**) := A$\to\perp$.

Inductive $\wedge$ (A B:**Prop**) :=
  | $\wedge_I$: A$\to$B$\to$A $\wedge$ B.

Disjunction, however, has two constructors; and existential quantification has one constructor parameterized by a computational type.

Inductive $\vee$ (A:**Prop**) :=
  | $\vee_{IL}$: A$\to$A $\vee$ B
  | $\vee_{IR}$: B$\to$A $\vee$ B.

Inductive $\exists$ (A:**Set**) (P:A$\to$**Prop**) :=
  | $\exists_I$: $\Pi_{\mathsf{x:A}}$.(P x)$\to$($\exists_{\mathsf{x:A}}$.(P x)).

Furthermore, these connectives *are* eliminated in constructive mathematics to define functions, and not just once or twice. A simple example is the definition of reciprocal on the real numbers.

The model of the real numbers in the FTA-library, described in [31], is the standard construction as Cauchy sequences: a real number is a Cauchy sequence of rationals, and two real numbers are equal iff their difference (which is a Cauchy sequence of rationals) converges to zero.

So, given a real number $x \# 0$, how does one define $\frac{1}{x}$? Well, one looks at the proof of $x \# 0$.

In this model, apartness is defined as a disjunction:

$$x \# 0 \stackrel{\text{def}}{=} (x > 0 \vee x < 0).$$

In turn, the order relation is defined via an existential:

$$x > y \stackrel{\text{def}}{=} \exists_{N \in \mathbb{N}}.\exists_{q \in \mathbb{Q}}.(q > 0) \wedge \forall_{n \in \mathbb{N}}.N \leq n \to q \leq (x_n - y_n).$$

So, to define the sequence of rationals corresponding to $\frac{1}{x}$, first eliminate the disjunction to decide whether $x > 0$ or $x < 0$; then eliminate the existential quantifier to find an appropriate N. Since 0 is represented by the constant sequence $\lambda_{n \in \mathbb{N}}.0$, x_n is a rational different from 0 whenever $N \leq n$, and one can define

$$\left(\frac{1}{x}\right)_n \stackrel{\text{def}}{=} \begin{cases} 0 & n < N \\ \frac{1}{x_n} & n \geq N \end{cases}$$

and it should be clear that the concrete sequence very strongly depends on the proof term (although the represented real number is the same).

This simple example shows that the logic cannot be kept in **Prop**; so in the FTA-library everything was done in **Set** instead. However, this undermines the whole mechanism of program extraction within Coq, since this relies precisely on **Prop** being used to mark irrelevant terms!

To recover program extraction, then, a different approach is needed.

Positive and negative statements: CProp

The previous discussion suggests that the logic should be moved back to **Prop** in order to use the program extraction tool. At the same time, however, at least part of it has to stay in **Set** to allow constructive reasoning and function definition by cases. How can these two conflicting demands be satisfied?

Close inspection of the elimination rules for inductive types provides an answer. Elimination of the empty inductive type ($\perp$) never causes any problem; this means that one can have $\perp$: **Prop**. Disjunction and existential quantification, though, must always return a computational object. This observation suggests that using two-sorted logic might be the key. And in fact this approach *is* actually used throughout the Coq standard library, though this is never stated as such: besides the standard logic connectives in **Prop**, there is a "boolean sum" operator, with type **Prop** → **Prop** → **Set**, also known as "decidable or" or "informative disjunction", which is needed for precisely the same reasons that led to the non-standard use of logic in the FTA-library.

Therefore, one should consider not only a sort **Prop** for types of propositions, but also a sort **CProp** for types of *computationally meaningful* propositions. For the time being, **CProp** will be taken to be **Set**; later on, the side effects of this choice will be discussed and other possibilities will be considered. Notice that, even if **CProp** is defined as **Set**, it is useful to have two different names for it, as this allows one to distinguish datatypes from types of propositions just by looking at their type[3].

In accordance with the terminology often used in constructive mathematics, propositions living in **CProp** will sometimes be called *positive* statements, those living in **Prop** *negative* statements.

The main question now is to determine, given a proposition A, whether it should be represented by $\mathsf{A} : \mathbf{Prop}$ or $\mathsf{A} : \mathbf{CProp}$.

Postponing the issue of how to type atomic propositions, most connectives are easy to treat. The above discussion makes it clear that $A \vee B$ and $\exists x.P$ always have computational content, and should thus be typed in **CProp**; $\perp$ does not, and can therefore go into **Prop**.

Conjunction is a more delicate matter. Although it is defined as an inductive type with only one constructor, its computational content depends on the computational content of the conjuncts. If both have sort **Prop**, then their conjunction can also be safely typed in **Prop**; otherwise it will have type **CProp**.

Finally, for implication and universal quantification one would like to use the standard arrow and product rules from type theory. As it turns out, these are easy to justify from a computational perspective. Considering the case of implication (universal quantification is similar), the type $\mathsf{A} \to \mathsf{B}$ corresponds to, under the Curry–Howard isomorphism, and is extracted as, a function type. Suppose that $\mathsf{B} : \mathbf{CProp}$, i.e., B has computational content. Then, as any term of type $\mathsf{A} \to \mathsf{B}$ can be used to produce elements of type B, this type also has computational content regardless of the type of A. Dually, if B has no computational content whatsoever, then a term of type $\mathsf{A} \to \mathsf{B}$ will correspond to a function which only produces non-computational output,

[3]This had been pointed out as an undesirable characteristic of the FTA-library, as was extensively discussed on the MoWGLI [52] mailing list. If one wants to define rendering tools for Coq code, it is important to be able to distinguish between propositions and data so as to know, for example, whether to render product types using $\forall$ (for propositions) or Π (for data); notice that this is done in the display of Coq code throughout this thesis, as explained in Appendix A.4.

and is therefore itself computationally irrelevant. This coincides precisely with the type-theoretical rule for the type of $\mathsf{A} \rightarrow \mathsf{B}$.

As a consequence, $\neg A$ always gets the type **Prop** (hence the name "negative" statements).

In Figure 6.1 the types of the logical connectives are summarized. Even though [22] seems to be the first place where this precise formulation of how computational content is passed through the logical connectives, it should be noticed that several authors had already mentioned specific cases. Thus, the typing rules for $\rightarrow$ and $\forall$ are simply the rules for products in type theory as described e.g. in [4]; the fact that disjunction and existential statements always have computational content whereas negation never does is stated very clearly by Goad in [36]; and the rule for conjunction is very similar to the interpretation of this connective under the modified realizability interpretation mentioned in [60].

$$
\begin{array}{rcl}
\perp & : & \mathbf{Prop} \\
\rightarrow & : & s_1 \rightarrow s_2 \rightarrow s_2 \\
\vee & : & s \rightarrow s \rightarrow \mathbf{CProp} \\
\wedge & : & s_1 \rightarrow s_2 \rightarrow \begin{cases} \mathbf{Prop} & s_1 = s_2 = \mathbf{Prop} \\ \mathbf{CProp} & \text{otherwise} \end{cases} \\
\forall & : & \Pi_{\mathsf{A:t}}.(A \rightarrow s) \rightarrow s \\
\exists & : & \Pi_{\mathsf{A:t}}.(A \rightarrow s) \rightarrow \mathbf{CProp}
\end{array}
$$

(s, s_1 and s_2 denote either **Prop** or **CProp**; t is a datatype)

Figure 6.1: Types of the logical connectives

As for the atomic propositions: when in doubt, play it safe. In other words, if nothing is known about a proposition, then it should be typed in **CProp**, as this is the less restrictive choice. But relations very often come in pairs: equality is the negation of apartness (see page 11), less or equal is the negation of the strict order (see page 13). So, if apartness and less than have to be typed in **CProp** to be on the safe side (and the example in the previous section shows that they do have computational meaning), equality and less or equal are negative relations, and can be put into **Prop** with the guarantee that no definition someone might want to make in the future will ever be forbidden because of typing constraints.

Due to the absence of overloading in Coq[4], implementing logic in this way

[4]And other technical harassments, such as the conventions for automatically naming variables.

was slightly less trivial than might be apparent—but updating the library was still a more or less mechanical task.

Much to everyone's surprise, after this simple change the program extraction mechanism worked, and an ML program was actually obtained from the formalized proof of the FTA. And not such a big program at that: 15Mb (about half of which consists of white spaces), which, though impressive when compared with typical sizes of ML programs, took a mere minute to extract.

The power of negation

Although unexpected at the time, the fact that putting negative statements back in **Prop** is enough to make extraction feasible is with hindsight not surprising. In fact, most of the FTA-library consisted in building the Algebraic Hierarchy; most of the proofs one had to do spoke about equality—the FTA itself existentially quantifies an equality—; and all the tactics originally present in the FTA-library, discussed in Section 4.2, were devoted to automating proofs of equalities. Since equality is apartness negated, all these proofs are computationally irrelevant, and can now thrown away by the extraction mechanism.

For comparison purposes, several other examples from the C-CoRN library were extracted after the success of the FTA. Interestingly, though the files on Real Analysis are themselves bigger than the whole original FTA-library, the extracted code was smaller by a factor of 100. Examination of the formalization revealed yet another important fact: as mentioned in Section 2.4, Bishop's definition preferably uses negative relations because this makes reasoning somewhat easier. In particular, less or equal is used rather than the strict order. In the formalization of the FTA, no such care had been taken.

This turned out to be specially relevant in two places. First, the model of the real numbers (which represented about 7.5Mb, or half of the extracted code) was based on Cauchy sequences. These are formalized as records, containing not only the functional part but also a proof of the Cauchy property:

$$x \text{ is a Cauchy sequence} \stackrel{\text{def}}{\Leftrightarrow} \forall_{\varepsilon>0}.\exists_{N\in\mathbb{N}}.\forall_{m,n>N}.\,|x_m - x_n| < \varepsilon. \tag{6.1}$$

Everything in this proof is computational, which means everything will be kept by extraction. However, the following definition is equivalent:

$$x \text{ is a Cauchy sequence} \stackrel{\text{def}}{\Leftrightarrow} \forall_{\varepsilon>0}.\exists_{N\in\mathbb{N}}.\forall_{m,n>N}.\,|x_m - x_n| \leq \varepsilon. \tag{6.2}$$

Equation 6.1 trivially implies Equation 6.2, since $<$ is stronger than $\leq$; and the other direction is a consequence of $\frac{\varepsilon}{2} < \varepsilon$ for positive ε. But the computational content of both definitions is very different: in the latter case, only a function of type $\mathbb{R} \to \mathbb{N}$ is extracted (giving the N for each ε).

Accordingly changing the definition of Cauchy sequence is a relatively trivial matter which yet again has dramatic consequences. The extracted real numbers now shrank to one-fifth of their previous size, and the whole program to a total of 8Mb.

A similar optimization can be done to the proof of the Kneser lemma (described in [35]), which is the core of the FTA proof. The lemma basically states that, given a complex polynomial p and a point z such that $|p(z)| > 0$, a point z_0 can be found with $|p(z_0)| < |p(z)|$; iteration (under suitable conditions) yields a Cauchy sequence which converges to a root of p.

Unlike the definition of Cauchy sequence, in the Kneser lemma it is fundamental that the inequality stated be strict. Still, a careful look at the proof reveals that the extracted code can be much optimized.

To understand this, one should first look at the transitivity lemmas for the order relation on an ordered field F.

$\mathsf{Lemma\ less\_transitive} : \forall_{\mathsf{x,y,z:F}}.(\mathsf{x} < \mathsf{y}) \to (\mathsf{y} < \mathsf{z}) \to (\mathsf{x} < \mathsf{z})$.
$\mathsf{Lemma\ less\_leEq\_trans} : \forall_{\mathsf{x,y,z:F}}.(\mathsf{x} < \mathsf{y}) \to (\mathsf{y} \leq \mathsf{z}) \to (\mathsf{x} < \mathsf{z})$.
$\mathsf{Lemma\ leEq\_less\_trans} : \forall_{\mathsf{x,y,z:F}}.(\mathsf{x} \leq \mathsf{y}) \to (\mathsf{y} < \mathsf{z}) \to (\mathsf{x} < \mathsf{z})$.
$\mathsf{Lemma\ leEq\_transitive} : \forall_{\mathsf{x,y,z:F}}.(\mathsf{x} \leq \mathsf{y}) \to (\mathsf{y} \leq \mathsf{z}) \to (\mathsf{x} \leq \mathsf{z})$.

The type of the corresponding extracted functions is the following[5].

$\mathsf{less\_transitive} : \Pi_{\mathsf{x,y,z:F}}.(\mathsf{x} < \mathsf{y}) \to (\mathsf{y} < \mathsf{z}) \to (\mathsf{x} < \mathsf{z})$.
$\mathsf{less\_leEq\_trans} : \Pi_{\mathsf{x,y,z:F}}.(\mathsf{x} < \mathsf{y}) \to (\mathsf{x} < \mathsf{z})$.
$\mathsf{leEq\_less\_trans} : \Pi_{\mathsf{x,y,z:F}}.(\mathsf{y} < \mathsf{z}) \to (\mathsf{x} < \mathsf{z})$.

The last lemma, of course, is simply not extracted.

This means that, from the perspective of extraction, it is very different to prove that $a < b$ through a chain of inequalities

$$a < x_1 < x_2 < x_3 < b,$$

which is how it is done in [35], or through a similar chain

$$a < x_1 \leq x_2 \leq x_3 \leq b.$$

The first proof is formalized as

[5] Abusing a bit the informal notation. It is important to stress that the two last extracted functions are *not* in themselves correct; it is the correctness of the extraction mechanism that guarantees that they will only be applied to the "right" arguments.

```
(less_transitive a x1 b H_a_x1
    (less_transitive x1 x2 b H_x1_x2
        (less_transitive x2 x3 b H_x2_x3 H_x3_b))),
```

where H_x_y is a proof term of $x < y$. The extracted term will have exactly the same form.

The second proof can be formalized as

```
(less_leEq_trans a x1 b H_a_x1
    (leEq_transitive x1 x2 b H_x1_x2
        (leEq_transitive x2 x3 b H_x2_x3 H_x3_b))),
```

which is extracted simply as

```
less_leEq_trans a x1 b H_a_x1
```

since the last argument to less_leEq_trans is negative.

Furthermore, the latter program should even be more efficient. The transitivity lemmas are all proved using cotransitivity of $<$, and this is implemented in the model as follows:

1. given: x, y, z and a proof of $x < y$;
2. from the proof of $x < y$ find an index N_{xy} and a positive rational q such that $x_n \leq y_n - q$ for n greater than N_{xy} (see the definition of the order above);
3. find N_x such that $|x_m - x_n| \leq \frac{q}{24}$ whenever $m, n \geq N_x$;
4. find N_y and N_z satisfying a similar condition;
5. define $N = \max\{N_{xy}, N_x, N_y, N_z\}$;
6. apply cotransitivity (of the order in the rationals) to decide whether $x_N + \frac{q}{3} < z_N$ or $z_N < y_N - \frac{q}{3}$;
7. return N, $\frac{q}{6}$ and the appropriate inequality $x < z$ or $z < y$, according to the previous step.

Not only is this algorithm not too simple, but also the N and q in the proof being returned are potentially much more complicated than those in the input. The less often this is invoked, the "simpler" the terms N and q being kept.

This optimization brought the size of the extracted program down by a further 1.5Mb (the Kneser proof itself was reduced to one-third of its original size) to 6.5Mb.

Division revisited

It was at this stage that partial functions were once again brought to the foreground. In Chapter 3 the original definition of division in the FTA-library was shown to generalize to a notion of partial function which was *a posteriori* found out to be very inefficient, due both to the size of the terms stored and to the time spent in $\delta\iota$-reduction. The changes to this definition described in Section 3.3, which turned division into an instance of the C-CoRN notion of partial function, were executed at this point.

Although division is not used a lot in the construction of the model of the real numbers (except, of course, when the concrete division function is defined), it occurs only too often in the proof of the FTA. Redefining it using propositional partial functions reduced that part of the extracted program by a breathtaking 60%, bringing the total extracted code to a mere 3.4Mb.

The final program size was eventually brought down to 3Mb by some minor changes in the proofs, which are too localized and too specific to be of interest to detail.

The changes in the formalization and their effects on the extracted code are summarized in Figure 6.2.

Change	Reals (Mb)	FTA (Mb)	Total (Mb)	Δ(%)
Original	7.5	7.5	15	
New Cauchy seq.	1.5	6.5	8	47
New Kneser proof	1.5	5.0	6.5	19
New Division	1.4	2.0	3.4	48
Various	1.4	1.6	3.0	12

Figure 6.2: Changes on the FTA-library

It should be pointed out that the *ratios* between the sizes of the different versions are actually more relevant than the actual sizes: one can safely assume that if the changes had been done in a different order, these ratios would be similar whereas the differences in size would obviously not.

Inlining and coercions

Throughout this section only the FTA part of the extracted program will be considered.

Although reduced to 20% of its original size, the extracted program was still quite large (1.6Mb) when one considers that the algorithm which it implements is not so complex. Therefore, it was decided to examine the program carefully to try to understand precisely what was taking up so much space.

One of the immediate things one noticed was that, even though the formalization included a constant $\mathbb{C}$ representing the type of complex numbers, this constant was nowhere to be found in the extracted program. This was a bit surprising, as after all the function being extracted operated on complex numbers.

As it turned out, the definition of $\mathbb{C}$ had been fully expanded every time it occurred in the proof term! Since the type of complex numbers was explicitly mentioned around 130 times (every algebraic operation is parameterized on it, for instance), and the definition was around 5000 characters long, this fact alone accounted for nearly half of the program code.

Likewise, the ring of polynomials was another construction which was fully spelled out each time. Although a bit smaller than the previous one, it was mentioned more than sixty times and accounted for about one-fifth of the code. Manually replacing these occurrences by a defined constant therefore reduced the program to 300kb, and plainly showed that not much more simplification was likely to be possible, as most of the functions now became quite short.

(In fact, the only reason why the same thing did not happen with $\mathbb{R}$ was because the FTA-library was parameterized on a generic real number structure. Thus, there were actually *two* programs being extracted: one from the FTA proof, with a parameter $\mathbb{R}$, and one from the model of the reals; only at compilation time were these two programs appropriately connected.)

Personal communication with Letouzey revealed this to be a bug in the extraction mechanism, which was fixed in the distribution of Coq version 7.4. This also had dramatic consequences, as will be discussed further ahead.

At this stage, the coercions now made up most of the program. For example, the real number 0, whose representation in the notation of this thesis is simply $\mathsf{0}$, and which can be written down in the notation of C-CoRN as Zero, has as full form the staggering

$$\mathsf{cm\_unit}\ (\mathsf{cg\_crr}\ (\mathsf{cag\_crr}\ (\mathsf{cr\_crr}\ (\mathsf{cf\_crr}\ (\mathsf{cof\_crr}\ (\mathsf{crl\_crr}\ \mathbb{R}))))))$$

and a similar-looking extracted code.

The final program

The bug-fixed version of the extraction mechanism available brought the size of the extracted program down to around 200kb, almost 100 times less than the original value.

There are two important points to be made before the analysis of the actual *execution* of the program, which has not been mentioned so far.

First, this work raises the question of whether program extraction can be really said to come "for free". One gets rather the opposite idea, since the changes described in the previous pages were made through a period of three months; but this is a deceiving thought.

In fact, most of the changes were actually for the better also from a mathematical perspective. It had already been shown in Section 3.3 that the propositional approach to partial functions was better than the subsetoid approach, so the new definition of division is more satisfactory for reasons other than program extraction. The distinction between positive and negative statements is not an *ad hoc* one, but a meaningful one in constructive mathematics—as is shown by the fact that the part of the C-CoRN library which was built following a reference book privileged the use of negative relations, whereas the FTA-library contained an adaptation of a classical proof and was much less coherent. And although some of the changes can be said at most to be irrelevant from a mathematical perspective, none are illogical or counterintuitive.

Also, this section provided guidelines for future work. The C-CoRN library was originally written with no thought being given to program extraction; but future additions to it can follow some very simple rules, like using negative statements wherever possible, automatically yielding small programs if these ever get extracted.

The second point relates to the bug found in the extraction mechanism. When Coq version 7.4 came out, one of the tests that was made was to re-extract the original 15Mb program, which turned out to be only 650kb now. And—once again, much to everyone's surprise—the *original* FTA-library, with all the logic in **Set**, also extracted after 5 minutes producing... 7.5Mb of output.

A look at this output quickly explained why early attempts at extraction had failed. At the time, it was not possible to extract parameterized proofs and later realize their axioms, so the parameter $\mathbb{R}$ in the proof of the FTA had to be first instantiated with the model of the real numbers, within Coq, and the whole thing then extracted. Because of the bug, the model would not appear as a constant, but every occurrence of it would be replaced by its definition—around 18.800 times. The size of the resulting program can then be roughly estimated at 130Gb, which explains why it was never produced.

So was this whole work done because of a bug? Well, yes. But this doesn't mean that it was made irrelevant because the bug was fixed. In fact, one should be grateful for the presence of the bug in the original extraction mechanism; without it, everyone would probably have been content with the extracted program and no second thought would have been given to how the sorts of Coq could be used in a better way. Thanks to this bug, the very basics of C-CoRN have been rethought and can now be much more convincingly presented as having been done in the right way.

6.3 The problem of *computing*

It is now time to switch focus to the problem of *running* the extracted programs. So far, only optimization of the size of the extracted code has been discussed; arguably all the changes made either had no effect on the complexity of the algorithm or lowered the execution time, since they referred mostly to code elimination or simplification of subroutines. But how does the extracted program actually perform?

There was incidentally a good reason for not testing any but the last extracted program. Due to a known issue (not a bug!) in the extraction mechanism of version 7.3 of Coq, the extracted programs did not completely type check in ML, and some changes had to be done by hand. This meant (in the first program) manually inserting around ten thousand so-called explicit type casts, which understandably wasn't done for the subsequent programs. The problem was solved with the release of version 7.4 of the system, together with the afore-mentioned bug, so the time was ripe to start executing the extracted programs.

Experience shows that when one mentions "program extraction" and "proof of the FTA" in the same sentence someone in the audience will invariably answer with "$x^2 - 2$". So this polynomial (as a polynomial in the complex plane) was defined in Coq, proved to be non-constant, extracted to ML and fed as argument to the FTA program in the hope that some time in the future an answer would come out.

One week later, the only good news was that the program required almost no memory to run. All that time it had been trying to compute the second term[6] in the Cauchy sequence representing the root of $x^2 - 2$, and nothing had happened.

The problem is, even after it was reduced to a reasonable size, the extracted program was still too complex for anyone to understand what exactly was going on (even though the algorithm, in broad terms, is simply the Newton–Raphson method, as explained in [35]). So, instead of looking at the whole program, it was decided to focus on simpler examples from the

[6]In the model being used, the first approximation of any irrational real number is 0/1; therefore the first approximation of this root is $0/1 + (0/1)i$, which isn't very hard to compute.

C-CoRN library in the hope that some insight could be gained on the broader picture.

Rational arithmetic

Real numbers were being modeled as sequences of rationals. Therefore, the obvious candidate for immediate testing was the performance of the extracted program on computation with rational numbers. A number of simple tests showed that arithmetic on $\mathbb{Q}$ was very fast (instantaneous, from a user's perspective).

A closer look at the program showed that an immediate improvement could be made: rational numbers were formalized as pairs $\langle \mathsf{p}, \mathsf{q} \rangle$ with $\mathsf{p} : \mathbb{Z}$ and $\mathsf{q} : \mathbb{N}$, corresponding to the rational number $\frac{p}{q+1}$. The natural numbers ($\mathbb{N}$) of Coq are Peano's unary numbers (since the destructor for these is the usual induction principle), whereas its integers ($\mathbb{Z}$) are binary and therefore computationally much faster (they were in fact developed to be used in computation). Changing the type of rationals to pairs $\langle \mathsf{p}, \mathsf{q} \rangle$ with $\mathsf{p} : \mathbb{Z}$ and $\mathsf{q} : \mathbb{Z}^+$ not only performs better (since binary numbers are being used) but it also makes the formalization easier, since $\mathbb{Z}^+$ does not contain 0 and $\langle \mathsf{p}, \mathsf{q} \rangle$ can then be taken to represent $\frac{p}{q}$.

Adding up to e

The type of real numbers in C-CoRN is simply that of ordered fields with two added components: the Archimedean axiom and a limit operation that assigns to every Cauchy sequence a real number (its limit). Since rationals also form an ordered field satisfying the Archimedean axiom, the only primitive way to construct irrationals is as limits of Cauchy sequences.

All real and complex numbers defined in C-CoRN are in fact either (injections or pairs of) rationals or limits of sequences. In the first category fall numbers like 0, 1, $\frac{1}{2}$ and i; the second is more interesting, since it contains numbers like e, π or the root of $x^2 - 2$. As expected, all the numbers in the first category can be computed just as fast as their rational counterparts: images of rationals are constant sequences. But numbers which are computed as limits behave very differently.

Within this category, the "simplest" numbers to compute are intuitively those which are defined directly as the limit of a sequence of rationals: taking the limit of a sequence of rationals is, in this concrete implementation of the reals, simply taking the first rational in each sequence, and it is easy to understand which numbers are being produced. In the general case some diagonal construction has to be made, the details of which can be found in [31]. This means that in general computing the tenth approximation of a real number x may require computing the hundredth or thousandth approximation of another real x', and it is very easy to start very heavy computations without noticing just by asking for the second approximation of an innocent-looking real number.

One example of a real number which is defined in C-CoRN as the limit of a sequence of rationals is e.

As was shown in Section 5.1, e is defined as the sum of the series $\sum_{n=0}^{\infty} \frac{1}{n!}$. This unfolds within Coq to the limit of the sequence of partial sums of $\lambda_{n\in\mathbb{N}}.1/n!$; and it can be seen[7] that indeed the real number e will be represented by that same sequence of partial sums (seen as a sequence of rationals), as shown in Figure 6.3.

Index	0	1	2	3	4	5	6	7
Num.	0	1	2	5	32	780	93888	67633920
Den.	1	1	1	2	12	288	34560	24883200
Value	0	1	2	2.5	2.66667	2.70833	2.71667	2.71806

Index	8	9	10
Num.	340899840000	13745206960128000	4987865758275993600000
Den.	125411328000	5056584744960000	1834933472251084800000
Value	2.71825	2.71828	2.71828

Figure 6.3: Computed values of e

The first results were not promising at all. Computing any of the first five approximations of e was virtually instantaneous; the sixth took a few seconds; and the seventh didn't finish after more than one hour of computation.

It didn't take long to find the problem. The defining sequence for e, $\lambda_{n\in\mathbb{N}}.\frac{1}{n!}$, is formalized as $\lambda_{\mathsf{n}:\mathbb{N}}.\mathsf{1}/(\mathsf{nring}\ (\mathsf{fac}\ \mathsf{n}))//(\mathsf{fac\_ap\_zero}\ \mathsf{n})$. The argument is a natural number; fac is the factorial function on natural numbers; and the result of this function is then injected into the ring of real numbers via nring, which is defined by induction on the natural numbers.

Thus, computing the n^{th} approximation of e requires computing $k!$ for $0 \leq k \leq n$ *in unary notation*, translating the result as a real number *by adding* 1*'s*, computing its reciprocal and then adding everything together. What's worse, to compute the reciprocal the proof of $k! \# 0$ is analyzed as explained above; and in this case this proof extracts as "compute $k!$ in the ring as 1 added to itself so many times, and since $1 \# 0$ the result is also

[7]In other words, this is a proof by so-called "experimental induction".

apart from 0", *even though in the end the relevant information is that* $k! > 0$ *and, for all terms* $(k!)_n$ *of the Cauchy sequence representing* $k!$*,* $(k!)_n > 1$ *holds!*

This is hardly satisfactory, but fortunately it is also easy to fix. Instead of computing factorials in $\mathbb{N}$ and injecting the result in the ring, one defines a new operator $\mathsf{nring\_fac} : \mathbb{N} \rightarrow \mathsf{R}$, where R is a ring, that computes $n!$ for a given n using directly the multiplication in the ring.

```
Fixpoint nring_fac (n:ℕ) : R := Cases n of
    | 0   ⇒ 1
    | S p ⇒ (nring (S p))×(nring_fac p)
```

It can then be proved by induction that $(\mathsf{nring\_fac}\ \mathsf{n}) \# \mathsf{0}$ for every $\mathsf{n} : \mathbb{N}$, yielding a proof term that will be linear in n.

These changes indeed make it possible to compute more approximations of e, in particular the ones shown in Figure 6.3. However, the program's complexity still remains[8] $O(10^n)$.

Once again, some *ad hoc* improvements both to the definition of nring and to the model of the real numbers helped bring this value down to around $O(2^n)$; but this is still exponential complexity, and it means that around twenty approximations of e could be computed.

The most unsatisfactory thing about this situation is that profiling reveals that the most time is spent... computing values of the zero function on the natural numbers. Practically all of the computation is done within the function extracted from the proof term in the division, which produces a trivial output as was explained above. But the algebraic way in which the operations are formalized requires resorting to a very sophisticated way to find this trivial output.

It was with this in mind that a different approach was attempted by Letouzey. Since the inefficiency seems to be arising from the highly abstract way of computing the proof of $n! \# 0$, why not compute it directly in the model? His idea was: instead of trying to find a more efficient proof term of that type, simply parameterize the definition of e on it. In other words, extract e as a program of type $\Pi_{\mathsf{n}:\mathbb{N}}.\mathsf{n}! \# \mathsf{0} \rightarrow \mathbb{R}$. Then, find *in the model* a proof term of type $\forall_{\mathsf{n}:\mathbb{N}}.\mathsf{n}! \# \mathsf{0}$; this will be a very simple term, containing the information "greater than; 0; 1" (see definition of apartness in Section 6.2). Extract a program from it and give it as argument to the e program.

[8]All estimates of the program's complexity are strictly non rigorous, based on the experimental data rather than on any theoretical analysis, and should thus be read as an informal indicator of the program's performance and not as mathematical statements. A rigorous analysis of the complexity of the extracted programs is well beyond the scope of this work.

The computational behaviour of the extracted program is unrecognizable. The new program can compute the 100^{th} approximation of e in just over one minute. Its complexity appears to still be exponential, but now with a factor of around $\log_2(10)$, meaning that in twice the time one can compute ten more approximations.

The conclusions to draw from this example point in opposite directions. At first sight, it seems that formalizing and computing require different techniques, which means that the whole idea of getting an extracted program for free simply doesn't work. Some of the modifications needed to get a program that works in reasonable time are difficult to justify from a mathematical perspective, and one can argue that they make the formalization end up being *less* elegant.

More careful analysis, though, reveals one small but important detail, and that is the *location* of the changes. The potentially controversial modifications described in this section took place either in the model of the reals, which is by its own nature a part of the library where computational efficiency is more desirable than elegance of the formalization (at least to a degree); or in the specific term which was being extracted. Nothing in the general theory of series was changed, for example, even though most of it was being used. Thus, one might expect that, in general, to make an extracted program executable in reasonable time, one will only need to make a few adjustments to the term being extracted.

This issue will be referred to again in the following section.

Bisecting towards $\sqrt{2}$

After the success with e, it was decided to tackle the next problem on the road to the FTA. And this turned out to be square roots (or rather, generic roots of non negative real numbers).

Roots are defined quite early in the Algebraic Hierarchy, as they are needed for many elementary results (for example, to prove the fundamental relation $|x \times y| = |x| \times |y|$). These are *not* obtained via the Fundamental Theorem of Algebra, but rather more simply as a corollary of the Intermediate Value Theorem (IVT).

THEOREM 6.3.1 Let f be a continuous function on the interval $[a, b]$ with $f(a) < f(b)$. For every $c \in (f(a), f(b))$ there is a point $x \in (a, b)$ such that $f(x) = c$.

This general case is a consequence of the following theorem.

THEOREM 6.3.2 Let f be a continuous function on the interval $[a, b]$ with $f(a) < 0 < f(b)$. Then there is a point $x \in (a, b)$ such that $f(x) = 0$.

Unfortunately, this formulation is only valid classically (as is to be expected, as it implies decidability of equality); to be able to assert the conclusion constructively, some extra assumptions about f have to be made.

It is easier to understand what such extra assumptions should look like by examining the proof of Theorem 6.3.2. Classically, this is done iteratively: one defines $a_0 = a$ and $b_0 = b$. To define a_{n+1} and b_{n+1}, one looks at the midpoint x of the interval $[a_n, b_n]$. If $f(x) \geq 0$, then one takes $a_{n+1} = a_n$ and $b_{n+1} = x$; else $a_{n+1} = x$ and $b_{n+1} = b_n$. It is easy to prove that $\{a_n\}$ and $\{b_n\}$ are convergent sequences whose common limit x satisfies $f(x) = 0$.

Constructively, however, the order on the reals is not decidable, so this method does not work. However, if instead of taking the midpoint of $[a_n, b_n]$ in the previous construction one chooses simply an x in the same interval such that $f(x) \# 0$ then the same method can be made to work (since knowing $f(x) \# 0$ one can decide whether $f(x) > 0$ or $f(x) < 0$). Of course, x has to be “near” the middle of the interval, otherwise the sequences a_n and b_n may not converge to the same limit and the proof fails. Typically one demands that x lie in the middle third of $[a_n, b_n]$.

The most general condition one can put on f is then to say that in any proper interval $[a, b]$ there is an x such that $f(x) \# 0$. This was the version of the IVT that was formalized in the FTA-library.

Proving that polynomials (except for the zero polynomial) satisfy this property is not too difficult. Given p of degree n and a proper interval $[a, b]$, one can choose $n+1$ distinct points in $[a, b]$ (for example, dividing the interval in $n + 2$ parts of equal length and taking the border points between these parts) and write p in the form

$$p(x) = \sum_{i=0}^{n} p\left(a_i\right) \prod_{k=0, k\neq i}^{n} \frac{x - a_k}{a_i - a_k} \tag{6.3}$$

(the equality, in the ring of polynomials, holds because both expressions are polynomials of degree at most n which coincide on $(n + 1)$ points).

Since p is apart from 0, the expression on the right is also apart from 0; strong extensionality of sum and the fact that if a product is not zero then all its factors are not zero allow to conclude that $p(a_i) \# 0$ for some i.

Applying this to polynomials of the form $x^2 - a$, with $a > 0$, one can prove that they always have a root on the interval $[0, a + 1]$. This allows square roots of positive numbers to be defined[9]. The actual formalization

[9]The formalized version improves on this, allowing square roots of non negative num-

defines these as the limit of the sequence of lower endpoints of the successive intervals considered; the Cauchy sequence representing $\sqrt{a}$ will have as n^{th} term the left end of the interval obtained after $(n-1)$ iterations of the IVT.

Original experiments suggested that extraction from this proof yielded a program computing $\sqrt{2}$ that ran forever (in practice), except of course for the trivial first approximation. The success of the experiment with e, though, gave reasons for optimism; and a little bit of patience showed that, after all, the program *did* eventually provide a second approximation for $\sqrt{2}$. After 52 hours of intense computation it gave an output—0. Of course, one should point out that it is a better 0 than the first approximation: it is now a "0" between 0 and somewhere around 2, instead of a "0" between 0 and 3.

All but a few seconds of the computation time were spent in the iterative part of the IVT, and a more detailed analysis[10] showed that the bottleneck was finding the point where the interval should be split. This is not too surprising, as even for a simple polynomial like $x^2 - 2$ Equation 6.3 yields the polynomial (on x)

$$-\frac{23}{16}\left(\frac{x-\frac{3}{2}}{-\frac{3}{4}}\right)\left(\frac{x-\frac{9}{4}}{-\frac{3}{2}}\right)+\frac{1}{4}\left(\frac{x-\frac{3}{4}}{\frac{3}{4}}\right)\left(\frac{x-\frac{9}{4}}{-\frac{3}{4}}\right)+\frac{49}{16}\left(\frac{x-\frac{3}{4}}{\frac{3}{2}}\right)\left(\frac{x-\frac{6}{4}}{\frac{3}{4}}\right)$$

which then has to be analyzed by repeated calls to strong extensionality.

Is there no simpler way to proceed? Well, in fact there is. The constructive version of the IVT which is being used is a very general one which makes minimal assumptions on the function; but it is precisely this lack of information that makes the algorithm so complicated.

If one considers instead strictly monotonous (partial) functions defined on $[a, b]$, then it is immediate to see that the IVT holds; and the same proof can be done in a much more immediate way. In the case the function is increasing (the other case is analogous), given *any* two points x and y in $[a, b]$ such that $x < y$, necessarily $f(x) < f(y)$; one appeal to cotransitivity allows one to conclude that either $f(x) < 0$ or $0 < f(y)$, and the remainder of the iterative construction goes through. As before, by choosing x and y such that the new interval is sufficiently smaller than the original $[a, b]$ (for example, by dividing the interval in three and taking the resulting points) one can prove convergence of the sequences of endpoints and obtain a zero of the function.

This alternative constructive analogue of the IVT had already been formalized in C-CoRN (it was necessary for the definition of the inverse trigonometric functions), so adapting the definition of square root to use this theo-

bers.

[10]The data was provided by the `gprof` utility.

rem instead was rather trivial. The difference in performance, however, was amazing: the second approximation was now produced in just over 15 seconds, and turned out to be 1—so not only did the new program run faster, but it even seemed to converge more quickly.

As with e, it was now possible to look at the concrete code and find out where further optimizations in the proof could be made. It was quickly discovered that a more careful use of the transitivity lemmas, as previously described, could make a huge difference in the result; and, since the expensive part of the program seemed to be (once again) computing a proof term, experiments have been made to compute this part of the proof in the model, as this had been the key idea that made e computable. So far the program still runs in exponential time (taking about 3 times longer to compute each successive approximation), but there is hope this might be improved on. Figure 6.4 shows the approximations of $\sqrt{2}$ computed by the extracted program. The 12^{th} approximation took more than 30 hours to compute.

Index	0	1	2	3	4	5	6	7(= 8 = 9)
Num.	0	3	3	3	945	945	945	1172721820203
Den.	1	3	3	3	729	729	729	847288609443
Value	0	1	1	1	1.2963	1.2963	1.2963	1.3841

Index	10(= 11 = 12)
Num.	8972023290084239199282064334921354345290756185
Den.	6362685441135942358474828762538534230890216321
Value	1.4101

Figure 6.4: Computed approximations of $\sqrt{2}$

What about the FTA?

After the discussion in the previous sections, it should come as no surprise that the program extracted from the proof of the Fundamental Theorem of Algebra never terminated. In particular, it required computing square root of two through a method that took 52 hours to produce its first approximation.

However, the results obtained so far are promising. One may hope that in a not too distant future it will be possible to run the extracted program in reasonable time, even if never at a speed that can rival computer algebra systems. Of course, unlike those, all the programs that have been or ever will be extracted from the C-CoRN library come with the guarantee of correctness for free.

Another point which has not been mentioned is the efficiency of the representation of the reals. Optimizing this also fell beyond the purpose of this work, which is a reason why only very basic changes (like changing the unary denominators into binary) were made. In particular, it would probably make a huge difference if fractions were at least simplified; the tenth approximation of $\sqrt{2}$, which was

$$\frac{897202329008423919928206433492135434529075618 5}{63626854411359423584748287625385342308902163 21},$$

can be expressed as the irreducible fraction

$$\frac{27755}{19683};$$

intuition does seem to say that computing with the latter expression would probably be quicker than with the former...

Still a lot can be learned from the experiments done so far. The next section is devoted to a more theoretical analysis of the logic in C-CoRN and to the use of sorts in Coq in a more general perspective which will hopefully make large formalizations more easily yieldable to program extraction in the future.

6.4 On the logic of sorts

In Section 6.2 it was shown how the C-CoRN library was changed, in particular its use of the types in the definition of the logical connectives, so that a program could be extracted, and reduced to a reasonable size.

In this section the use of sorts will be brought up again; in particular it will be argued that having **CProp** defined as **Set** is not desirable, and a better solution will be presented.

The use of sorts in Coq has been a topic of disagreement since the time of the FTA-project. As this chapter has shown, the logic was moved first from **Prop** to **Set**, and then partly back to **Prop**, with the computational statements in **Set** under the pseudonym of **CProp**.

Also, there have been some informal discussions on whether the setoids of C-CoRN should not in fact be “typoids”; that is, it has been argued that the carrier of a structure of type CSetoid should in fact have type **Type** instead of **Set**.

There are several reasons for this.

One good reason, which has already been mentioned at the end of Section 3.3, is that structures like the collection of partial functions over a setoid obviously have setoid structure themselves, but the typing rules forbid these to be formalized in C-CoRN. An even simpler example is the type of subsetoids of a subsetoid S, which must also be typed in **Type** (since one of its fields has type $\mathsf{S} \to \mathbf{CProp}$).

Another desirable structure, which cannot be defined with the previously defined setoids, is the setoid of logical propositions with equivalence as an equality relation[11]. Since **Prop** has itself type **Type**, moving the type of setoids up in the hierarchy of the Coq sorts would also allow this example to come to life.

However, there is one problem which arises if setoids are moved up to **Type**, and that is again the issue of the logic. Since inductive types (such as record types) of sort **Set** cannot be eliminated over **Type** because of consistency issues, as explained in [18, 55], the C-CoRN library would immediately collapse unless **CProp** were also redefined to be **Type**.

There are other reasons to think about redefining **CProp** as **Type**. For example, the logical connectives presented in Figure 6.1 could be defined in a much more elegant way with no need for overloading anymore, since **Prop** is a subtype of **Type**. Also, one would be allowed to give an object of type **Prop** whenever one of type **CProp** is expected—something that sounds very intuitive, since a non-informative proposition can be seen as an informative one where the computational content is void.

Unfortunately, the machinery to implement these changes only became available very recently when version 8 of the Coq system was distributed. The C-CoRN library has been changed accordingly, and the results are quite satisfactory. In particular, the simplified notation for the logical connectives that has been used throughout this thesis almost coincides with the notation now used in C-CoRN.

Another interesting consequence of using a two-sorted logic is the possibility of adding extra axioms at different levels with different consequences. One good example of this is the principle of excluded middle.

The natural way to do classical logic on top of C-CoRN is by adding the axiom scheme[12]

$$\Pi_{\mathsf{A}:\mathbf{CProp}}.A \vee \neg A.$$

(As a curiosity, the type of this term is **Type** and not **CProp**: the indexing mechanism of the **Type** universes works in such a way that **CProp** will

[11]Although here there are other subtle issues, like the non existence of an apartness relation in this structure; but the typing question is in a precise sense much more fundamental.

[12]Notice the quantification over **CProp**: if this is defined as **Type**, this quantifier also ranges over all terms of type **Prop**, see Appendix A.1.

correspond to a specific $\mathbf{Type}_i$—which one, the user has no way to know—and the typing rules then require the previous term to have type $\mathbf{Type}_{i+1}$. In [19], a more detailed analysis of similar situations can be found.)

But this axiom destroys some of the nice properties of constructive formalizations, such as program extraction. One might wonder, then, if it would be possible to add a similar axiom in a way that classical reasoning would be permitted, but the core of the formalization would remain constructive.

There is a way to do this, and interestingly enough it proceeds by using *non-informative* versions of disjunction and the existential quantifier which are typed into **Prop**. These will be denoted as $\underline{\vee}$ and $\underline{\exists}$, to distinguish them from the connectives previously introduced; therefore, the logic now contains two disjunctions and two existential quantifiers:

$$\begin{array}{rcl@{\qquad}rcl} \vee & : & s \to s \to \mathbf{CProp} & \exists & : & \Pi_{\mathsf{A:t}}.(A \to s) \to \mathbf{CProp} \\ \underline{\vee} & : & s \to s \to \mathbf{Prop} & \underline{\exists} & : & \Pi_{\mathsf{A:t}}.(A \to s) \to \mathbf{Prop} \end{array}$$

where s is either **Prop** or **CProp** and t is a datatype.

The important point here is that $\underline{\vee}$ and $\underline{\exists}$ cannot be eliminated over **Set**; in particular, they cannot be used to define functions by case analysis.

One can then add the axiom

$$\Pi_{\mathsf{A}:\mathbf{CProp}}.A \,\underline{\vee}\, \neg A,$$

which is typed in **Prop**, to the formalization *without* destroying its computational content. This can be interpreted as doing constructive mathematics with classical reasoning on the meta-level. Classical statements can be proved in **Prop**, but program extraction among others remains available; however properties of the extracted programs can be proved using classical logic.

6.5 The moral of the story

This chapter is not intended as an exhaustive analysis of the topic of program extraction within the framework of C-CoRN, but rather as an introduction to the potential thereof.

Although much work was required first to extract a program from the original FTA-library, then to reduce it to a reasonable size, and finally to make its performance somewhat better, a number of positive results were derived from it.

One of the main consequences of this experiment was that the sorts of Coq are now used in a more satisfactory way in the logic of C-CoRN. The distinction between positive and negative statements, which is known in constructive mathematics, is now built in the C-CoRN library by the use of

Prop and **CProp**. Furthermore, this is done in an almost automatic way, so that the user does not have to worry about the types of any but the atomic propositions.

Redefining **CProp** to be **Type** also opens up several new directions for further development within C-CoRN, which will hopefully be explored in coming years.

On a different note, no such large-scale attempt at program extraction had ever been made, and the fact that a working program *could* be obtained is already a great victory. Even though the top goal, which was to be able to extract roots of polynomials using the program extracted from the proof of the FTA, is nowhere near being achieved, the partial successes which were obtained on the way are encouraging enough and suggest that program extraction might yet one day become a powerful mechanism. Although it is doubtful that extraction will ever become the preferred way to develop programs, it seems reasonable to expect that program extraction will one day yield usable programs.

Chapter 7

Concluding Remarks

The main purpose of this work, as discussed in the Introduction, was to investigate the extent to which it is possible to formalize a chapter of a reference book staying as close as possible to the original presentation; what machinery is required to achieve this; and what the applications of such a formalization are.

This thesis attempts to answer these questions in a satisfactory way. Chapter 5 showed how a whole chapter of Bishop's book on Constructive Mathematics [10, Chapter 2] was formalized in the theorem prover Coq with only minor modifications. The most difficult work turned out to be the beginning, since as the mathematics get more advanced the list of available results also gets much larger; and the formalization of the section on transcendental functions (the last one in the reference chapter) was in fact an almost direct translation of the natural language presentation into Coq input.

The result was the expansion of the original FTA-library to a broad and general-purpose repository, C-CoRN. The present-day directory structure of C-CoRN (each directory roughly corresponds to a subject; arrows denote dependency upon) is shown in Figure 7.1. Figure 7.2 presents the relative sizes of the directories.

As can be seen from this data, the present day repository is much broader than the original FTA-library, which is contained in the four directories `algebra`, `reals`, `complex` and `fta`. The first of these contains the formalization of the Algebraic Hierarchy described in [32], while the original proof of the Fundamental Theorem of Algebra [35] makes up the last directory.

The directories `rings` and `fields` include the model of real numbers from [31]. The isomorphism theorem is included in `reals`, since it speaks about generic real number structures.

More than one-third of the library consists of the formalization of Real Analysis described in Chapter 5; this is mostly included in the directory `ftc`

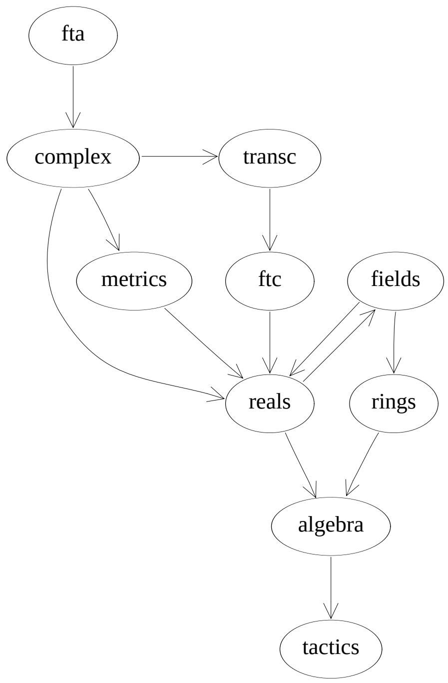

Figure 7.1: Directory structure of C-CoRN

Directory	# files	# lemmas	Size (Kb)
`tactics`	9	—	60
`algebra`	30	1694	583
`reals`	21	737	390
`complex`	4	197	81
`fta`	9	161	90
`rings`	2	55	11
`fields`	3	107	36
`ftc`	30	974	650
`transc`	9	297	191
`metrics`	7	175	90
Total	97	4397	2182

Figure 7.2: Contents of C-CoRN (input files)

except for the results about transcendental functions, including the study of the generalized power function, which are included in directory `transc`.

User-defined tactics are kept together in a separate directory `tactics`. These include not only the original tactics from the FTA-library, `Algebra` and `Rational` with the improvements described in Section 4.2, but also the new `Step` tactic which was the subject of Section 4.3. The tactics described in Section 4.4, specific for reasoning within the field of Real Analysis, are also included here.

Later additions to C-CoRN include the formalization of metric spaces by Loeb, which is included in directory `metrics`, and work by Hinderer [41] proving that the complex numbers form a complete metric space and bringing together the formalizations of complex numbers and of real-valued functions by formally defining the complex exponential and proving its main properties.

However, C-CoRN is just in the beginning. Future plans for development include formalizations of subjects less related to the present-day content of the library, such as Number Theory and Group Theory. Formalizing such subjects within C-CoRN, both reusing what it contains and extending it in a coherent way, will be the real test of its potential.

Still, the fact that people already *have* started working using the library of Real Analysis described in this thesis is the greatest measure of its success. Unlike so often is the case, namely with many of the formalizations mentioned in Section 2.1, one of the main goals in developing C-CoRN—to have a reusable repository of formalized mathematics—seems to be in the good path towards achievement.

But will formalized mathematics really be a part of our future?

While the results in this thesis are promising, there is still a long way to go before the available machinery is powerful enough to convince mathematicians to do their *real* work using a proof assistant. Even though this work showed that formalizing a whole chapter from a standard mathematics textbook is possible, the mathematics it refers to have nevertheless been understood for some three hundred years; even if one takes the constructive aspects into account, it is still a theory that has been fully developed for almost half a century.

On the other hand, formalizations do exist of results which have just been proved; however, these are not developed in the spirit of the C-CoRN development, which was argued in Section 2.3 to be the best way to work, but rather as *ad hoc* formalizations of specific theorems. In particular, they typically suffer from all the characteristics often pointed out as negative aspects of formalized mathematics: little reusability, lack of generality, being geared towards the proof of a very concrete result...

Another problem, which was made very clear in Chapters 2 through 5, is the incipient state of the present day technology that makes some very trivial proofs extremely hard or, at the very best, long. Although automation was a big help (in fact, an indispensable tool specifically in the formalization of Taylor's Theorem, as explained from page 111 onwards), some issues remain intrinsically difficult; and moreover, informal discussions with the people mentioned in Section 2.1 strongly suggest that these issues are pretty much system-independent.

No one has yet found out what the best way is to deal with partial functions; although Chapter 3 tried to answer this question, and an ideal (yet to be implemented) system was described on page 50, this discussion only makes sense for Type-Theory–based systems—and even so, it is not clear that the best way to do partial functions in Coq should be the best way to treat them in Nuprl, for example. Also, as Section 3.3 shows, even within the restricted framework of C-CoRN there are different ways of formalizing partial functions, each with its own advantages and disadvantages; and for other applications the choice made within C-CoRN may not be the best one.

At the moment, a desirable feature for Coq (and C-CoRN in particular) would be a good interface to simplify the use of partial functions from the user's perspective. Since partial functions can be recognized from their type, it shouldn't be too difficult to teach the system to treat application of partial functions to their arguments in a slightly different way than application of total functions (as was briefly suggested in Section 3.3, page 50). Then, the user could type in a lemma like

$$\forall_{\mathsf{x},\mathsf{y}:\mathbb{R}}.\log(x/y) = \log(x) - \log(y)$$

and, from the types of the logarithm and division, the system would deduce the necessary side conditions and translate this result as

$$\forall_{\mathsf{x},\mathsf{y}:\mathbb{R}}.\mathsf{x} > \mathsf{0} \to \mathsf{y} > \mathsf{0} \to \mathsf{y} \# \mathsf{0} \to \log(\mathsf{x}/\mathsf{y}) = \log(\mathsf{x}) - \log(\mathsf{y}).$$

Similarly, if during an interactive proof the user typed in an expression such as $\log(x)$, the system would recognize the need for the hypothesis $x > 0$ and either prove using some pre-determined tactic or add it to the context and leaving it as a separate goal. An immediate advantage is that all proof arguments to partial functions would no longer need to be printed (although they would have to be kept by the system).

Implementing such an interface would bring Coq closer to systems like Nuprl and PVS from the user's perspective, but the nice properties of Coq's type system, namely decidability of type checking and type synthesis, would not be lost, since the terms would not change.

The chapter on integration, Section 5.4, presented another difficult task: formalizing the intuitive equalities that one "knows" hold for sums, such as the rule for changing the order of summation. The proof of Lemma 5.4.1, for example, required proving an equality between two sums which was so intuitively obvious it wasn't even clear *how* it should be stated. Once again, these problems have been remarked upon by people working with other systems, indicating that they are of a more general nature.

Until these issues have been solved, there is little hope of building a system that mathematicians will want to use on a daily basis.

But not all is bad news. After all, even with all these problems it *was* possible to formalize Bishop's chapter on Real Analysis from [10], and the resulting formalization stays quite close to the original. This is encouraging for future work, and suggests that the obstacles above mentioned are probably not unsurmountable; after all, the lemmas that were needed for working with sums are now part of C-CoRN, so one can reasonably hope that future work with sums will be significantly easier. And while the perfect way to represent partial functions has not been found yet, the C-CoRN solution seems to be pretty satisfactory so far.

Another very encouraging aspect is the fact that the automatically generated documentation (available from `http://c-corn.cs.kun.nl/`) is not only very readable, but in many instances quite closely resembles the original piece of mathematics it is meant to represent. This, of course, has also been repeatedly pointed out throughout Chapter 5, where informal and formal statements were presented side by side and can be seen very easily to correspond to one another.

Finally, the work on program extraction, albeit once again far from providing an answer to all problems of mankind, does suggest that it may one day become more than just a nice toy to play with. Although computing 150 digits of e is a feat that any computer algebra system can perform in almost no time, it is still impressive that it can be done in just over a minute by a program which was extracted automatically from an abstract formalization of real numbers. One can once again reasonably hope that in the future program extraction will be efficient enough to be a useful tool in applications where correctness is more important than speed (at least to a reasonable degree).

Appendix A

Coq in a nutshell

In this appendix, a very brief introduction to Coq is presented. This is not meant as a thorough description of Coq, but only as a description of the underlying type theory and of the notation used throughout this thesis.

For a complete overview of the Coq system the reader is advised to refer to the Coq manual[1] [17, Chapter 4].

A.1 The Calculus of Constructions

The language of the Coq proof assistant is a particular type theory known as the Calculus of (Co)Inductive Constructions (CIC). This can be seen as a Pure Type System (PTS) as defined in [4] to which a mechanism for defining inductive and coinductive types is added.

This section presents the CIC in two stages: first, the underlying PTS is described; afterwards, the mechanism for defining inductive types is introduced. Coinductive types are not used in this work and will not be mentioned.

The PTS of the Calculus of Inductive Constructions has the following specification:

$$\begin{array}{rcl}
\mathcal{S} & = & \{\mathbf{Set}, \mathbf{Prop}\} \cup \{\mathbf{Type}_i \mid i \in \mathbb{N}\} \\
\mathcal{A} & = & \{\mathbf{Set} : \mathbf{Type}_i, \mathbf{Prop} : \mathbf{Type}_i \mid i \in \mathbb{N}\} \cup \\
& & \quad \cup \{\mathbf{Type}_i : \mathbf{Type}_j \mid i < j\} \\
\mathcal{R} & = & \{(s_1, s_2) \mid s_1 \in \{\mathbf{Set}, \mathbf{Prop}\} \text{ or } s_2 \in \{\mathbf{Set}, \mathbf{Prop}\}\} \cup \\
& & \quad \cup \{(\mathbf{Type}_i, \mathbf{Type}_j, \mathbf{Type}_k) \mid i, j \leq k\}
\end{array}$$

In other words, the sorts of Coq are **Set**, **Prop** and an infinite family $\mathbf{Type}_i$ indexed on $\mathbb{N}$. The index of $\mathbf{Type}_i$ is hidden from the user, being

[1] Chapter numbers are taken from the manual for version 7.2 of the Coq system.

necessary for consistency of the theory (since simply assuming **Type** : **Type** leads to an inconsistency, see [4]), so that in practice one just "sees" three sorts. Forgetting the indices, **Set**, **Prop** and **Type** all have type **Type**, with the proviso that in the judgement **Type** : **Type** the index of the second term is higher than that of the first one.

The product rules state that whenever $A : s_1$ and $B : s_2$ the product type $\Pi_{x:A}.B$ has type s_2. Again, when s_1 and s_2 are both **Type**, the index of the resulting type has to be at least as large as that of both s_1 and s_2.

The rule $(\mathbf{Type}, \mathbf{Set})$, known as *impredicativity* of **Set**, is known to have some unexpected consequences when coupled with the rules for dealing with inductive types described below. Such consequences include making the principle of excluded middle false, as shown in [30, 58]. Since version 8 of Coq this has been replaced by the rule $(\mathbf{Type}, \mathbf{Set}, \mathbf{Type})$.

Besides this PTS structure there is a *cumulativity* rule, which states that if a term has type **Set** or **Prop** then it also has type $\mathbf{Type}_i$, for every i; and a term of type $\mathbf{Type}_i$ also has type $\mathbf{Type}_j$, for all $j \geq i$. These rule allows the sorts of Coq to be seen as growing universes, each of which contains the previous ones.

A.2 Inductive Types

Besides the types that can be formed from the usual rules of the PTSs, the Calculus of Inductive Constructions also allows the formation of inductive types as described in [17, 18].

Inductive types correspond to least fixed points of monotone operators in complete partial orders. An inductive type A is defined by giving a number of constructors for that type; the inhabitants of A will then be the closed terms which can be written down from those constructors. For example, the type $\mathbb{N}$ of natural numbers is defined by

$\mathsf{Inductive}\ \mathbb{N} : \mathbf{Set} :=$
$\quad | \ \mathsf{0} : \mathbb{N}$
$\quad | \ \mathsf{S} : \mathbb{N} \to \mathbb{N}$

and closed terms of type $\mathbb{N}$ are either $\mathsf{0}$ or generated by application of S to a closed term of type $\mathbb{N}$; that is, they are exactly the terms of the form $\mathsf{S}^n(\mathsf{0})$.

Each inductive type A that is defined generates some constant terms, known as *destructors*, which will be used to eliminate terms of type A. Exactly which destructors are defined depends on the type of A itself, as some elimination rules lead to inconsistency, as explained in Chapter 6.1 and described in more detail in [17, 55]. These destructors allow one to define new terms by case analysis on terms of type A; for example, if A is $\mathbb{N}$, one will be allowed to define functions by cases or primitive recursion, and to prove properties of natural numbers by induction.

A special kind of inductive types are records, corresponding to the Σ-types of Type Theory. Record types are inductive types with only one constructor, corresponding to labelled tuples in programming languages; in the Calculus of Inductive Constructors these tuples may even be dependent, i.e., the type of its components may depend on the previous ones. They are implemented in Coq using a special notation which, besides being more readable than the corresponding inductive definition would be, allows the destructors (which are simply [dependent] projections) to be given user-defined names. The constructor for an inductive type A gets the name Build_A.

An example of a record is the type of setoids (see page 11), defined as

```
Record CSetoid : Type :=
  { cs_crr   : Set;
    cs_eq    : (Relation cs_crr);
    cs_ap    : (Relation cs_crr);
    cs_proof : (is_CSetoid cs_crr cs_eq cs_ap)}.
```

Intuitively, an element X : CSetoid is a tuple which would be represented, in informal notation, as

$$X = \langle |X|, =_X, \#_X, p_X \rangle$$

(the last component would be left out in a mathematics text and written down in natural language).

The inductive type built by Coq is

```
Inductive CSetoid : Type :=
  Build_CSetoid : Π_{cs_crr:Set}.Π_{cs_eq:(Relation cs_crr)}.Π_{cs_ap:(Relation cs_crr)}.
      (is_CSetoid cs_crr cs_eq cs_ap) → CSetoid.
```

A.3 Reduction Rules

There are a number of notions of reduction in the Calculus of Inductive Constructions, which will now be briefly described. Once again, more detailed information can be found in [17, Chapter 4].

The main rule is β-reduction, which is the usual reduction in λ-calculus. Besides this, there are some more specific notions of reduction available:

- In the CIC, one is allowed to *define* new terms from existing ones. If A is defined to be the term t, then it is important that A reduce to t; this reduction is known as δ-reduction.

- Besides this mechanism, Coq also has a mechanism of local definitions, which slightly complicates the typing rules as can be seen in [17]. Unfolding of local definitions is known as ζ-reduction. Local definitions are seldom used in this thesis.

- Inductive types also give rise to a special kind of reduction, called ι-reduction. This is the reduction caused by a destructor applied to a term of an inductive type whose head is a constructor: since destructors computationally perform case analysis on a term of the inductive type, when applied to a term which has as head a constructor the result is simply the case corresponding to that constructor.

The conversion rule for typing, which states that if $\mathsf{t} : \mathsf{A}$ and $\mathsf{A} = \mathsf{B}$ then $\mathsf{t} : \mathsf{B}$, uses the equality generated by all the above reductions (β, δ, ζ and ι) together with the cumulativity rule discussed earlier. This strong notion of convertibility is central to the automation tactics based on reflection discussed in Chapter 4, since they leave the "hard" (but mechanical) part of the proof to the type checker and thus can store relatively large proofs in a small term.

It is important to stress that, even though this is far from being a trivial notion of equality, type checking is still decidable.

A.4 Notation

The Coq system comes with an ASCII interface. Although this is quite practical, it generates notation which is sometimes a bit hard to read. For that reason, several conventions have been adopted throughout this thesis for the display of Coq terms to make them easier to understand.

Coqterms will always be displayed using sans serif font.

Lambda abstraction is denoted as is usual in type theory: $\lambda_{\mathsf{x}:\mathsf{A}}.\mathsf{t}$. Often, in particular when the type of x is too long and can be easily inferred from the context, this will be shortened to $\lambda_{\mathsf{x}}.\mathsf{t}$.

Pi abstraction (i.e. the type of a lambda abstraction) is denoted in two different ways. Since product types are used, according to the Curry–Howard isomorphism, both to denote functional types and universally quantified propositions, dependent products are presented either as $\Pi_{\mathsf{x}:\mathsf{A}}.\mathsf{B}$, in the first case, or as $\forall_{\mathsf{x}:\mathsf{A}}.\mathsf{B}$, in the second. As before, to prevent cluttering up of the terms, when the type of x can be easily inferred these will be shortened to $\Pi_{\mathsf{x}}.\mathsf{B}$ and $\forall_{\mathsf{x}}.\mathsf{B}$, respectively.

Nondependent products are denoted as usual by arrows.

Definitions are presented in the usual notation for Coq:

keyword term *parameter_list* : *type* := *body*.

where the keyword is usually either Definition (for definitions) or Lemma (for terms corresponding to propositions). The parameter list is a list of variables with their corresponding types, which are universally quantified in the body of the term. Finally, *type* is the type of term, and *body* is the definition of *term*. Often (especially in the case of lemmas) the body will be omitted; the keyword is also occasionally omitted.

Inductive types, records and recursive definitions are displayed in the standard Coq notation. For inductive types, the keyword is Inductive and the constructors are separated by the token "|". Record types begin with the keyword Record; the body is inserted in braces, with fields (components of the tuple) in separate lines and divided by semicolons. A simple example of a record definition can be found on page 11. Finally, recursive definitions have as keyword Fixpoint and the body is a match construction with one clause for each constructor of the corresponding inductive type. An example using the inductive type of natural numbers, a variant of the factorial function, can be found on page 155.

A.5 Implicit Arguments and Coercions

Finally, two important features of Coq, not related to the type theory but rather to the user interface, are also important for the understanding of most of the notation: implicit arguments and coercions.

The mechanism of implicit arguments of Coq [17, Chapter 5.7] is a tool designed to make the notation lighter. Very often, due to the possibility of defining type-dependent functions and types, the first argument(s) of a term are types which are reconstructible from the remaining arguments (assuming the term is fully applied). For example, the polymorphic mapping function map on lists has type

$$\Pi_{\mathsf{A},\mathsf{B}}.\Pi_{\mathsf{f}:\mathsf{A}\to\mathsf{B}}.(\mathsf{list}\ \mathsf{A})\to(\mathsf{list}\ \mathsf{B});$$

now, if map is totally applied, say in (map A B f l), it is clear that A and B can be reconstructed from the types of the remaining arguments (in particular, from the type of f). In Coq these two first arguments can be declared as *implicit arguments*, meaning that they will not have to be typed in; the above term could then be entered as (map f l), and the system would figure out its complete form.

A more general possibility, useful in some cases, is to replace an argument by an underscore; the system then tries to figure out what term it should place instead of the underscore (and hopefully succeeds).

Throughout this thesis, whenever arguments to a defined function or type are implicit this will be stated, and afterwards the terms will always be written down in the simplest way possible.

The second very useful feature is Coq's mechanism of coercions [17, Chapter 14]. Coercions are functions which the user tells the system to insert automatically when a given term fails to type-check. That is, if a term t has type A whereas it is expected to have type B, the system will look for a term of type A→B that has been declared as a coercion and, if such a term f exists, will replace t with (f t). Unlike implicit arguments, coercions are not printed by the system; they are very useful to simulate subtyping—they are used systematically throughout the Algebraic Hierarchy so that the user can view and use for example a ring where a group is expected (see Chapter 2.2 for a more precise explanation of how this is done).

More specific notation is described throughout this thesis at the place where it is introduced.

Bibliography

[1] M. Aagaard and J. Harrison, editors. *Theorem Proving in Higher Order Logics,* 13^{th} *International Conference, TPHOLs 2000*, volume 1869 of *LNCS*. Springer–Verlag, 2000.

[2] S.F. Allen, R.L. Constable, D.J. Howe, and W.E. Aitken. The semantics of reflected proof. In *Proceedings, Fifth Anual IEEE Symposium on Logic in Computer Science*, pages 95–105, Philadelphia, PA, 1990. IEEE Computer Society Press.

[3] T. Apostol. *Calculus*, volume 1 of *Blaisdell Mathematics Series*. Blaisdell Publishing Company, New York, 1961.

[4] H.P. Barendregt. Lambda-calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Clarendon Press, Oxford, 1992.

[5] M. Beeson. *Foundations of Constructive Mathematics*. Springer–Verlag, 1985.

[6] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath System*. PhD thesis, Eindhoven University of Technology, 1977. Published as Mathematical Centre Tracts nr. 83 (Amsterdam, Matematisch Centrum, 1979).

[7] L.S. van Benthem Jutting. Checking Landau's "Grundlagen" in the Automath system. In R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors, *Selected Papers on Automath*, pages 701–732, 763–780, 805–808. North-Holland, 1994.

[8] S. Berardi. Pruning simply typed lambda terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.

[9] S. Berardi and L. Boerio. Using subtyping in program optimization. In M. Dezani-Ciancaglini and G.D. Plotkin, editors, *Typed Lambda Calculi*

and Applications, TLCA'95, Proceedings, volume 902 of *LNCS*, pages 63–77. Springer–Verlag, 1995.

[10] E. Bishop. *Foundations of Constructive Analysis.* McGraw–Hill Book Company, 1967.

[11] E. Bishop and D. Bridges. *Constructive Analysis.* Springer–Verlag, 1985.

[12] M. Bridger and G. Stolzenberg. Uniform calculus and the law of bounded change. *American Mathematical Monthly*, August/September 1999.

[13] P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors. *Types for Proofs and Programs, Proceedings of the International Workshop TYPES 2000*, volume 2277 of *LNCS*. Springer–Verlag, 2001.

[14] J. Carlström. Subsets, quotients and partial functions in Martin-Löf's type theory. In Geuvers and Wiedijk [33], pages 78–94.

[15] V.A. Carreño, C.A. Muñoz, and S. Tahar, editors. *Theorem Proving in Higher Order Logics, 15^{th} International Conference, TPHOLs 2002*, volume 2410 of *LNCS*. Springer–Verlag, 2002.

[16] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, New Jersey, 1986.

[17] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, January 2002. Version 7.2.

[18] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *LNCS*. Springer–Verlag, 1990.

[19] J. Courant. Explicit universes for the calculus of constructions. In Carreño et al. [15], pages 115–130.

[20] L. Cruz-Filipe. Formalizing real calculus in Coq. Technical report, NASA, Hampton, VA, 2002.

[21] L. Cruz-Filipe. A constructive formalization of the Fundamental Theorem of Calculus. In Geuvers and Wiedijk [33], pages 108–126.

[22] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics,* 16^{th} *International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag, 2003.

[23] L. Cruz-Filipe and F. Wiedijk. Hierarchical reflection. Submitted.

[24] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1958.

[25] J. Dieudonné. *Calcul Infinitésimal*. Hermann, Paris, 1968.

[26] J. Dieudonné. *Foundations of Modern Analysis*. Academic Press, New York, 1969.

[27] B. Dutertre. Elements of mathematical analysis in PVS. In *Theorem Proving in Higher Order Logics,* 9^{th} *International Conference, TPHOLs 1996*, volume 1125 of *LNCS*. Springer–Verlag, 1996.

[28] M. Fernandez, I. Mackie, P. Severi, and N. Szasz. A uniform approach to program extraction: Pure type systems with ultra σ-types. Technical report, XXVIII Latin-American Conference on Informatics, Uruguay, November 2002.

[29] J.-C. Filliâtre. *CoqDoc: a Documentation Tool for Coq, Version* 1.05. INRIA, September 2003. `http://www.lri.fr/~filliatr/coqdoc/`

[30] H. Geuvers. Inconsistency of classical logic in type theory. Available at `http://www.cs.kun.nl/~herman/note.ps.gz`.

[31] H. Geuvers and M. Niqui. Constructive reals in Coq: Axioms and categoricity. In Callaghan et al. [13], pages 79–95.

[32] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The algebraic hierarchy of the FTA Project. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, pages 271–286, 2002.

[33] H. Geuvers and F. Wiedijk, editors. *Types for Proofs and Programs*, volume 2646 of *LNCS*. Springer–Verlag, 2003.

[34] H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In Aagaard and Harrison [1], pages 162–178.

[35] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the Fundamental Theorem of Algebra without using the rationals. In Callaghan et al. [13], pages 96–111.

[36] C.A. Goad. Proofs as descriptions of computation. In G. Goos and J. Hartmanis, editors, 5^{th} *Conference on Automated Deduction*, volume 87 of *LNCS*, pages 39–52. Springer–Verlag, 1980.

[37] H. Gottliebsen. Transcendental functions and continuity checking in PVS. In Aagaard and Harrison [1].

[38] J. Harrison. *Theorem Proving with the Real Numbers*. Springer–Verlag, 1998.

[39] J. Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics,* 12^{th} *International Conference, TPHOLs 1999*, volume 1690 of *LNCS*, pages 113–130. Springer–Verlag, 1999.

[40] A. Heyting. Die intuitionistische Grundlegung der Mathematik. *Erkenntnis*, 2:106–115, 1931.

[41] S. Hinderer. Formalisation d'éléments d'analyse complexe et de topologie en Coq, 2003. Bachelor thesis, École Normale Supérieure de Lyon.

[42] W.A. Howard. The formulae-as-types notion of construction. In J.R. Sindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[43] A.E. Hurd and P.A. Loeb. *An Introduction to Nonstandard Real Analysis.* Pure and Applied Mathematics. Academic Press, London, 1985.

[44] P.B. Jackson. *The Nuprl Proof Development System, Version* 4.2 *Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 1996.

[45] A.N. Kolmogorov. Zur Deutung der intuitionistische Logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

[46] E. Landau. *Grundlagen der Analysis*. Leipzig, 1930.

[47] P. Letouzey. A new extraction for Coq. In Geuvers and Wiedijk [33], pages 200–219.

[48] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Napoli, 1984.

[49] M. Mayero. *Formalisation et Automatisation de Preuves en Analyses Réelle et Numérique*. PhD thesis, Université Paris VI, December 2001.

[50] M. Mayero. Using theorem proving for Numerical Analysis. In Carreño et al. [15], pages 246–262.

[51] `http://www.mizar.org`.

[52] `http://www.mowgli.cs.unibo.it`.

[53] E. Palmgren. Continuity on the real line and in formal spaces. Technical Report 2003:32, Department of Mathematics, Uppsala University, Uppsala, September 2003.

[54] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[55] C. Paulin-Mohring. Definitions in the system Coq – rules and properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

[56] L.C. Paulson. *The Isabelle Reference Manual*. University of Cambridge, May 2003.

[57] D. Pavlović and M. Escardó. Calculus in coinductive form. In V. Pratt, editor, *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 408–417. IEEE Computer Society, 1998.

[58] Loïc Pottier. Quotients dans le CCI. Technical Report RR-4053, INRIA, November 2000.

[59] F. Prost. Marking techniques for extraction. Technical Report 95-47, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, December 1995.

[60] H. Schwichtenberg. Minimal logic for computable functionals. Technical report, Mathematisches Institut der Universität München, December 2003.

[61] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *The PVS System Guide*. SRI International, December 2001.

[62] K. Slind. *HOL98 Draft User's Manual*. Cambridge University Computer Laboratory, January 1999. Athabasca Release.

[63] A.S. Troelstra. Realizability. In S.R. Buss, editor, *Handbook of Proof Theory*, pages 407–473. North-Holland, 1998.

[64] X. Yu, A. Nogin, A. Kopylov, and J. Hickey. Formalizing abstract algebra in type theory with dependent records. Technical report, Institut für Informatik Albert–Ludwigs–Universität, Freiburg, September 2003.

Index

Samenvatting

Constructieve Reële Analyse:
een Type-Theoretische Formalisatie en Toepassingen

Dit proefschrift beschrijft het formaliseren van wiskunde in het bewijssysteem Coq, met name de formalisatie van Bishops ontwikkeling van Constructieve Reële Analyse [10, Hoofdstuk 2]. Als voorbereiding voor die formalisatie moest er aandacht worden besteed aan kwesties waar men nog nooit aan had gedacht in deze context, zoals de representatie van bepaalde concepten en de ontwikkeling van de nodige hulpmiddelen voor de automatisering.

Deze formalisatie werkt als motivatie voor algemener beschouwingen over de beste manier om een grote bibliotheek te ontwikkelen en te organiseren zodat de inhoud makkelijk kan worden hergebruikt.

Het werk waarover dit proefschrift gaat kan als volgt worden samengevat:

- opbouwen van de C-CoRN bibliotheek (formaliseren van Reële Analyse en ontwikkeling van speciale tactieken);
- ontwikkeling van een werkmethodiek;
- toepassingen op programma-extractie (case study: extractie en optimalisatie van een programma uit de geformaliseerde bibliotheek).

Hoofdstuk 2 geeft een uitgebreider inleiding tot het werk. Hierin worden zowel andere formalisaties van Reële Analyse beschreven, als de bestaande bibliotheek waarop dit werk werd gedaan: de FTA-bibliotheek. Hierna volgen algemenere overwegingen over de methodiek die door de hele formalisatie is gevolgd. Dit hoofdstuk sluit af met een beschrijving van de betreffende wiskunde, Hoofdstuk 2 van [10], en de identificatie van de problemen die behandeld moeten worden.

Partialiteit is een van deze problemen, en daar gaat Hoofdstuk 3 over. Hierin worden mogelijke manieren besproken om partiële functies te formaliseren, in Coq en in andere bewijssystemen, met nadruk op de twee natuurlijkste manieren in verband met dit werk. De voor- en nadelen van beide opties worden geanalyseerd voordat er één wordt gekozen. Delen van dit hoofdstuk zijn gepubliceerd in [20].

Het volgende punt is automatisering. Hoofstuk 4 behandelt dit punt in detail. Zowel verschillende methoden om tactieken in Coq te definiëren, als voorbeelden en voordelen van elke methode worden gepresenteerd. In het bijzonder wordt er aandacht besteed aan de nieuwe tactieken om bewijzen in Reële Analyse te automatiseren. Dit hoofdstuk werd gedeeltelijk gepubliceerd in [20]; delen daarvan zijn in samenwerking met Freek Wiedijk en Hugo Herbelin gedaan.

Hoofdstuk 5 kijkt naar de formalisatie zelf. Daar worden de belangrijkste problemen besproken die voorkwamen, net als de keuzen die op iedere stap moesten worden gemaakt. Dit hoofdstuk is een zeer uitgebreide versie van [21].

Het laatste hoofdstuk onderzoekt één van de toepassingen van de geformaliseerde constructieve wiskunde: programma-extractie. In Hoofdstuk 6 worden de ideeën achter programma-extractie gepresenteerd samen met een overzicht van voorafgaand werk. Daarna komen er twee punten aan de orde: hoe men een programma uit de bestaande bibliotheek *krijgt*; en hoe men een *werkend* programma uit diezelfde bibliotheek kan krijgen. Het grootste deel van dit hoofdstuk komt voor uit samenwerking met Bas Spitters, eerder gepubliceerd in [22]; het eerste deel van Sectie 6.3 komt voor uit samenwerking met Pierre Letouzey en Bas Spitters.

Aan het einde van het proefschrift is er een overzicht van de bereikte doelen en algemene conclusies te vinden.

De hele formalisatie, samen met haar documentatie, is te vinden via de C-CoRN webpagina, `http://c-corn.cs.kun.nl/`.

Curriculum Vitae

Born in Lisbon on July 10, 1978.

Graduated in Applied Mathematics and Computer Science at the Instituto Superior Técnico (I.S.T.) in Lisbon in June 2001 with a final mark of 19/20.

PhD student at the University of Nijmegen under the supervision of Prof. dr. Henk Barendregt between September 2001 and June 2004.

Honourable mention in the 37^{th} International Mathematical Olympiad, held in July 1996 in Mumbai.

Teaching assistant between September 1999 and July 2001 at the Mathematics Department of the I.S.T.

Co-founder of the *Seminário Diagonal*, (seminar for Mathematics students) at the I.S.T. and co-organizer between October 2000 and July 2001.

Member of the Center for Logic and Computation (Lisbon) since June 2001.

Teaching assistant from September 2003 until February 2004 at the Computer Science Department of the University of Nijmegen.

Degree in Gregorian Chant at the Gregorian Institute of Lisbon obtained in July 1999.

Attends singing classes since September 1997, having worked with Elsa Cortez until July 2001 and with Nicoline Krassenburg since January 2002. First appearance as soloist in the oratory “Mariken van Nieumeghen” by Jiri Teml in November 2003.

Has been a member of several amateur choirs since March 1995, among which Coro Ricercare (from September 1997 until July 2001) and Vocaal Ensemble PANiek (since November 2001).

Speaks, reads and writes fluently in Portuguese, English, French and Dutch. Diploma Nederlands als Tweede Taal (July 2003). Also reads Latin, Spanish and Italian.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

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

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

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

P.F. Hoogendijk. *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

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

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

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

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, UL. 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. *Schedulere 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

J.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 Progam Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology 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.F.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

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components.*

Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07