Program extraction from large proof developments

Luís Cruz-Filipe[1,2] and Bas Spitters[1]

[1] University of Nijmegen, The Netherlands
[2] Center for Logic and Computation, Portugal
{lcf,spitters}@cs.kun.nl

Abstract. It is well known that mathematical proofs often contain (abstract) algorithms, but although these algorithms can be understood by a human, it still takes a lot of time and effort to implement these algorithms on a computer; moreover, one runs the risk of making mistakes in the process.
From a fully formalized constructive proof one can automatically obtain a computer implementation of such an algorithm together with a proof that the program is correct. As an example we consider the fundamental theorem of algebra which states that every non-constant polynomial has a root. This theorem has been fully formalized in the Coq proof assistant. Unfortunately, when we first tried to extract a program, the computer ran out of resources. We will discuss how we used logical techniques to make it possible to extract a feasible program. This example is used as a motivation for a broader perspective on how the formalization of mathematics should be done with program extraction in mind.

Keywords: Program extraction, Constructive mathematics, Formalized mathematics, Type Theory.

1 Introduction

It has long been realized that constructive mathematics has computational content in the sense that proofs of existential statements actually correspond to algorithms to compute a witness, see [3,14].

Also intuitionistic logic, which describes the reasoning in constructive mathematics, is the natural language for type-theory based proof-assistants. Among these, Coq currently provides a tool that translates proofs of mathematical statements into functional programs which are guaranteed to be correct. This mechanism, described in detail in [13], works by assigning different types to terms which represent data and terms which represent properties of the data. The latter are assumed to be *computationally irrelevant* in the sense that they can not be used to define data; they can, however, be used to prove properties of functions (termination, correctness, etc.) which operate on this data, and therefore need never be extracted. Throughout this paper, we will refer to this mechanism as *program extraction*. For a short overview of this, see Section 2.

The FTA-project in Nijmegen [10] was the first attempt to formalize a large piece of constructive mathematics, namely the Fundamental Theorem of Algebra, in Coq; therefore it was a natural testing ground for the program extraction mechanism. However, a problem soon arose regarding what, in the formalization, should be considered as "properties". The intuitive approach that all predicates are properties does not work. As an example, consider the logarithm function; a possible way to formalize it is as a function taking a real argument x and a proof that x is positive. According to this approach, this proof is a property of the data which the algorithm itself never needs to examine; but there is actually no algorithm which satisfies this criterion.

This is just an instance of the fact that constructively, proof terms are computationally important and analysis of these can be used to define objects, e.g. functions. Therefore it is necessary to keep the proof-objects, which means that all predicates should be regarded as data. This implies that all proof terms get extracted, which increases the program size enough to make extraction unfeasible.

In Section 3 we describe how the Coq type theory was used to explicitly distinguish between computationally relevant and irrelevant mathematical statements in order to make program extraction possible for this example. Although this is not a new idea, our approach does differ from others in that we focus on the development of mathematics (including logic) as opposed to the precise implementation of the extraction procedure. This is in some sense an orthogonal approach which, we expect, can be combined with the various existing extraction mechanisms.

In Section 4 we examine the more subtle problems that could be discovered once a program could be extracted for the first time, and present solutions that allow to reduce this program to reasonable size; in Section 5 we also show some of the current limitations of the Coq type theory and refer to the work in progress that suggests that these might be overcome in the near future.

In Section 6, we generalize from our experience and analyze how the formalization of constructive mathematics can be done so that program extraction automatically succeeds. To make the development of both constructive and classical mathematics go more smoothly, we propose and discuss some changes to the Coq type theory.

2 An Overview of Extraction

Computer implementations of program extraction have been around for about two decades now. Among these, Paulin [16] was the first to provide an extraction mechanism for Coq.

There are several approaches to the issue, and several different ways to formalize it; for more detailed information, the reader is advised to check the cited works and their references. For a more detailed overview, although in a slightly different setting, see [19].

In all approaches, however, the basic outline is the same: through the Curry-Howard isomorphism, proofs are identified with programs in a given programming language. However, the resulting programs contain much irrelevant information (from the computational point of view), therefore some mechanism is devised to identify this irrelevant parts and remove them from the final program.

One approach, studied in the context of Coq by Prost [18], is to look at the proof term and recursively mark its subterms according to whether they contribute to the final output of the program or not; this marking is done in a way that is coherent with type checking, so that removing the marked parts will still return a correct λ-term. This *a posteriori* approach, also known as *pruning*, has the advantage of typically yielding smaller and more efficient programs, as it is actually an algorithm for dead code removal, but its time cost is exponential, as it invokes the type checking algorithm.

A different approach, which is taken in the Coq proof assistant [20,13], is to define *a priori* a type for data (e.g. propositions) which will never be extracted. The underlying type system then ensures that terms of this type will never be allowed to have computational significance, so that simply removing them yields a correct program. This method itself is naturally faster than the previous approach, but it has the disadvantage that the terms which are not to be extracted have to be specified in advance; therefore the extraction process cannot recognize and eliminate e.g. dummy arguments. It has actually been shown in [18] that the *a posteriori* method of Prost can simulate the *a priori* method of Paulin of which Letouzey's is a generalization.

The most significant drawback of this approach is that propositions are always assumed to be irrelevant for the extraction. This is not true, however, when we are dealing with constructive formalizations where case analysis on proof terms can be (and is) used to define functions. Therefore, we sought a way to combine the time-efficiency advantages of the *a priori* approach with the flexibility of the *a posteriori* approach while maintaining a syntactic distinction between data and propositions. In the example we will focus on, *a posteriori* methods seem to be unusable, as they require too much computer time and resources. Moreover, it seems to be difficult to combine *a posteriori* methods with a module system, which we will later on argue to be desirable, since modules behave as black boxes.

The second issue is where the extracted program will live. Traditionally, efficiency criteria strongly suggested an *external* extraction, in the sense that the proof term is translated from the proof language into a program in a (different) programming language; this also has some other advantages, as programming languages are usually less restrictive and allow among others for potentially non-terminating loops and partial functions. Also, existing technology such as compilers and interfaces can be reused.

Internal extraction (which is a *contradictio in terminis*) has the advantage that the original and final type-systems are the same. This allows a simple implementation as additional reduction rules inside the type theory, such that the proof terms simply reduce to the appropriate programs [8]. Moreover, realizability (extraction) can be used to strengthen results. For instance, the realizability interpretation validates both the axiom of choice and the independence of premise [22,21], so these principles do not have to be assumed as axioms, but can be derived for each particular instance. Finally, in the specific case of Coq, the new version will incorporate a Coq compiler [12]; therefore the speed of the program does not seem to be a real issue anymore.

Our approach

We chose to work with the Coq extraction mechanism to ML, which is an external *a priori* extraction. The main reason for this is simply that the formalization we focused on was already present and had been developed in Coq. It should be pointed out, however, that we expect our work to be quite straightforwardly applicable to the other theories of extraction based on type theory. Also, in our opinion, there is no best method; rather an intelligent combination of *a priori* and *a posteriori* approaches will likely yield the most efficient program.

The reader may wonder why we only focus on the size of the extracted program and not on its efficiency. There are three main reasons for this. First, as we pointed out, size is the main obstacle in the actual extraction. Second, size is an important issue, as it is reasonable to want the extracted programs to be readable; and if two different programs implement the same algorithm in essentially the same way, it sounds reasonable to argue that the shorter one is better. Finally, in the FTA-project the real numbers were formalized as arbitrary Cauchy-sequences, which is computationally very inefficient; therefore analyzing the efficiency of the extracted program seemed uninteresting. These reasons also justify that we do not compare our extracted program with the algorithm which one might directly implement.

The algorithm implicitly present in the Kneser proof is actually as efficient as the well known Newton-Raphson method, as was shown in [11]; unfortunately, the presently inefficient formalization of the real numbers actually prevents the computation of zeros of a polynomial using the proof of the FTA. Fortunately, two efficient formalizations of the reals [6,15] are almost completed.

3 Positive and negative statements

The Coq type theory, based on the Calculus of Inductive Constructions [20], contains two kinds of sorts. On the one hand, there are sorts Set and Prop, representing respectively data types and types of properties; these correspond both to what is usually represented by $*$ in Pure Type System (PTS) [1]. On the other side, there is a family $(\mathsf{Type}_i)_{i\in\mathbb{N}}$ of sorts (corresponding to $\Box$ in the PTS approach) which, among other things, rules how higher types are formed.[3] Also the typing statements $\mathsf{Set} : \mathsf{Type}_0$ and $\mathsf{Prop} : \mathsf{Type}_0$ hold.

The Calculus of Inductive Constructions also provides mechanisms which allow to define inductive types. We will not go into details on how this is done,

[3] Herman Geuvers argued (in private communication) that the datatypes should have sort Type; we will return to this question.

but it is important to point out that the way they deal with inductive types is actually where Set and Prop mostly differ: Set allows stronger elimination rules for inductive types than Prop (for more details, refer to [20]). The reasons for this have to do precisely with not wanting to allow data to depend on properties of other data.

In order to take advantage of this distinction, we classify properties in positive and negative statements, that is, with and without computational content. The former need to be extracted, the latter should not be extracted. This is achieved by representing the first ones by objects of type Set and the latter ones by objects of type Prop.

In fact, we would like to have some overloading which would allow us to treat all propositions (whether typed in Set or in Prop) uniformly; unfortunately, this is not possible in the present version of Coq.

However, the situation is not as bad as it might seem at first glance. Suppose for a moment that we have decided which types the primitive statements have; then, the types of compound statements can be automatically derived as we now explain. We use s, s_1 and s_2 to denote either Set or Prop.

The computational content of the implication is determined by its second argument; this can be seen by taking into account that implication is actually represented (through the Curry-Howard isomorphism) by non-dependent functional abstraction. Therefore, $\rightarrow: s_1 \rightarrow s_2 \rightarrow s_2$.

The negation of a statement does not have computational content, as it simply means that a certain case does not occur. So $\neg : s \rightarrow \mathsf{Prop}$. This is also consistent with the fact that $\neg A$ is usually defined as $A \rightarrow \bot$, which according to the previous rule always has type Prop.

For a disjunction we want to keep track of which option is relevant, in order to define functions by cases. Therefore, $\vee : s_1 \rightarrow s_2 \rightarrow \mathsf{Set}$.

The type of the conjunction is determined by the types of the conjuncts. In case they both have type Prop, neither of them has computational content, and therefore neither does their conjunction; otherwise, the conjunction must have type Set.

Notice that all these maps are already present in the Coq standard library, except for some variants of $\wedge$ and $\neg$. These are straightforward to define.

In summary, the types of logical connectives are:

$$
\begin{aligned}
\neg &: s \rightarrow \mathsf{Prop} \\
\rightarrow &: s_1 \rightarrow s_2 \rightarrow s_2 \\
\vee &: s_1 \rightarrow s_2 \rightarrow \mathsf{Set} \\
\wedge &: s_1 \rightarrow s_2 \rightarrow \begin{cases} \mathsf{Prop} & s_1 = s_2 = \mathsf{Prop} \\ \mathsf{Set} & \text{otherwise} \end{cases}
\end{aligned}
$$

Unfortunately, Coq does not allow overloading, it is not possible to use the standard notations for the new connectives. Therefore, we defined some abbreviations. Conjunction and disjunction are represented respectively by `*` and `+`, with arguments from Prop enclosed in braces. (This notation for disjunction is already defined.) The proposition $\neg A$ is represented by `(Not A)`, while implication is written as usual: `A->B` for $A \to B$.

Finally, we use the quantifiers already defined in Coq. Universal quantification behaves similarly to implication, so $\forall : \Pi(A : s_1).(A \to s_2) \to s_2$. In Coq, `(x:A)(P x)` stands for $\forall_{x:A}(Px)$.

As regards existential quantification, we always need to keep track of its witness, so $\exists : \Pi(A : \mathsf{Set}).(A \to s) \to \mathsf{Set}$. We represent $\exists_{x:A}(Px)$ in Coq by either `{x:A & (P x)}`, if $P : A \to \mathsf{Set}$, or `{x:A | (P x)}`, if $P : A \to \mathsf{Prop}$.

The Coq system also provides a Prop-valued existential quantifier, but we never had occasion to use it in our example.

3.1 Primitive formulas

The only thing now missing is to describe how to type the primitive formulas. Our development is based on the notion of a *setoid*, that is a set together with an apartness relation.[4] An apartness relation is a relation # such that for all x, y and z:

1. $\neg x \# x$;
2. $x \# y \to y \# x$;
3. $x \# y \to x \# z \vee y \# z$.

Usually, for instance in the case of the real numbers, the apartness is computationally meaningful, so apartness has type Set. This can easily be seen from the third axiom where the conclusion is a disjunction; its computational content can only be realizable if the apartness is in Set. This is consistent with what was said above: in the Cauchy model of the real numbers, for example, apartness is defined by means of an existential statement which must be typed in Set.

Our setoids all have a *stable* equality, that is $x = y$ iff $\neg\neg x = y$. So the equality does not have computational content, and should be placed in Prop.

Another way to look at it is to see that the equality can be characterized as the negation of the apartness: $x = y \iff \neg x \# y$, and as such should go to Prop. This is actually a stronger condition than the previous one, as it then follows that

$$x = y \iff \neg x \# y \iff \neg\neg\neg x \# y \iff \neg\neg x = y.$$

Finally, we have an abstract model of the real numbers, in which $<$ is a primitive relation. This relation is computationally meaningful; for instance, in the model of the Cauchy sequences it is defined as an existentially quantified proposition. In abstract, this meaningfulness can be easily seen from the fact that $x < y$ should be equivalent to $x \# y \wedge \neg(y < x)$, and the latter has type Set.

The relation $\geq$ is defined as the negation of $<$, so $\geq$ is of type Prop. (Remark that constructively the relation $x \geq y$ is not the same as either $x = y$ or $x > y$ but weaker, as the latter would give us a way to decide which of the two options is the case.)

[4] These are actually known as setoids *with apartness*, or *constructive setoids*, but we will simply call them setoids as they are the only ones we will refer to.

All other relations are defined from these primitives, so we can easily derive their type. The situation here is reminiscent to model theory, where we study abstract signatures and define more complicated functions and predicates defined from these primitives, but we will not pursue this line of thought.

Using this approach, we found that much more than 90% (!) of the proof terms in the formalization were then assigned the type Prop, which meant that a significant proportion of the formalization would be ignored by the extraction mechanism. Thus, we managed for the first time to extract a program, although still quite large: its size was still 15Mb (roughly equally divided between the construction of the real numbers and the proof of the FTA).

Unfortunately, due to known issues [13], the extracted ML program does not type check. This can be overcome by manually editing the program; in our case, this meant inserting an explicit type cast in around ten thousand places in the extracted program. We understand that this problem will be fixed in the next Coq version, but at the time this work was done it made it impossible to analyze the improvements described in the next section in terms of actual performance.

4 Optimization

Having successfully extracted a program, we got for the first time an opportunity to analyze our formalization from a new perspective and the viability of actually producing usable programs from formal mathematics. In this section we will analyze the reasons that made our extracted program so big and discuss how some apparently trivial modifications lead to impressive changes in its size.

When looking at the program code two main problems stand out:

1. much of the program consists of coercions between algebraic structures; for example, addition is an operation on semigroups, therefore if R is a ring a coercion is needed to cast it into a semigroup so that we can speak of the addition on R.
2. there is still much irrelevant information on the propositional level.

The first problem has to do with the fact that the FTA project attempted at generality, therefore building a cumulative algebraic hierarchy where new structures are built on top of existing ones. Unfortunately, the absence of a notion of subtyping in Coq implies that there must exist explicit casts between these structures instead of the inclusions one would expect. The new Coq version contains a module system; the extraction mechanism is being adapted to work with this modules. Hopefully, this will solve this problem.

The second problem is more interesting, and underlines an unexpected aspect of the formalization. Because of the way primitive predicates are typed, (mathematically) equivalent definitions can generate quite different extracted programs according to how they are typed. Consider the usual statement saying that $(x_n)_{n\in\mathbb{N}}$ is a Cauchy sequence:

$$\forall_{\varepsilon>0}\exists_{N\in\mathbb{N}}\forall_{m,n>N}\,|x_m - x_n| < \varepsilon. \tag{1}$$

Because $<$ is a Set-valued predicate, the extracted proof that x is indeed a Cauchy sequence will consist of a function from the positive real numbers to the natural numbers (which computes the witness N) and a function that takes two numbers bigger than N and returns a computational proof of the desired inequality.

However, the following definition is easily seen to be equivalent:

$$\forall_{\varepsilon>0}\exists_{N\in\mathbb{N}}\forall_{m,n>N}\,|x_m - x_n| \leq \varepsilon. \qquad (2)$$

(Equation (1) trivially implies (2); for the reciprocal, just take $\varepsilon > 0$ and apply (2) with $\frac{\varepsilon}{2}$.)

Now, the computational part of a proof that x satisfies condition (2) consists simply of an algorithm to extract the witness. This is intuitively more efficient, as it involves less computation, and is just as informative as we argued above.

Even if one is not interested in program extraction, there are good mathematical reasons for preferring to use $\leq$ instead of $<$ whenever possible. This is simply because $\leq$, being a negative predicate, allows constructive proofs by contradiction. It should also be noted that Bishop [2] very carefully uses the latter in all ε-δ-definitions and -proofs for this reason.[5]

In summary, there are two good reasons to work with negative statements: not only do the programs become shorter, but also proving is made easier.

As our construction of the real numbers was based on the model of Cauchy sequences of rationals, changing this definition (which by the way did not require changing too much of the formalization) reduced the size of the extracted reals by 80% and the whole ML program to around 8Mb.

4.1 Proof Optimization

Examining the extracted program, we realized that in fact there was still too much unnecessary or redundant information. Specifically, the proof of the Kneser lemma [11], which basically consists of several long chains of inequalities, seemed a good candidate for size reduction, and so we decided to focus on it.

The formalization of the FTA contains several results dealing with order. Among those, there are several transitivity rules for $<$ and $\leq$ and results dealing with preservation through algebraic operations.

The previous experience with the Cauchy sequences suggested that a clever use of the $\leq$ relation could significantly improve the size of the extracted program. To understand this in more detail, we will look at two examples, both of which deal with the use of transitivity.

There are four main transitivity rules with the following types (all depending on an ordered field F and variables $x, y, z : F$):

[5] We actually realized that this was happening when we compared the size of the FTA program with the sizes of extracted programs for Rolle's theorem or Taylor's theorem; these were around 100 times smaller, and one of the main reasons for this was in fact that all ε-δ concepts such as continuity and derivative were defined with $\leq$, following Bishop.

```
less_transitive : (x[<]y)->(y[<]z)->(x[<]z)
less_leEq_trans : (x[<]y)->(y[<=]z)->(x[<]z)
leEq_less_trans : (x[<=]y)->(y[<]z)->(x[<]z)
leEq_transitive : (x[<=]y)->(y[<=]z)->(x[<=]z)
```

The first three are extracted to ML as functions with following types:

```
less_transitive : (x[<]y)->(y[<]z)->(x[<]z)
less_leEq_trans : (x[<]y)->(x[<]z)
leEq_less_trans : (y[<]z)->(x[<]z)
```

(The fact that some conditions disappear does not compromise the correctness of the program, as the extraction mechanism guarantees that they will hold whenever these functions are applied in the extracted program.)

It then becomes clear that the last two lemmas will generate smaller programs, significantly smaller when you take into account that each proof of $a < b$ is typically long. By trying to cut out the bigger branches, significant improvement in the size of the program (and, one would hope, indirectly on its efficiency) can be made.

But we can even do better. Suppose that a statement of the form $a < b$ is proved through the chain of inequalities

$$a < x_1 < x_2 < x_3 < b.$$

A naive proof-term of type `a[<]b` would then be

```
(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_a_x1:a[<]x1` and analogously for the other terms (which in general will be quite large).

Now, because every subterm of this proof has type Set, the extracted program will have exactly the same structure. However, we could also justify the same inequality by stating first that $a \leq x_3$ and then that $x_3 < b$. The corresponding proof term could then be `(leEq_less_trans a x3 b H_a_x3 H_x3_b)`.

Furthermore, `H_a_x3` (which will include most of the terms in the previous proof) has type Prop; therefore, the extracted program is simply

```
(leEq_less_trans a x3 b H_x3_b)
```

Notice that we gain not only by omitting `H_a_x1`, `H_x1_x2` and `H_x2_x3`, but also in not having to mention x_1 and x_2; analysis of the extracted program shows that (e. g. in the Kneser proof) this is also very relevant, as these can also be quite long expressions.

This optimization of the proof of the Kneser lemma reduced the size of the extracted program by 1.5Mb, corresponding to roughly 30% of the size of the proof of the Kneser lemma.

In the next step, we decided to experiment with the known distinction between subsets as propositional functions or as subsetoids [4,5]. We found that when using propositional functions, not only do proofs become easier to write, but also the extracted program greatly reduces in size. It also increases the internal coherence of the formalization, as using this approach we can treat division simply as a partial function and apply to it the general (formalized) theory of partial functions.

This turned out to be a significant improvement; although the extracted real numbers do not change much (which is actually to be expected, as division plays no important role in their formalization), the FTA part is reduced by 60%. More significantly, there is essentially no change in the actual program due to the fact, explained in [5], that we are basically performing explicit $\beta\delta$-reduction on the proof terms (and corresponding extracted programs).

The program size was finally brought down to 3Mb by some minor changes in the proofs, which are too specific to be interesting to describe here.

We summarize these results in a table:

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

It should be pointed out that the *ratios* between the sizes of the different versions are actually more relevant than the actual sizes: we 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 not.

5 Future Optimization

Throughout this section we will only consider the FTA-part of the extracted program.

Although reduced to 20% of its original size, the extracted program is still considered quite large ($1,6$Mb) when one considers that the algorithm which it implements is not so complex. Therefore, we decided that we should carefully look at the program to understand precisely what was taking up so much space.

One of the first things we noticed was that even though the formalization included a constant `CC`, representing the type of complex numbers, this constant was nowhere to be found in the extracted program. This is a bit surprising, as we are after all extracting a function that operates on complex numbers.

As it turned out, in fact, the definition had been fully expanded every time it occurred in the proof term! Considering that the type of complex numbers is explicitly mentioned around 130 times, and that the definition is around 5000 characters long, this fact alone accounts for nearly half of the program code.

The ring of polynomials is another construction which was fully spelled out each time. Although a bit smaller than the previous one, it is still mentioned more than sixty times. Manually replacing these occurrences by a defined constant therefore reduced the program to a mere 300kb, and allowed us also to see that in fact not much more simplification was likely to be possible, as most of the functions now became quite short.[6]

At this point, the need to explicitly insert coercions is the main reason for the size of the program. A good module or subtyping system for Coq would at this stage be very useful. Our experiments suggest that this would allow the extracted program to be as small as 100kb—a factor of 75 when compared with the size of the original extracted program.

A module system also seems to be useful for the following reason: for certain datatypes we will want to have several implementations, each one tailored for a certain goal, be it a specific kind of computation or a convenient way of proving theorems. If we have an adequate module system, these different implementations can be used in harmony. Work in this area is being done [7] and will probably be included in the next Coq version.

In summary, the extracted program consists of:

Description	Size (kb)	% of total
"Relevant" code	110	6.5
Unfolding of $\mathbb{C}$	1050	62.5
Unfolding of polynomials ($R[x]$)	330	19.5
Coercions	190	11.5
Total	1680	100

As before, the relative sizes are more important than the absolute values; that is, in the earlier versions of the extracted program the relative amount of wrongly unfolded definitions and the relative amount of coercions were roughly the same as in the final version.

6 A more abstract view

Above we explained how to use the sorts Set and Prop for the propositions. To do this we only need to decide where to place the primitive relations, the proper place for composed relations can then be derived.

However, the use of Set is a bit unsatisfactory. For example, it is not possible just by looking at the type of an object to know whether it represents data or a proposition; this is felt as a limitation, as it makes the task harder for automatic tools to interpret the contents of the library.[7] We have tried to hide this ambiguity by defining a new constant CProp (Contentful Propositions) as an alias

[6] This turned out to be a bug in the extraction mechanism. After mentioning it to Pierre Letouzey, he was able to identify and fix the problem, thereby reducing the size of all extracted programs by around 80%.

[7] This was quite actively discussed on the MoWGLI mailing list.

for Set, and then explicitly typing all propositional statements in CProp. However, this approach is only partially successful for two reasons: on the one hand, in some instances Coq actually requires a *sort* to be used, and CProp, though δ-convertible to a sort, is not one; this is a secondary issue that is expected to be solved in the next version of Coq. The second problem, which is more basic, is that the distinction between data types and propositions strongly depends on the user's discipline: because Set and CProp are δ-equivalent, they can be interchanged from the type theory point of view, therefore reinstating the ambiguity which we aimed at removing.

The use of Set also has serious limitations from the logical point of view. We would like to allow the addition of e.g. the axiom of the excluded middle to our formalization and build classical mathematics on that, but this is not possible for several reasons. First, it is not clear how the principle of excluded middle should (or even could) be written: because of the fact that propositions do not all have the same type there are at least four different (and not equivalent) ways to write it. Moreover, the axiom $\Pi_{A:\mathsf{Set}} A \vee \neg A$ leads to an inconsistent theory, as noted in [9,17].

With these issues in mind, we now propose a slightly modified version of the Coq type system where these problems do not arise.

Our previous discussion of the difference between positive and negative statements suggests that there should be two basic sorts for propositions. We will denote these by Prop^+ and Prop^-, with the obvious meaning. Prop is then defined to be the disjoint union of these two sorts: $\mathsf{Prop} := \mathsf{Prop}^+ \oplus \mathsf{Prop}^-$. In Figure 1 we represent the current and the proposed version of the Coq type hierarchy.[8]

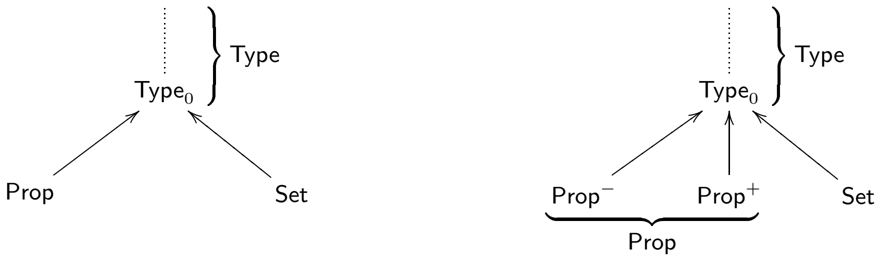

Fig. 1. Type hierarchy in Coq: present (left) and proposed (right)

In order to do classical mathematics, Prop^+ also needs to be predicative.[9] On the other hand, Prop^- should enjoy proof irrelevance, which is a natural

[8] Herman Geuvers argued (on the MoWGLI mailing list) that the datatypes should have sort Type. This has the advantage that the logic one obtains is very similar to higher order logic, which has been thoroughly studied and is by now well-understood. In this approach, the sort Set is no longer needed; however, some additional complications arise, so we will not discuss this approach.

[9] This is the main reason why one cannot simply work within Set.

consequence of the non-informativeness of the proofs—this is actually in line with the plans of the Coq team [7].

Intuitively, Prop^+ and Prop^- behave respectively as Set and Prop in the Coq type theory, the main differences residing in the predicativeness of Prop^+ and the proof irrelevance in Prop^-. With this in mind, it is then natural to define connectives in Prop following what was said in section 3:

$$
\begin{array}{ll}
\rightarrow : \mathsf{Prop} \rightarrow s \rightarrow s & \forall : \Pi(A : t_\forall).(A \rightarrow s) \rightarrow s \\
\vee : \mathsf{Prop} \rightarrow \mathsf{Prop} \rightarrow \mathsf{Prop}^+ & \exists : \Pi(A : t_\exists).(A \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop}^+ \\
\underline{\vee} : \mathsf{Prop} \rightarrow \mathsf{Prop} \rightarrow \mathsf{Prop}^- & \underline{\exists} : \Pi(A : t_\exists).(A \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop}^- \\
\wedge : s_1 \rightarrow s_2 \rightarrow \begin{cases} \mathsf{Prop}^- & s_1 = s_2 = \mathsf{Prop}^- \\ \mathsf{Prop}^+ & \text{otherwise} \end{cases} & \neg : \mathsf{Prop} \rightarrow \mathsf{Prop}^-
\end{array}
$$

where $\{s, s_1, s_2\}$ denote either Prop^+ or Prop^-. As regards quantifiers, $t_\forall$ can be a type of propositions or a datatype, and $t_\exists$ is a generic datatype.

Notice the existence of two disjunctions and two existential quantifiers. The connectives $\vee$ and $\exists$ are the informative connectives previously described, whereas their underlined versions are non-informative. These connectives are all present in the current Coq implementation, and we will soon discuss how $\underline{\vee}$ and $\underline{\exists}$ can be used.

Also observe that from the user point of view all these connectives return objects of type Prop; that is, we achieve uniformity in the types of propositions as desired.

As regards classical reasoning, there are two versions of the Principle of Excluded Middle (PEM) which can safely be added—corresponding to the two available disjunctions.

The weaker version is the axiom

$$\Pi_{A:\mathsf{Prop}} A \mathbin{\underline{\vee}} (\neg A) : \mathsf{Prop}.$$

This is a *non-informative* version of the PEM, which allows classical reasoning over the domain but not e.g. defining functions according to whether a given predicate holds. With this axiom it is possible to prove classically valid properties of constructive formalizations, while keeping the possibility of program extraction. At this stage the connective $\underline{\exists}$ might come in handy, as it allows us to construct non-informative existential proofs by reasoning classically.

An alternative way to achieve the same effect is to *define* the weak disjunction and existential as follows: $A \mathbin{\underline{\vee}} B := \neg(\neg A \wedge \neg B)$ and $\underline{\exists} x : A.Px := \neg(\forall x : A.\neg(Px))$. Then the above axiom is constructively provable.

To obtain full classical mathematics, in which we can define functions by case distinction, the stronger axiom

$$\Pi_{A:\mathsf{Prop}} A \vee (\neg A)$$

is needed. However, this poses a more subtle typing issue. Because of the predicativity of Prop^+, it cannot have type Prop^+; however, it can safely be typed in a higher type (like the Coq Type type). Although this may at first look somewhat strange, it can be regarded as a natural consequence of the fact that it represents a scheme of axioms rather than a single axiom. Given any $A : \mathsf{Prop}$, we can then prove $A \vee (\neg A) : \mathsf{Prop}$. However, adding this axiom obviously destroys program extraction.

6.1 Properties of Prop^+ and Prop^-

The approach just described has many similarities with the marked types of [18]. However, there is one very important distinction, namely that propositions are assigned types *a priori* and these propagate outwards through the formalization, whereas in [18] the markings propagate inwards from the type of the final term. This means that our approach requires no extra analysis of the proof term at extraction time.

Another issue is the relation between Prop^+ and Prop^-. It is natural to define maps between these types in both directions, with the following motivations:

– $(\cdot)^+ : \mathsf{Prop}^- \to \mathsf{Prop}^+$ represents the intuitive notion that any non-informative proposition can be viewed as informative with empty content (e.g., as a constant);
– $(\cdot)^- : \mathsf{Prop}^+ \to \mathsf{Prop}^-$ is a forgetful map that forgets all information associated with the proof.

These maps can be (and have been) implemented as inductive types, meaning that the only way to get a proof of P^+ (respectively P^-) is as the image of a proof of P.

These functions are important to allow uniform treatment of propositional concepts. Consider for example the formalization of partial function on a datatype A as a pair $\langle P, f\rangle$, where $P : A \to \mathsf{Prop}^+$ and $f : \Pi_{x:A}(Px) \to A$; that is, P is a computationally relevant predicate on A and f is defined on elements of A that satisfy P (and its output can eventually depend on the proof term). We would also like the situation where P is non-informative to fit into this definition; this can be achieved by using P^+ instead of P (and hopefully the system can be induced to insert $(\cdot)^+$ automatically by means e.g. of a coercion mechanism).

On the other hand, $(\cdot)^-$ can be used to mark informative terms as irrelevant in specific situations. Suppose that one wants to define a function (to be extracted) whose specification is a positive predicate P, but the function will actually never be used in subsequent work. Then it is natural to specify it using P^- instead of P, as the extracted result will be smaller.

This map also has the important property of being preserved through connectives; that is, it can be proved uniformly on A and B that e.g. $(A \wedge B)^- \to (A^- \wedge B^-)$ and similarly for the other connectives (replacing $\vee$ and $\exists$ by their weaker counterparts $\underline{\vee}$ and $\underline{\exists}$). This is trivially not true of $(\cdot)^+$, because of the lack of information in Prop^-.[10]

The map $(\cdot)^-$ has already been proposed by [18], although in a slightly different setting, and its main properties analyzed and discussed.

[10] This difference comes from the unavailability of some elimination rules for inductive types.

7 Conclusions

We have pointed out how to greatly reduce the size of extracted programs and presented a general guideline for developing constructive proofs in order to make extraction possible. We have also proposed some modifications to the Coq type theory that we feel will make constructive formalizations somewhat easier; at the same time, we show how these modifications make it possible to safely add the principle of the excluded middle to constructive formalizations.

Some of these ideas have been discussed with the Coq team, and have helped to improve the Coq extraction mechanism, which is still under development. As a side remark, we would like to mention that after the correction of the bug described in Section 5, it became possible to extract a program from the *original* formalization of the FTA; this program turned out to be around 20 times larger than the one originally obtained using the Set/Prop distinction we described, thus confirming our claim that this distinction allowed around 90% of the code to be removed. We would also like to note that the huge resources needed to extract this program (around 2Gb RAM memory) make the use of a method such as ours not only convenient, but indeed necessary for large program developments. Finally, we hope that it will be possible to change the Coq type theory in the near future in a way similar to what we have described.

The final version of the formalization we discussed, together with the extracted program, can be found on C-CoRN repository[11] together with a short file which only contains the basic material needed for the development of other formalizations along the lines we described.

After this paper was submitted considerable progress was made on the work we described; as this is too extensive to include here, we will report on it in due time.

Acknowledgments

Support for the work of first author was provided by the Portuguese Fundação para a Ciência e Tecnologia, under grant SFRH / BD / 4926 / 2001 and by the FCT and FEDER via CLC.

The authors wish to thank Jean-Christophe Filliâtre and Pierre Letouzey for discussions on the Coq extraction mechanism; and Henk Barendregt, Herman Geuvers, Dan Synek and Freek Wiedijk for their suggestions, ideas and comments.

References

1. H. P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science, Vol. 2*, pages 117–309. Oxford Univ. Press, New York, 1992.
2. Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill Book Company, 1967.

[11] http://www.cs.kun.nl/fnds/ccorn.html

3. Errett Bishop. Mathematics as a numerical language. In *Intuitionism and Proof Theory (Proceedings of the summer Conference at Buffalo, N.Y., 1968)*, pages 53–71. North-Holland, Amsterdam, 1970.
4. Venanzio Capretta. *Abstraction and Computation*. PhD thesis, University of Nijmegen, 2002.
5. Jesper Carlström. Subsets, quotients and partial functions in martin-löf's type theory. In *Proceedings of the TYPES Conference 2002, to appear*, LNCS. Springer-Verlag, 2003.
6. Alberto Ciaffaglione and Pietro Di Gianantonio. *A co-inductive approach to real numbers*, volume 1956 of *LNCS*. Springer-Verlag, 2000.
7. Judicaël Courant. MC2: A module calculus for pure type systems. Technical Report 1292, LRI, September 2001.
8. Maribel Fernández, Ian Mackie, Paula Severi, and Nora Szasz. A uniform approach to program extraction: Pure type systems with ultra σ-types. `http://www.cmat.edu.uy/~severi/publications.html`
9. Herman Geuvers. Inconsistency of classical logic in type theory. `http://www.cs.kun.nl/~herman/note.ps.gz`
10. Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. The algebraic hierarchy of the FTA Project. In Sebasitani S. Linton, editor, *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, pages 271–286. Elsevier, 2002.
11. Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the Fundamental Theorem of Algebra without using the rationals. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, Proceedings of the International Workshop, TYPES 2000*, volume 2277 of *LNCS*, pages 96–111. Springer, 2001.
12. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings ICFP'02*.
13. Pierre Letouzey. A new extraction for Coq. In *Proceedings of the TYPES Conference 2002, to appear*, LNCS. Springer-Verlag, 2003.
14. Per Martin-Löf. Constructive mathematics and computer science. In *Logic, Methodology and the Philosophy of Science VI*, pages 153–175. North-Holland, 1982.
15. Milad Niqui. Exact arithmetic on Stern-Brocot tree. 2003. submitted.
16. C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calulus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, 1989. ACM.
17. Loïc Pottier. Quotients dans le CCI. Technical Report RR-4053, INRIA, November 2000. http://www-sop.inria.fr/rapports/sophia/RR-4053.html.
18. Frédéric Prost. Marking techniques for extraction. Technical Report 95-47, Laboratoire de l'informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1995.
19. H. Schwichtenberg. Minimal logic for computable functionals. Technical report, Mathematisches Institut der Universität München, 2002.
20. The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.3*. INRIA-Rocquencourt, 2002.
21. A.S. Troelstra. *Realizability*, volume Handbook of Proof Theory, pages 407–473. North-Holland, 1998.
22. A.S. Troelstra and D. van Dalen. *Constructivism in mathematics. An introduction.* Number 123 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.