Formalizing Real Calculus in Coq

Luís Cruz-Filipe

Department of Computer Science, University of Nijmegen, The Netherlands
Center for Logic and Computation, IST, UTL, Portugal
lcf@cs.kun.nl

Abstract. We have finished a constructive formalization in the theorem prover Coq of the Fundamental Theorem of Calculus, which states that differentiation and integration are inverse processes. This formalization is built upon the library of constructive algebra created in the FTA (Fundamental Theorem of Algebra) project, which is extended with results about the real numbers, namely about (power) series.
Two important issues that arose in this formalization and which will be discussed in this paper are partial functions (different ways of dealing with this concept and the advantages of each different approach) and the high level tactics that were developed in parallel with the formalization (which automate several routine procedures involving results about real-valued functions).

1 Introduction

In this paper we show how a significant part of real analysis can be formalized in Coq. We deal with differentiation and integration, proving the Fundamental Theorem of Calculus (which states that differentiation and integration are in some sense inverse processes) and Taylor's Theorem (which allows us to express a function in terms of its derivatives while giving an estimate for the error), as well as defining some standard constructions such as function definition by power series and as an indefinite integral.

In parallel with the development of the theory some automation tools (tactics) were built with two aims: allowing a significant part of the proofs to be done automatically and enabling the proof assistant to perform the kind of computation that the average person working in this field can do. With these tools, Coq can prove a large number of results involving derivatives and calculate the derivative of functions in a wide class, looking also at the context where this computation is being done. We hope to extend the system in a near future to be able to solve the problem of integrating rational functions, providing both an answer and a proof that this answer is correct.

The basis for this work was chapter 2 of Bishop's book on constructive analysis ([3]). The formalization was built upon the algebraic hierarchy developed at the University of Nijmegen, described in [7] and available in the Coq library, which included most of the results about real numbers that were needed, namely most of sections 1 to 3 of [3] (where real numbers are defined and their main properties are proved); new results about series were formalized, and sections 4 (dealing with continuity, sequences and series of functions), 5 (differential calculus and Taylor's theorem) and 6 (integration and the Fundamental Theorem of Calculus) were completely formalized. Work is in progress regarding section 7 (which is concerned with exponential and trigonometric functions and their inverses).

Our work centered on formalizing the definitions of basic notions in differential and integral calculus, including notions of:

- continuous function;
- derivative;
- differentiable function;
- Riemann integral;
- (convergent) sequence or series of functions;
- Taylor sum and Taylor series of a function.

Using these definitions, many theorems in this area were formally proved inside Coq; the most important among these were:

- the preservation of continuity through algebraic operations on functions;
- the uniqueness and continuity of the derivative function;
- the derivation rules for algebraic operations on functions and the chain rule for composition;
- Rolle's Theorem and the Mean Law;
- integrability of any continuous function;
- the Fundamental Theorem of Calculus;
- preservation of limits and derivatives through limit operations;
- convergence criteria for series of functions (the ratio test and the comparison test);
- Taylor's theorem.

In section 2 we briefly describe some characteristics of this formalization, including the consequences of working with Coq and of working constructively.

The basic notion which had to be defined and studied at the beginning of the work was the notion of partial function, as most of the common functions of analysis are partial (for example, the logarithm and tangent functions). In section 3 we present the different approaches that were studied and why we chose the one we did.

Section 4 describes how procedures were built that deal with a large class of the most common goals which show up in the area of differential calculus. At the end, we will briefly compare this formalization with similar work already done in other proof systems.

2 Formalizing Mathematics in Coq

Before we go into the specific details of our work, we will briefly discuss some specific Coq issues that influence the way in which our formalization is done.

Coq is a proof tool based on a type system with inductive types called the Calculus of Inductive Constructions (CIC). Through the Curry-Howard isomorphism, proofs are identified with terms and proof-checking with type checking; the construction of a proof then becomes simply the interactive construction of a term which is at the end type-checked.

In the CIC there are two main universes for terms: **Set** and **Prop**. **Set** is meant to be the type for sets and other structures we want to reason about; **Prop** is the type of propositions. There is also an infinite family $\{\mathbf{Type}(i) : i \in \mathbb{N}\}$ such that both **Set** and **Prop** have type $\mathbf{Type}(0)$ and $\mathbf{Type}(i) : \mathbf{Type}(i+1)$, but types in this family will be irrelevant in this paper.

The logic associated with the CIC through the Curry-Howard isomorphism is intuitionistic; this means that to formalize mathematics we must either add the axiom of double negation (in order to be able to work classically) or work constructively. We chose the second alternative, and decided to work following Bishop's approach (see [3]). From our point of view, this is the most general way to work: constructive mathematics results being valid classically, we can always switch to classical reasoning if we want and we will still be able to use all the results we have proved so far[1].

The main characteristic of constructive reasoning is the absence of proofs by contradiction. All proofs have computational content, that is, they provide algorithms to effectively extract witnesses of their statements. So, for example, a proof of an existentially quantified statement $\exists x : A.Px$ will amount to an algorithm that presents an element t of type A such that Pt holds.

One of the immediate consequences of this is that some weak form of the Axiom of Choice should be also available for use; that is, if the only way we can prove a statement like $\exists x{:}A.Px$ is by giving an element satisfying P, then it is also natural to have an operator that allows us to extract such an element from every proof of such a statement.

Unfortunately, Coq does not allow us to define an operator of this kind with type $\mathbf{Prop}\rightarrow\mathbf{Set}$ for two different reasons. At a mathematical level, consistency of the system requires such an operator not to be allowed to exist (see [4], pp. 81-83). On the other hand, Coq comes with a program extraction mechanism (briefly described in Chapter 17 of [4]) which allows programs to be derived from informative proofs; for efficiency reasons, this mechanism assumes that proof objects (living in **Prop**) are irrelevant, as they contain no computational interest. The existence of this operator would undermine this assumption.

Another problem is equality. In type theoretical systems, the natural equality to use is Leibniz equality (given $x, y : A$, $x = y$ iff $\forall P : A \rightarrow \mathbf{Prop}.Px \leftrightarrow Py$); however, this turns out to be too strong a concept for most purposes. Therefore, we have to define ourselves a structure with our own equality. This is done through the notion of *setoid*: a setoid is a pair $\langle S, =_S\rangle$ where $=_S$ is an equivalence relation on S.

1 An approach following the first alternative was independently chosen by Micaela Mayero, see [11].

For the purpose of formalizing real analysis, equality turns out actually not to be so basic a notion, as it is undecidable on the set of real numbers. However, given two real numbers it is always possible to tell when they are distinct (although if they are not distinct we may never know). This motivates us to use what are called *setoids with apartness*: setoids where a second relation $\#_S$, called strong apartness, is defined, with the following properties:

- irreflexivity: for all $x : S$, $\neg(x \#_S x)$;
- symmetry: for all $x, y : S$, if $x \#_S y$ then $y \#_S x$;
- co-transitivity: for all $x, y, z : S$, if $x \#_S y$ then either $x \#_S z$ or $z \#_S y$;
- compatibility with equality: for all $x, y : S$, $x =_S y$ iff $\neg(x \#_S y)$.

The last property actually allows us to do away with equality altogether, although it is not usually done.

Functions and relations on setoids are usually required to reflect this apartness relation; that is, if f is a (unary) function from a setoid S_1 to a setoid S_2, then the following property holds: for any two elements $x, y : S_2$,

$$f(x) \#_{S_2} f(y) \to x \#_{S_1} y \quad .$$

This property is known as *strong extensionality* of f. Predicates in general might not be required to have a similar property (and indeed in many interesting cases they do not), but sometimes the following weaker condition, known as *well definedness*, is required: for all $x, y : S$,

$$x =_S y \to P(x) \to P(y) \quad .$$

From now on, we will use the term "setoid" to mean "setoid with apartness" and denote the equality and apartness relations in a setoid simply by $=$ and $\#$ respectively whenever the carrier set is clear from the context.

At this point we run into another problem of Coq. These definitions work out nicely, but it turns out that if we want to use equality and apartness in a nice way they cannot have type $S \to S \to \mathbf{Prop}$, as would be normal for relations. For this reason, and our desire to use the weak form of the Axiom of Choice which we already mentioned previously, we chose to use also **Set** as the universe for propositions and define our logical connectives to work in this universe with the usual properties.

3 Partial Functions

In our work we only consider partial functions from one setoid to itself. The reason for this is that we are mainly interested in working with real-valued real functions, which satisfy this condition; but generalizing to arbitrary partial functions is quite straightforward and will be done in the near future.

3.1 How to Define Them

Throughout this section A will denote an arbitrary setoid.

The main characteristic of partial functions is that they need not be everywhere defined. Thus, it is natural to associate with each partial function $f : A \not\rightarrow A$ a predicate $dom_f : A \rightarrow \mathbf{Set}$.

In the algebraic hierarchy which we started from, we have a notion of *subsetoid* as being the subset of elements of a setoid S satisfying some property P with the equality relation induced from S; formally, an element of a subsetoid is a pair $\langle x, p\rangle$, where x is an element of S and p is a proof that Px holds. Using this notion, it seems natural to associate every partial function f with a total function on the subsetoid of the elements of A which satisfy dom_f. That is, the type of partial functions will be a dependent record type looking like (in Coq notation):

```
Record PartFunct :=
  {dom    : S->Set;
   dom_wd : (pred_well_def S dom);
   fun    : (CSetoid_fun (Build_SubCSetoid S dom) S)}
```

Here, `dom` is the domain of the function; the second item of the record simply states that this predicate is well defined[2]; and the third item is a setoid function from the subsetoid of elements satisfying the predicate to S.

Then functional application will be defined as follows: given a partial function f, an element $x{:}A$ and a proof $H{:}(dom_f\ x)$, $f(x)$ is represented by the lambda term

$$(\text{fun}\ f\ \langle x, H\rangle)\ ,$$

where fun extracts the subsetoid function from the partial function record.

There are several problems with this definition. One of them is that proofs get mixed with the elements (in the subsetoid construction), which does not seem very natural from a mathematical point of view (where we normally forget about the proof, as long as we know that it exists); another important one is that the terms that we construct quickly get bigger and bigger. For instance, if we have two partial functions $f, g : A \not\rightarrow A$ and we want to compose them, the relevant predicate $dom_{g\circ f}$ will look like

$$dom_{g\circ f} := \lambda x{:}A.(\exists H_f{:}(dom_f\ x).(dom_g\ (\text{fun}\ f\ \langle x, H_f\rangle)))\ .$$

Assuming that for some $x : A$ we know that H has type $(dom_{g\circ f}\ x)$, that is, H is a pair consisting of a proof $H_f$of $(dom_f\ x)$ and a proof that $(dom_g\ (\text{fun}\ f\ \langle x, H_f\rangle))$, then, denoting by π_l and π_r the left and right projections, $(g \circ f)(x)$ will reduce to

$$(\text{fun}\ g\ \langle(\text{fun}\ f\ \langle x, (\pi_l\ H)\rangle), (\pi_r\ H)\rangle)\ .$$

[2] Although this is not required from the predicate in order to build the subsetoid, it turned out to be fundamental for our work, namely to prove results about composition—the chain rule for derivative, for example.

This last expression has several unpleasant characteristics, namely it is totally unreadable and very unintuitive; the fact that we are simply applying g to the application of f to x is totally hidden among the pairing and unpairing operations and the proof terms appearing in the expression. Also, if f and g happen not to look at the proof at all (as is the case if they are total functions), they still have to apply projections to recover the argument from the setoid element. This makes the simplification procedure very time consuming.

Thus, a different approach is needed, and we turn to a common alternative which has already been used for example in the Automath system (see for example [2]). As before, we associate to every partial function f the predicate dom_f, but now we identify f with a function of two arguments: a setoid element and the proof that it satisfies dom_f. That is, our type of partial functions will now be:

```
Record PartFunct :=
  {dom      : S->Set;
   dom_wd   : (pred_well_def S dom);
   fun      : (x:S)(dom x)->S;
   fun_strx : (x,y:S)(Hx:(dom x))(Hy:(dom y))
                (((fun x Hx)[#](fun y Hy))->(x[#]y))}.
```

In this definition, `dom` and `dom_wd` are as before, but the last item of the record type (which was itself a record) has been unfolded into two components: the function itself (as an element of a product type) and the proof of strong extensionality of that function (which was previously hidden in the type of the setoid function). Given f, x and H as before, functional application now looks like

$$(\text{fun } f\ x\ H) \enspace ,$$

which differs from the previous representation in that we removed one functional application (the pairing operation) and that the element x and the proof H are kept completely separated. This means that, for example, if f is total then it can be computed in a much simpler way, because x is directly available and no extra reduction is needed to get it.

Also comparing with the previous example, the application of a functional composition can be written more nicely given f, g, x and H as

$$(\text{fun } g\ (\text{fun } f\ x\ (\pi_l\ H))\ (\pi_r\ H)) \enspace .$$

Notice that in many cases we won't even need to perform any computation on $(\pi_l\ H)$ and $(\pi_r\ H)$, because we won't need to look at the structure of these proofs.

3.2 Working with Function Domains

Once we have partial functions, natural operations with them immediately suggest themselves. The most obvious one (which we have already mentioned) is composition, but algebraic operations (defined point-wise) are also important, at least from the analytical point of view. However, as soon as we try to define this it turns out that it is useful to do some work just with domains.

Since we have identified function domains with predicates, it turns out that what we need is simply a mapping between operations on subsets and operations on logical formulas; that is, given predicates P and Q that characterize subsets X and Y of A we want to define predicates that characterize the sets $X \cap Y$, $X \cup Y$, A, $\emptyset$ and the property $X \subseteq Y$. These can be simply taken to be $\lambda x : A.(P\ x) \wedge (Q\ x)$, $\lambda x : A.(P\ x) \vee (Q\ x)$, $\lambda x : A.\top$, $\lambda x : A.\bot$ and $\lambda x : A.(P\ x) \to (Q\ x)$, respectively. These constructions preserve well definedness (that is, if P and Q are well defined then so will all the predicates defined from them).

As we are concerned with real analysis, it is also important to look at the specific kind of domains we will find. Constructively, it turns out that the most important one is the compact interval, which can be characterized by two real numbers a and b and a proof that $a \leq b$. The predicate corresponding to the interval $[a, b]$ is, of course, simply $\lambda x : \mathbb{R}.a \leq x \wedge x \leq b$.

The reason for this domain to be so important is that all function properties (continuity, differentiability, etc.) are always constructively defined for compact intervals. Bishop argues (see [3]) that point-wise definitions make no sense in constructive mathematics for the reason that equality is undecidable, and so the information that a function f is continuous at some point x is useless because most times we will not be able to know whether we are exactly at x or not. However, if we work with compact intervals we will often be able to tell that we are inside them (unless we happen to be exactly on the boundary), and so use that information. Another important reason is that constructively it is not necessarily true that e.g. point-wise continuity in a compact interval implies uniform continuity in that interval (a counterexample can be constructed with some extra assumptions, see for example [1]), and so in practice it is more natural to begin with the uniform concept altogether.

The other important kind of domain is the interval. In practice, it is difficult to find examples where we really want to work in a domain which is not an interval or a union of two or three intervals, and the main operations (differentiation, integration) and theorems (Rolle's theorem, Taylor's theorem, the Fundamental Theorem of Calculus) always require that the function(s) involved be defined in an interval.

We model intervals as an inductive type with nine constructors, corresponding to the nine basic kinds of intervals: the real line, the left infinite open or closed, the right infinite open or closed and the finite open or closed on either side. To each kind of interval a constructor is associated: for example, finite, closed intervals are identified with applications of a constructor `clcr`[3] of type $\Pi a, b : \mathbb{R}.\mathtt{interval}$. To each of these the obvious predicate is associated, and a property P defined for functions in a compact interval is generalized by

$$P' := (\lambda I : \mathrm{int}, f : \mathrm{fun})(\forall a, b : \mathbb{R})((a \leq b) \to ([a, b] \subseteq I) \to (P\ [a, b]\ f)) \ .$$

[3] <u>C</u>losed <u>L</u>eft <u>C</u>losed <u>R</u>ight

This approach implies that we often have to state very similar lemmas for properties holding in compact intervals and in arbitrary intervals. This is not felt as a disadvantage, however, and is in fact quite close to Bishop's formulation, as most proofs of such properties require distinct reasonings for the compact and the general case.

4 Automation

We will now discuss what kinds of goals we would reasonably expect to be automatically proved and how successful we have been so far in getting the proof assistant to prove them by itself.

So far we have mainly developed a theory of differentiation, so one of the goals we would expect to pop up very often and which should be automatically proved would be given representations of two functions f and g to prove the relation

$$g \text{ is the derivative of } f. \tag{1}$$

We must also keep in mind that we are doing constructive mathematics, where continuity plays a key role: intuitively, one can argue that all functions that we can define constructively are continuous, but no one reasonably expects this ever to be proved (see [3]); therefore, to make proofs easier, it is typically assumed in the statement of every lemma that all the functions involved are continuous. This means that we expect to come quite often across goals such as

$$f \text{ is continuous.} \tag{2}$$

Finally, the third goal comes as a typical side condition of the lemmas we must apply to prove any statement of the previous two kinds: given a set X and a function f, prove that

$$X \subseteq dom(f). \tag{3}$$

In order to get a better understanding of why goals of type 3 show up so often, we have to look at how we define equality of two functions. This concept is parameterized by domains, that is, for every two functions f and g and subset X of $\mathbb{R}$, we say that f and g coincide on X ($f =_X g$) iff they are both defined in X and they coincide on every point of X, that is, for any element $x : X$ and any appropriate proof terms H_x and H'_x,

$$\forall_{x:X}\forall_{H_x,H'_x}\ f(x, H_x) = g(x, H'_x). \tag{4}$$

Two comments are due on this definition:

- The inclusion of X in the domains of both f and g is essential if we want to get something that looks like an equality, namely a transitive relation. If we did not require this condition then every function would be equal in every set to the undefined function, and no substitution properties[4] would hold.

[4] Like if $f =_X g$ and f is continuous in X then so is g.

– The reason why we explicitly state that f and g are defined in X is to make proof development easier. This way, we are left with three independent goals to prove: the two inclusions and (4), which we can prove independently.
  If we did not state the inclusion explicitly, then we would only have to prove

$$\forall_{x:X} \exists_{H_x, H'_x} \; f(x, H_x) = g(x, H'_x) \;\; ,$$

  which differs from the third one in that the proof terms are existentially quantified. However, the existential quantifiers make this goal much more difficult to prove and less suited to automation, which is why we chose the first approach.

We begin by considering goals like (3). Typically, they are proved by looking at the algebraic structure of f and the knowledge that inclusion is preserved through algebraic operations, that is, if $X \subseteq dom(f_1)$ and $X \subseteq dom(f_2)$ then $X \subseteq dom(f_1 + f_2)$ and similarly for other operations. There are some side conditions that have to be verified when division shows up, but these too are usually taken care of by one of a small group of lemmas.

When f has no algebraic structure, there is also a small number of results we can use, namely the facts that constant and identity functions are defined everywhere and that any function which is continuous in X, has a derivative in X or is the derivative in X of some function is also defined in X.

With this knowledge, we can (and we have) easily implement a tactic with the Coq `Hints` mechanism which simply looks at the form of the goal and chooses the right lemma to apply from a not too big list. This turns out to be satisfactory enough for small to medium sized goals, although it doesn't always work when the structure of f is too complicated. In those situations, typically the user has to break down f in smaller parts by himself, and then invoke the automatic prover.

Goals like (2) work in quite a similar way, and have been treated in the same way.

When we turn to goals like (1), however, things turn out to be quite different. From a naive perspective, we would expect this situation to be similar to the previous ones, as we intuitively reason in this situation by cases using a small set of lemmas—the derivation rules. However, when we analyze the situation more carefully it is not as simple as it looks, as we show with a small example. Let f and g be functions everywhere defined by the rules $f(x) = 3x + 4$ and $g(x) = 3$, respectively. If we want to prove $f' = g$, then we would like to begin by applying the derivation rule for the sum; however, in order to do this we also need to have a function that is the sum of two other functions on the right side, and this is not the case. Hence we are stuck.

The trick to to this is, obviously, to replace g by what we get if we differentiate f by using the differentiation rules—in this case, by h such that $h(x) = 3*1+0$. Then we can easily prove that $f' = h$ and we are left with the goal $g = h$, which is also easy to prove. The problem, then, amounts to finding h.

Intuitively, we would like to make some kind of recursive definition that looks at the algebraic structure of f. However, there is no inductive structure in the class of partial functions, so this is not directly possible. However, the Coq tactic language allows us to do something similar: we meta-define (that is, we define in the meta-language) an operator that looks at the structure of f and correspondingly builds h. This operator recognizes algebraic operations, functional composition and can look at the context for relevant information (for instance, if there is a hypothesis stating that for some functions f_1 and f_2 the relation $f_1' = f_2$ then it will use f_2 as a derivative for f_1); however, the proof is still left to be done by the user.

Another, and more powerful, approach is to use reflection (a method which is described in full detail in [8]). We select among the class of all partial functions those whose derivative we know how to compute, and model this as an inductive type $\mathcal{PF}$. This type will have not only constructors for constant and identity functions and algebraic operations on these, but also two special constructors that allow us to add any function about which we know something from the context. This will allow us, for instance, to prove that $(2f)' = 2g$ if we know from the context that $f' = g$.

On $\mathcal{PF}$ we will define two operations: a canonical translation map $[\![\cdot]\!] : \mathcal{PF} \to (\mathbb{R} \not\to \mathbb{R})$ to the real-valued partial functions and a symbolic differentiation operator $'$: $\mathcal{PF} \to \mathcal{PF}$ with the property (stated as a lemma) that for every $s : \mathcal{PF}$

$$[\![s']\!] \text{ is the derivative of } [\![s]\!]. \tag{5}$$

Our problem now amounts to the following: given a function f, how do we determine an element $s : \mathcal{PF}$ such that $[\![s]\!] = f$? That is, how can we define a (partial) inverse to $[\![\cdot]\!]$? Again, this is done at the tactic level in Coq: we meta-define an operator by case analysis that looks at the structure of f and breaks it down; whenever it finds an algebraic operator, constant or identity function, it replaces this by the corresponding constructor in $\mathcal{PF}$; whenever it finds a function that it knows nothing about (that is, an expression like "f") it tries to find an hypothesis in the context that allows it to use one of the two special constructors. If everything goes well, we get indeed an element s with the required property; otherwise we get an error message.

With these tools we can then write down our tactic as follows: given f and g,

1. Find $s : \mathcal{PF}$ such that $[\![s]\!] = f$;
2. Compute s';
3. Replace f by $[\![s]\!]$;
4. Replace g by $[\![s']\!]$;
5. Prove that $[\![s]\!] = f$;
6. Prove that $[\![s']\!] = g$;
7. Apply lemma (5) to prove that $[\![s']\!]$ is the derivative of $[\![s]\!]$.

Steps 3 and 5 may seem superfluous, as s was constructed so that $[\![s]\!] = f$ would hold. The problem, however, is that we did not define this construction as an element of type $(\mathbb{R} \not\to \mathbb{R}) \to \mathcal{PF}$ (because no such element with the required properties exists), so we cannot prove anything in general about this operation.

Still, step 5 turns out to be trivial, as simplification on $[\![s]\!]$ yields f and we just have to invoke reflexivity of equality.

Step 6 is the tricky one. In the most cases, this will reduce to proving some inclusions of domains (which we have already automated) and then equality of two algebraic expressions (which the `Rational` tactic, described in [8], can normally deal with); in some cases, however, this step leaves some work to be done, for example if the equality between $[\![s']\!]$ and g relies on the fact that any two derivatives of a given function must coincide. Even in this cases, however, experience has shown that the goal has been much simplified, so that we do profit from this tactic.

At the present moment, the biggest limitation of this tactic is that it cannot deal with division or functional composition. However, experience shows it to be much more efficient (both regarding computation time and the size of the constructed proof-terms) than the first approach. Also the limitations turn out not such a big problem as they could seem, actually, because we can always add the relevant steps as hypothesis to the context and prove them later; but they still are limitations, and it is interesting to see why we can't deal with these cases in the same way as we dealt with the others.

When we look at the constructive rule for derivation of a division or composition of two functions, they turn out to differ from the other rules in that they have some side conditions that have to be met; as an example, to apply the rule for division, we have to prove that the absolute value of the denominator of the fraction we want to derive is always greater than some positive constant. In order to prove that this side conditions always hold (which we have to do if we want to prove something like $\forall_s [\![s']\!] = [\![s]\!]'$), we have to add in the constructor of $\mathcal{PF}$ corresponding to division an argument stating something about the interpretation of one of the other arguments. But this is not possible in Coq, because we cannot simultaneously define an inductive type and a recursive function over that type (although type theory allows us to do this, namely in this situation).

The case of composition is even worse, as one of the goals we get says something about one function being the derivative of another in an unknown interval. One way to solve this problem would be to make our tactic interactive in some way, but there is no obvious way to do this.

Presently, as we said, these limitations turn out not to be such a big deal. Division is not such an important operation when we work constructively, as most situations that use division can be rewritten so as to use only multiplication; and for composition we can usually achieve our goals by adding hypothesis to the context and applying the chain rule by hand. When none of this works, we can still rewrite the function on the right-hand side of the goal with the first operator we define and proceed by hand.

5 Related Work

This same fragment of real analysis has already been formalized in some systems by different people. We will now briefly describe these formalizations and how they differ from ours.

Micaela Mayero (see [11]) has formalized differential calculus in Coq, including notions of (point-wise) continuity and differentiability, derivation rules, and some work on transcendental functions. However, she does not treat integral calculus or more general theorems 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 other purposes, whence she develops just the theory that she needs for her examples. For the same reason, she argues that it makes more sense for her to work classically—which makes her work totally distinct from ours.

Mayero's treatment of partial functions also differs from ours. As we do, she always associates a domain with every function; however, they are kept completely separated: functions have type $\mathbb{R} \to \mathbb{R}$, domains $\mathbb{R} \to \mathbf{Prop}$, and the domain is always explicitly stated in the formulation of the lemmas. Although this makes it possible to write things down in a way closer to usual mathematical notation (that is, $f(x)$ instead of $f(x, H)$ or something similar) it does have the disadvantage that you can write down things like $\frac{1}{0}$, although it is not clear what they mean.

In the PVS system, Bruno Dutertre (in [5]) has also developed a classical theory of real analysis, including the main definitions in differential calculus. Building upon this work, Hanne Gottliebsen built a library of transcendental functions described in [9], where she defines exponential, logarithmic and trigonometric functions, proving similar results to ours. She then defines an automatic procedure to prove continuity of a large class of functions, which works in a similar way to ours, and shows how it can be used interactively with Computer Algebra systems to guarantee the correctness of applications of the Fundamental Theorem of Calculus.

John Harrison's HOL-light system (described in [10]) is another proof assistant that comes with a library of real analysis; once again, the reasoning in this system is classical. The results included in this library include the usual results on preservation of continuity through algebraic operations, derivation rules, Rolle's theorem and the Mean Law.

Also included in the 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 is proved.

6 Conclusions and Future Work

We have shown how to formalize an important fragment of constructive real analysis and how to use this formalization to build automation tools that can (partly) solve some problems in this area, by providing not only an answer but also a proof that this answer is correct. Presently we deal only with differentiation in a restricted class of functions, but work is being done to generalize the setting to all the elementary transcendental functions. We hope to complete this work with similar results regarding integration, namely by providing a way to integrate rational functions and prove the result correct.

In doing so, we have also shown that it is possibly to build and use modular libraries of mathematical formalizations, as our work was done using a library of real numbers which was already developed and to which no changes were made (although some results had to be added dealing with specific problems). We have also provided evidence to Bishop's claim that it is indeed possible to do useful mathematics without the double negation rule.

Acknowledgments

Support for this work was provided by the Portuguese Fundação para a Ciência e Tecnologia, under grant SFRH / BD / 4926 / 2001.

The author would also like to thank H. Barendregt, H. Geuvers and F. Wiedijk both for the many discussions throughout the development of this work, which contributed to its successful outcome, and for their suggestions regarding the contents and form of this paper.

References

1. Beeson, M., *Foundations of constructive mathematics*, Springer-Verlag, 1985
2. Benthem Jutting, L. S. van, *Checking Landau's "Grundlagen" in the Automath System*, in Nederpelt, R. P., Geuvers, J. H. and de Vrijer, R. C. (Eds.), *Selected Papers on Automath*, North-Holland, 1994
3. Bishop, E., *Foundations of Constructive Analysis*, McGraw-Hill Book Company, 1967
4. The Coq Development Team, *The Coq Proof Assistant Reference Manual Version 7.2*, INRIA-Rocquencourt, December 2001
5. Dutertre, B., *Elements of Mathematical Analysis in PVS*, Proc TPHOLS9, LNCS 1125, Springer, 1996
6. Geuvers, H. and Niqui, M., *Constructive Reals in Coq: Axioms and Categoricity*, in Callaghan, P., Luo, Z., McKinna, J. and Pollack, R. (Eds.), *Proceedings of TYPES 2000 Workshop*, Durham, UK, LNCS 2277
7. Geuvers, H., Pollack, R., Wiedijk, F. and Zwanenburg, J., *The algebraic hierarchy of the FTA project*, in *Calculemus 2001 Proceedings*, Siena, Italy, 13-27, 2001
8. Geuvers, H., Wiedijk, F. and Zwanenburg, J., *Equational Reasoning via Partial Reflection*, in *Theorem Proving in Higher Order Logics*, 13th International Conference, TPHOLs 2000, Springer LNCS 1869, 162-178, 2000
9. Gottliebsen, H., *Transcendental Functions and Continuity Checking in PVS*, in *Theorem Proving in Higher Order Logics*, 13th International Conference, TPHOLs 2000, Springer 2000
10. Harrison, J., *Theorem Proving with the Real Numbers*, Springer-Verlag, 1998
11. Mayero, M., *Formalisation et automatisation de preuves en analyses relle et numrique*, PhD thesis, Universit Paris VI, dcembre 2001