A Large-Scale Experiment in Executing Extracted Programs

Luís Cruz-Filipe, Pierre Letouzey

▶ **To cite this version:**

Luís Cruz-Filipe, Pierre Letouzey. A Large-Scale Experiment in Executing Extracted Programs. Calculemus 2005, Mar 2006, Newcastle upon Tyne, United Kingdom. pp.75-91, 10.1016/j.entcs.2005.11.024 . hal-00150908

HAL Id: hal-00150908
https://hal.science/hal-00150908

Submitted on 1 Jun 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Large-Scale Experiment in Executing Extracted Programs

Luís Cruz-Filipe [1]

CLC, Lisbon, Portugal

Pierre Letouzey [2]

Mathematisches Institut, LMU Munich, Germany

Abstract

It is a well-known fact that algorithms are often hidden inside mathematical proofs. If these proofs are formalized inside a proof assistant, then a mechanism called extraction can generate the corresponding programs automatically. Previous work has focused on the dificulties in obtaining a program from a formalization of the Fundamental Theorem of Algebra inside the Coq proof assistant. In theory, this program allows one to compute approximations of roots of polynomials. However, as we show in this work, there is currently a big gap between theory and practice. We study the complexity of the extracted program and analyze the reasons of its inefficiency, showing that this is a direct consequence of the approach used throughout the formalization.

Key words: Program extraction, Complexity, Constructive reals, Fundamental Theorem of Algebra, Formalization of Mathematics.

1 Introduction

Several approaches can be used for certifying software. A first one, perhaps the most natural, is to start with an handwritten program and then inspect it formally in a suitable logical system, like Hoare logic. But there exists an alternative approach where one needs not write the program, but rather obtains it automatically from a mathematical proof. This automatic transformation of proofs into correct-by-construction programs is called *(program) extraction*.

The ability of extracting programs from proofs is an instance of the Curry–Howard isomorphism, which explains in particular that a constructive proof is

[1] Email:lcfilipe@gmail.com

[2] Email:letouzey@lri.fr

©2005 Published by Elsevier Science B. V.

isomorphic to a functional program. In practice, though, extraction is not just a mere application of this isomorphism. In the case of proof assistants based on Type Theory (like Coq [20] or NuPrl [12]), the internal representation of proofs is an actual λ-term, so a proof *is* a functional program; but in practice large portions of these proofs correspond to logical justifications and are irrelevant from a computational point of view. In order to obtain realistic programs, one of the main tasks of the extraction is to identify and remove such parts.

For more than fifteen years, extraction has been both studied theoretically and incorporated in several proof assistants, like PX [11], NuPrl [12], Minlog [8], Isabelle [3] or Coq [17,18,13,14]. As a consequence of these various works, the correction of extraction is now generally well understood and this mechanism has become a framework of choice for certifying functional programs Moreover, this framework is quite elegant. In particular, it allows one to unify constructive mathematics and certified functional programming.

On the other hand, evaluating and/or improving the complexity of the extracted programs is still an active research subject. Automatic analysis of extracted program complexity has been studied in NuPrl [2]. Also worth mention are some recent efforts to determine which kind of restrictions on proofs can ensure that the final extracted program is polynomial in time [1].

Most of the experiments done so far with program extraction deal with examples from Computer Science. In this paper we focus on a different problem: the computational behaviour of a program extracted from a formalization of a mathematical statement. As we will show, this example raises quite different problems than the ones previously addressed, due both to its different nature and to its size, and the results are not so satisfactory.

The work described in this paper was partially included in the authors' PhD theses [5] and [14].

1.1 The C-CoRN library

In this experiment we work with the Coq proof assistant. More concretely, we study the behavior of its extraction mechanism when applied to a constructive proof of the Fundamental Theorem of Algebra [10], part of the C-CoRN library [6].

C-CoRN is a library of mathematics formalized in the proof assistant Coq [20] that has been in development since 1999. At present, it has achieved a considerable size, making it an appropriate environment for a large-scale experiment.

C-CoRN ranges over different subjects, including Algebra, Real Analysis and General Topology. From the perspective of program extraction, most of these are not very interesting to look at, since they deal with results with little or no computational content. Therefore the focus was set on two specific parts of the library: the constructive proof of the Fundamental Theorem of Algebra, described in [10], and the model of $\mathbb{R}$ as Cauchy sequences of rationals, whose

formalization is the subject of [9].

The Fundamental Theorem of Algebra (FTA) states that every non-constant polynomial over the complex numbers has a root. The proof of this statement in C-CoRN is a constructive proof that embodies the Newton–Raphson root-finding algorithm (see [10]). For this reason, the problem of extracting a program from this proof is not only a non-trivial case study, but also one that is computationally interesting.

1.2 Extraction from C-CoRN

Applying the Coq mechanism of extraction to C-CoRN was not easy. One of the reasons for this was the primitive state of the extraction mechanism at the time of the first attempts. Another reason was the size of the library, several orders of magnitude larger than any test done earlier. Finally, C-CoRN was not originally developed with extraction in mind, and this turned out to have important consequences as pointed out in [7]: obtaining a program of a reasonable size required rethinking the use of the Coq sorts in a constructive formalization and fine-tuning several proofs and definitions in the original proof of the FTA.

All the work done so far succeeded in *extracting* a program of reasonable size, but at the time it was not possible to execute the extracted programs and study their computational behaviour. The present paper builds on that work by studying what happens when one tries to compile and execute the root-finding program extracted from the formalization of the FTA in C-CoRN.

Section 2 discusses some issues encountered while compiling the extracted program and how they were solved. It then turned out that the compiled program was too slow and did not produce any output when executed, so Sections 3 and 4 focus on two smaller examples that are simple enough for one to understand the extracted code, while at the same time exhibiting essential problems of the larger program. In Section 5 we present a different formalization of the same theorem, developed by the second author together with H. Schwichtenberg; this formalization differs fundamentally from C-CoRN because it was done with extraction in mind based upon [19]. In Section 6 we compare the programs extracted from both formalizations and show that the problems identified earlier are fundamental issues; as a consequence, there is little hope that extraction from C-CoRN will ever produce an executable (in practice) program.

2 Compilation of the extracted program

Compiling the source code extracted from the proof of the FTA in C-CoRN into a program and running it was not done in [7]. In this section we address some technical issues that had to be solved before this could be done.

The goal of extraction is to produce code for real programming languages;

in the case of Coq, these are Ocaml and Haskell. Before 2003 the Coq extraction mechanism was only able to produce extracted code for FTA that failed to compile with usual Ocaml or Haskell compilers. This was due to the extreme richness of the type system of Coq, which allows features with no counterpart in ML-like type systems (for example, the definition of types depending on terms). The extracted code will therefore not pass the type-checking stage performed prior to the actual compilation. At the same time, though, the correctness of the extraction guarantees that this extracted code, albeit ill-typed, will always evaluate correctly.

Solving these problems was one of the main motivations for the recent redesign of Coq extraction [14]. This has been done via the use of unsafe type-changing primitives like `Obj.magic` in Ocaml. On the computational level, this function is transparent: its argument is returned unchanged. But it has a dramatic effect on typing: `Obj.magic` has type `'a → 'b`, so using it amounts to bypassing the type-checker locally.

Using `Obj.magic` everywhere is not satisfactory for several reasons. First, the readability of the extracted code is decreased; secondly, some optimizations of the compiler may be disabled; finally, the additional correctness properties coming from the ML typing are lost, requiring one to rely strongly on the correctness results of the extraction. For all these reasons, the extraction only inserts `Obj.magic` in the extracted code precisely where untypable expressions would occur. The details of this insertion mechanism are described in Chapter 3 of [14].

The following example is a typical situation in C-CoRN where untypable yet correct code is produced. Using the type `Monoid` for the mathematical structure of monoids, a group is defined in the following way.

```
Record Group : Type := {
  mon   : Monoid;
  inv   : mon -> mon;
  proof : Is_Group mon inv }
```

In other words, a group is composed of a monoid `mon` and an inverse function `inv` fulfilling some logical requirements. This record is quite different from an Ocaml record: the field `mon` is a type, used in the type of the second field `inv`, whereas the third field contains some proofs. During extraction, the first field is removed since a type cannot be expressed in ML on the object level, while the type of the second field is changed to something acceptable using `Obj.magic`.

The extracted code from the proof of the FTA contains around 400 occurrences of `Obj.magic` arising from similar situations. This code is accepted by the Ocaml compiler without any additional modification.

The same mechanism for ensuring the typability of extracted code artificially has been adapted to extraction to Haskell. Here `Obj.magic` is named `unsafeCoerce`, which is present in at least two of the main Haskell imple-

mentations, namely GHC and Hugs. Using these unsafe typing coercions, we were also able to extract Haskell code accepted by these two compilers and to compare the performance of these programs with the ones coming from the Ocaml extraction. However, this extraction of artificially-typed Haskell code should be considered as experimental: even if it seems to work in practice, we are still checking the lack of unexpected consequences.

3 Computing e

After the compilation issues discussed in the previous section were solved, we were able for the first time to execute the program extracted from the proof of the FTA. Therefore, we defined $x^2 - 2$ as a polynomial in the complex plane, proved it to be non-constant, extracted this proof to Ocaml, and gave this as argument to the FTA program. After one week, the program had not yet finished executing.

Unfortunately, this program was too complex to examine directly and understand why it was taking so long running. Instead, we decided to focus our attention on simpler examples from C-CoRN that we could examine and from these gain some insight into the broader picture.

In this section we focus on one of these simpler examples: the computation of e. Although its value is not relevant for the proof of the FTA, e is one of the simplest examples of real numbers in C-CoRN, in the sense that it is defined as the limit of a sequence of rationals.

3.1 Improving the extracted program

In C-CoRN, real numbers are defined as Cauchy sequences of rationals, see [9]. In particular, e is defined as the sum of the series $\sum_{n=0}^{\infty} \frac{1}{n!}$, which unfolds to the limit of the sequence of partial sums of $\lambda n.1/n!$; as a consequence, e is represented by that same sequence of partial sums (seen as a sequence of rationals).

The corresponding extracted program is a function that takes a natural number n as argument and returns the nth element in the Cauchy sequence representing 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. The problem was easy to find: the sequence converging to e is formalized using the natural numbers of Coq, which are the Peano numerals, meaning that to compute the nth approximation of e one first computes $k!$ for $0 \leq k \leq n$ as the mth successor of 0 for some m, translates the result to the real number $1 + 1 + \ldots + 1$ (m times), computes its reciprocal and then adds everything together. Furthermore, the proof of $k! \neq 0$ is also needed to compute this reciprocal, and this unfolds again to a computation of a factorial in unary notation.

Once this problem was identified, it was easy to fix: instead of computing factorials in $\mathbb{N}$ and injecting the result in $\mathbb{R}$, we defined directly the factorial in an arbitrary ring using its multiplication. Next, we proved by induction that this function is always non-zero, yielding a proof term linear in n. These changes allowed us to compute the first 15 approximations of e, however the execution time increased roughly ten-fold at each extra iteration. Some *ad hoc* improvements both to the definition of the injection of $\mathbb{N}$ in $\mathbb{R}$ and to the model of $\mathbb{R}$ itself helped bring this factor to around 2. This was still far from satisfying, as it meant that only around twenty approximations of e could be computed. Furthermore, profiling revealed that most of the time the program was computing values of $\lambda x.0$ for larger and larger values of x, which seemed to be a consequence of the abstract way in which the real numbers were defined in C-CoRN. This issue will be referred to again in Section 5.

Actually, it is awkward to devote so many efforts in order to compute a real number like $\frac{1}{n!}$. After all, this number is only a rational number, so a more natural thing to do would be to make this computation directly in the structure of rational numbers and inject the result into the reals later.

This cannot be done directly, since C-CoRN enforces a strong separation between an abstract structure of real numbers and any concrete counterpart to it. All parts of this development using real numbers, like the proof of FTA itself, are generic: they rely on one axiom stating the existence of a real number structure `IR`, and this structure is just known to be an archimedian ordered field where all Cauchy sequences admit a limit. Accordingly, we can access nothing but a minimal set of primitive objects and basic properties, according to the underlying structures.

In particular, the only known primitive reals are 0 and 1, which are respectively the units of the underlying structures of semi-group and ring. To refer to a rational number in `IR`, one must build it from 0, 1 and the operations $+$, $*$ and $/$, this last operation requiring in addition a proof of non-nullity as third argument. Such a proof is normally deduced from a proof of strict positivity (or negativity), which must be built from a restricted core of basic properties. In the case of the strict order $<$ over `IR`, these properties are: antisymmetry; transitivity; compatibility with addition and multiplication; and the property $x \neq y \leftrightarrow x < y \vee y < x$. Any other property such as $0 < 1$ is not primitive, but derived from the ones above. The same holds for more complex proofs like $0 < n!$; as a conclusion, building $\frac{1}{n!}$ in this setting, even with the best optimizations, cannot be done via a simple injection from the rationals to the reals.

On the other hand, C-CoRN also provides the concrete construction of a real number structure `Concrete_R` referred to above, ensuring that this axiomatization of the reals is coherent. A rational q can be immediately injected into `Concrete_R` via the Cauchy sequence with constant value `q`. In the same way, a proof of an inequality $a < b$ is straightforward in `Concrete_R`, since we can now access the *definition* of $<$: there should exist a strictly positive ratio-

nal Δ and a rank N such that, in the two Cauchy sequences being compared, all terms of rank greater than N are always apart by at least Δ. In particular, for two rationals $q < q'$ we immediately obtain $q < q'$ in `Concrete_R` by taking $\Delta = q' - q$ and $N = 0$.

From a mathematical point of view, this separation between abstract and concrete reals is legitimate, since it ensures that a proof made at the abstract level is independent of the particular representation selected as concrete model of the real numbers: if one changes this concrete model later the proofs will still hold. From a programming point of view, though, we are here in front of two modules interacting via a minimalist interface that frequently obliges the upper level module to reinvent the wheel, moreover in an ineffective way. The situation is similar to an integer arithmetic module whose interface would not export the multiplication, under the justification that it can be simulated by repeated additions.

To confirm that this distinction between the concrete and abstract levels constituted indeed a bottleneck for program extraction, we added to the abstract level a few axioms, namely the existence of a factorial function and the fact that the output of this function is always positive. Then we proved straightforwardly that these axioms hold in `Concrete_R`. Finally, we instructed the extraction mechanism to replace every use of the axioms at the abstract level by the extraction of the concrete proofs.

3.2 Performance of the extracted program

After having implemented all these changes, the efficiency of the extracted program improved significantly. It still runs in exponential time, which is a consequence of having to compute factorials in the naive way, but instead of doubling at each iteration the execution time only increases by around 40% for each 10 extra steps. A sample of the exact execution times on a P-IV 2.4Ghz with 500Mb RAM is given in Figure 1.

This means that we can compute the 100th iteration in just over one minute, and the 140th (which produces 241 correct digits) in less than five minutes. Also significant is the fact that the growth of the precision is superlinear (the error of the nth approximation is smaller than $1/n!$), so the time needed to compute n correct digits grows somewhat slower (though not much).

Surprisingly, it does not seem to make much difference whether one extracts to Ocaml or to Haskell. Although the Haskell program returns a value much quicker when just a few iterations are required, after 100 iterations the execution time is already half of that of the Ocaml program, and from 125 iterations it actually starts taking longer. Also, the Haskell program consumes huge amounts of memory (several Mb for 140 iterations), while the Ocaml program requires a fixed 4Kb. Apparently this is due to the little amount of dead code in the extracted program (see [7]), which favours call-by-value

execution.

Finally, we compared the performance of our programs with that of a computer algebra system. For this, we implemented the same algorithm to compute e in Mathematica [21] and looked at the execution times. These tests were done on a machine running at about a third of the speed (Pentium at 900MHz running Windows 2000, 384Mb RAM), but even so using Mathematica is amazingly faster: in Figure 1, the results concerning Mathematica are not in seconds, but in milliseconds (averaged over 1000 runs). Still we find these results encouraging: the ratio between the execution times of the Ocaml program and the equivalent Mathematica program is linear on the number of iterations. Asymptotically, the complexities of the extracted programs are not so bad, even if a huge practical constant factor separates them from the complexity of the Mathematica code.

Iterations	Ocaml(s)	Haskell(s)	Mathematica(ms)	Ocaml / Math.
20	0.11	0.01	0.070	1.57143
40	1.82	0.22	0.411	4.42822
60	8.99	1.95	1.302	6.90476
80	28.28	10.00	2.924	9.67168
100	68.77	47.48	5.778	11.902
120	147.80	151.53	10.295	14.3565
140	285.66	375.01	16.624	17.2836

Fig. 1. Execution times of programs computing e

4 Computing $\sqrt{2}$

After the encouraging results described in the previous section, we looked again into the program extracted from the FTA proof. As was said earlier, the full program in itself was too complicated to analyze directly. Therefore, we decided to focus on one subroutine that was being used over and over again, namely the function that computes square roots of real numbers using the Intermediate Value Theorem (IVT) for real-valued polynomials. In this section we will show that this situation is totally unlike the previous, and this in turn has dramatic impact on the full FTA program.

Throughout this section we will focus on the IVT applied to the polynomial $x^2 - 2$, which computes $\sqrt{2}$.

4.1 The Intermediate Value Theorem

Computing the first non-trivial approximation of $\sqrt{2}$ using the IVT took over 52 hours. The algorithm being implemented was again quite simple, being a constructivization of the bisection root-finding algorithm. Classically, one begins with an interval $[a, b]$ with $f(a)f(b) < 0$ and at each step changes either a or b to the midpoint of the interval, according to whether $f(a)f((a+b)/2) >$

0. Constructively this cannot be done, since we cannot decide on the sign of an arbitrary real number; thus the algorithm must first find the point c where the interval should be split.

There are several constructivizations of the IVT, and the one originally formalized in C-CoRN was the most general one. To prove that a function f satisfies the IVT it is sufficient to assume that, for any given y and interval I, there is always a point $x \in I$ such that $f(x) \neq y$. Using this, the above proof can be adapted as follows: instead of taking the midpoint of $[a, b]$, choose a point c in $[(2a + b)/3, (a + 2b)/3)]$ such that $f(c) \neq 0$; then we can decide whether $f(a)f(c) > 0$ or $f(a)f(c) < 0$ and proceed as above. The convergence is slower than in the classical case, since the length of the interval now only shrinks by one-third, but this is not very significant.

Profiling reveals that most of the execution time is spent on the proof that any polynomial satisfies the above property. This proof proceeded by rewriting each polynomial in a canonical form, factorizing it through its value on $n+1$ points on the given interval (n being an upper bound on the degree of the polynomial), and then repeatedly applying extensionality of the algebraic operations: if a sum or product is not zero, then one of the summands or factors must also be non-zero. In this way, the expansion of $x^2 - 2$ on the interval $[0, 3]$ is

$$-\frac{23}{16}\left(\frac{x-\frac{3}{2}}{-\frac{3}{4}}\right)\left(\frac{x-\frac{9}{4}}{-\frac{3}{2}}\right) + \frac{1}{4}\left(\frac{x-\frac{3}{4}}{\frac{3}{4}}\right)\left(\frac{x-\frac{9}{4}}{-\frac{3}{4}}\right) + \frac{49}{16}\left(\frac{x-\frac{3}{4}}{\frac{3}{2}}\right)\left(\frac{x-\frac{6}{4}}{\frac{3}{4}}\right)$$

and it is no longer such a surprise that the program is inefficient.

However, there are much better constructive variants of the IVT which are much better suited for the computation of roots of real numbers. In particular, since polynomials of the form $x^n - c$ (needed to compute $\sqrt[n]{c}$) are increasing, we can use the IVT for increasing functions: if f is increasing on $[a, b]$, then we know that $f((2a + b)/3) < f((a + 2b)/3)$, and hence either $f((2a + b)/3) > 0$ or $f((a + 2b)/3) < 0$ by the properties of the $<$ relation (in particular, *co-transitivity* of $<$).

The program extracted from this variant of the IVT is amazingly different: it produces the second approximation of $\sqrt{2}$ in just over 16 seconds, which were brought down to around 5 seconds by some rewriting of proofs, similar to what had been done for e. Computing each approximation, however, takes roughly thrice the time of the previous one.

The next step was to follow the route that had proved successfull to compute e: isolate the relevant proof-term that is being evaluated, parameterize the proof of the IVT on that term and build a direct proof on the model. In this case, the only proof term that is being repeatedly evaluated is the proof that $x > y \to x^n - c > y^n - c$ (for $x, y > 0$, $c >= 0$ and $n \geq 1$). Changing this proof term was quite straightforward, since it was just a question of adapting the work done for e. The results were however nothing like before: though the computation time required for the first iteration dropped dramatically, it still trebled at each step.

Close examination of what is happening does not help much. Profiling shows that most of the time is spent adding and multiplying (binary) integers. Although part of this may be a consequence of the concrete model we are working with, and it is sure worthwhile to experiment with other models, the bottleneck seems to be the abstract approach used throughout C-CoRN. Because many of the relevant results used in the proof of the IVT are stated for arbitrary rings, they yield relatively inefficient programs when extracted.

We will return to these issues in Section 6 and in the conclusions.

4.2 Performance of the extracted program

The last version of the program was extracted to both Ocaml and Haskell. This time, the difference in execution time is striking: the Haskell program runs about 250 times quicker. However, the memory consumption more than doubles at each step, which is unsustainable: the 12th approximation of $\sqrt{2}$ requires more than 2Gb of RAM memory, making it the limit of what we can compute in this way, and it gives but three correct digits. When the same algorithm was implemented in Mathematica, its performance was not in any way comparable with that of the extracted programs: it computes (predictably) in linear time. The precise execution times are given in Figure 2.

Iterations	Ocaml(s)	Haskell(s)	Mathematica(ms)	Ocaml / Haskell
2	0.13	<0.01	0.211	—
4	1.42	0.01	0.341	142
6	14.31	0.06	0.551	239
8	165.76	0.66	0.711	251
10	2008.00	8.34	0.891	241

Fig. 2. Execution times of programs computing $\sqrt{2}$

5 An alternative approach

The previous section shows that, even after important optimizations, the extracted program could compete in speed with unsafe, manually-written programs. In this section we analyze how specific to C-CoRN this conclusion is, and if we can derive more general lessons about extraction from it. For this purpose, we now describe another Coq formalization of Constructive Real Analysis, developed with extraction in mind.

This formalization was realized in collaboration with H. Schwichtenberg, who was willing to implement part of his lectures notes [19]. This small and quickly-made development was not meant to be a new C-CoRN; rather, it just focused on one particular problem, the computation of $\sqrt{2}$, and tackled it by considering constructive reals immediately from the point of view of extraction, unlike the FTA, where this idea of extraction came only *a posteriori*.

With this approach, we extracted a program that can compute an approximation of $\sqrt{2}$ with more than 40 correct digits in less than 3 minutes. Furthermore, this program uses the very same principle as the one extracted from C-CoRN, namely seeking a root of $x^2 - 2$.

It should be stressed that this formalization is not complete: while focusing on obtaining an extracted program, we have ignored several logical justifications and posed them as axioms. This development should then be considered as a proof of concept allowing to test several ideas. However, the extracted program obtained this way is independent of these unfinished parts: completing them will only add a guarantee of correction to this program.

We now describe the critical points that made this small experiment far more successful than the extraction from C-CoRN.

5.1 Rational numbers

In this project, we reused the rational numbers of C-CoRN with one major improvement. In C-CoRN, the operations on $\mathbb{Q}$ never reduced fractions to canonical forms. Consequently, the computed fractions e.g. during the computation of e grew very quickly without need. Although frequently simplifying the fractions can also be dangerous, because this also has a cost, our experimental development showed without possible ambiguity the enormous computation speed-up induced by some carefully introduced simplifications: instead of three digits of $\sqrt{2}$, we were able to compute several hundreds quickly.

The simplification of fractions is done by a function `Qred`: $\mathbb{Q} \to \mathbb{Q}$, which computes the gcd of the numerator and denominator and divides both by this number. We proved that any fraction returned by `Qred` is (numerically) equal to the initial fraction and inserted a `Qred` in the main computation loop for $\sqrt{2}$. More thorough tests would be needed to decide whether it would it be better to place one `Qred` at each elementary operation, or on the contrary to use it less frequently.

Induced by this we tried to add a similar function in the C-CoRN formalization. This induced indeed a speedup, but the gain factor was far smaller that the one we were expecting; for this reason we abandoned this idea.

5.2 Cauchy sequences

Another major difference with C-CoRN is the definition of Cauchy sequence. There, x is a Cauchy sequence if

$$\forall k.\exists N.\forall m, n > N.|f(n) - f(m)| \leq 2^{-k}.$$

In [19], the bound N is given explicitly as a function of k, this last function being named *modulus* of the Cauchy sequence. These two formulations are equivalent from the constructive point of view, since one can find the modulus $N(k)$ starting from the proof of $\forall k.\exists N \ldots$; nevertheless, the more explicit formulation of the modulus encourages one to choose it carefully, and we

endeavored to do this as accurately as possible. As a consequence, during the computation of an approximation of $\sqrt{2}$ by the extracted program, one directly obtains an upper error margin for the result. When the argument given to the program is n, then the result is within 2^{-n} of $\sqrt{2}$ or, said otherwise, we get n correct binary digits. In practice, we even get some additional correct digits due to approximations in the computation of the sequences modulus, but the order of magnitude is correct. In comparison, during the computation of e in C-CoRN, it is experimentally clear that the approximation of rank k, i.e. the k-th term of the Cauchy sequence, provides us with n correct decimal digits, but the relation between k and n is not explicit. Furthermore, if it were made explicit, it is not sure that this relation would be very accurate, due to the often naive choices for bounds in C-CoRN. For this reason, the error estimates mentioned in the text were carried out *a posteriori* using Mathematica.

5.3 Continuous functions

The definition of continuous real functions differs also appreciably between [19] and C-CoRN. In the latter, a continuous function is a function $\mathbb{R} \to \mathbb{R}$ coupled with some properties. Since $\mathbb{R}$ is a set of Cauchy sequences, this reduces to an object of type $(\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$. Having a function as first argument is not desirable, because it makes the extracted code more complex, more delicate to analyze and potentially less effective. The alternative is then to use a family of rational functions converging to the desired real function; hence, a function is now an object of type $\mathbb{N} \to \mathbb{Q} \to \mathbb{Q}$.

Abstracting from these representation questions, the underlying algorithm *is the same one* as in Section 4. In particular, we are also using here the variant of the IVT specialized for monotonic functions. The difference is that now we have defined a strictly minimal set of needed objects, and we have not introduced any abstraction layer between real numbers and results like the IVT. For instance, we have not expressed that $\mathbb{R}$ is a field, since division was useless. As a consequence, the extracted program is both:

- short: because of the search for minimality, the final code is made of only 600 lines, mostly library functions;
- simple: the main loop consists of 20 lines of fairly clear code where no complex structures are used (unlike in C-CoRN); the extracted program is directly typable and requires no use of `Obj.magic`;
- quick: with this approach, computing 40 digits takes about three minutes; of course, this is still slower than computing in Mathematica, due to the different data structures used to encode arbitrary precision integers, but this is still much faster than program extracted from C-CoRN.

This clearly illustrates the effect of representation choices on the efficiency of the extracted code, even if these representations are isomorphic from a mathematical point of view.

6 Abstraction and Computation

The discussion above suggests that we should temper one of our initial statements about the Curry–Howard isomorphism: proving theorems and building programs can be done in the same framework, but they are certainly not the same process, since they follow quite different requirements.

First, a good mathematical proof should be done in a setting as general as possible, independently of a particular data representation. But for a program, a quicker but more restrictive algorithm will be preferable to a slower but broader one, as the example of the IVT shows. Also, changes in data representation have a huge impact on the efficiency of programs, as well as whether high-level parts are allowed to access low-level representations.

A simple and illustrating example of the difference between the two processes is the choice of the representation of the integers. From a programming perspective, using binary integers is clearly the best choice, since they yield more efficient functions. From a mathematical perspective, though, the induction principle derived from the Peano (unary) integers is the usual one, while the induction principle derived from the binary integers is very awkward to use in practice.

Another instance of this difference, omnipresent in C-CoRN, is the use of *coercions*. In mathematics, it is common practice to identify sub-structures with their image by inclusion, e.g. the rational numbers with their injection into the reals, and to consider algebraic structures as weaker ones, e.g. to see rings as groups. Expliciting all these silent isomorphisms and injections would transform mathematics into a permanent nightmare; therefore, Coq provides a coercion mechanism that gives a precise meaning to these practices. These coercions are a syntactic facility, allowing the user to write less, but internally they are just ordinary functions that are not displayed to the user. However, they appear again after the extraction.

In the concrete case of C-CoRN, with a chain of ten imbricated algebraic structures ranging from setoids to complex numbers, the extracted code contains literally thousands of explicit coercions: the sole real expression $0 + 1$ needs three lines to be written. Although these coercions have a negligible effect on the execution time, they prevent any detailed analysis of the code by making it unreadable.

A last point concerns the optimizations one may want to integrate into the extracted code. In our framework, these optimizations have to be made on the proof level. Experimentally we have noticed that this is neither natural nor easy. For example, a experienced programmer would avoid writing twice the same function call with the same arguments, and would group these calls with a `let in` construct; but in a proof it is commonplace to use the same hypothesis repeatedly, and the extracted code then ends up repeating the corresponding computations. In the context of recursive functions, that changes the complexity of the program dramatically — this is certainly one major

explanation for the current inefficiency of the FTA program. Unfortunately, the size of this program currently prevents anyone from trying to factorize all redundancies automatically. Moreover, when one knows what to group with a `let in`, it is generally feasible to modify the proof accordingly; but for more subtle optimizations the proof may need to be completely redone. Manipulating, maintaining and improving a program-carrying proof is certainly not as easy as manipulating the program directly.

7 Conclusions

Although program extraction is not a new subject, practical work in this field is still scarce. There have been a few experiments, but they typically concern small formalizations, which can be explained by the investment in time that would be needed to build a library big enough for larger test cases.

For this reason, the work we presented is a *première*. The formalized proof of the FTA in C-CoRN consists of over 1400 lemmas and tens of thousands of lines of code; though we could not examine the program extracted from that specific proof, the smaller examples we discussed are still hundreds of times larger than the examples in e.g. [16]. Our work also differs from other experiments because it concerns a proof of a mathematical statement that embodies an algorithm used in mathematics, whereas previous work in program extraction has dealt mostly with examples from Computer Science.

The results we discussed are promising in some respects and very disappointing in others. On the positive side, the importance of obtaining mathematically correct programs cannot be overstated, specially when these programs embody non-trivial algorithms such as the root-finding one in the FTA proof. Also, the performance of the program computing e is very impressive, although of course not comparable with that of computer algebra systems. On the negative side, though, extracting an efficient program from the full proof of the FTA is beyond what can be done presently: simply computing $\sqrt{2}$ with two decimals of precision takes up all available resources.

One of the issues that strongly influences the performance of the extracted program is the real number structure being used. In C-CoRN, the only one that has been implemented to date defines the reals as Cauchy sequences of rational numbers, the latter formalized as pairs of one integer and one positive integer. This approach is very abstract, with the consequence that some simple functions on the reals are not optimal. We feel that this is one of the limitations of our work; among the plans for the near future is an implementation of reals using the Stern–Brocot representation described in [15], but at the time of writing this has still not been done.

The target language of the extraction is also important. The Coq extraction mechanism allows one to extract to two languages, namely Ocaml and in Haskell. In principle, the difference between the corresponding programs should be minor except when the extraction takes advantage of particular con-

structs of the language, like the records of Ocaml; in practice, the difference in the semantics and in the implementation of these languages may result in rather different behaviours of the extracted programs. A good example of this is the dramatic difference in performance between Ocaml and Haskell on $\sqrt{2}$. A more thorough understanding of these differences needs to be brought about by more tests, profiling and discussion with Ocaml/Haskell experts.

The big question is whether program extraction really comes for free. Already [7] points out that obtaining a program of a reasonable size requires changing proofs at several places; making the program run in reasonable time requires even more changes. Most of these changes made sense from the point of view of the formalization, yet it does not seem likely that we will ever be able to learn how to develop a formalization in a way that these changes will not be needed — unless extraction is a goal from the beginning. While there is a relatively well-defined notion of "efficient program", there is no definition of "good proof", let alone of "good formal proof"; and if there were, nothing guarantees that good formal proofs would extract to efficient programs; after all, natural things to do when proving a statement often yield inefficient programs. As the discussion in Section 5 pointed out, reproving the IVT with the goal of extraction produced an incomparably better program than all the optimizations to the existing formalization.

The last point worth mention is the extracted program itself. It is striking that it does not look at all like a program anyone would write.

References

[1] K. Aehlig, U. Berger, M. Hofmann, and H. Schwichtenberg. An arithmetic for non-size-increasing polynomial-time computation. *Theoretical Computer Science*, 318:3–27, 2004.

[2] R. Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, January 2001.

[3] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[4] 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.

[5] L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. PhD thesis, University of Nijmegen, April 2004.

[6] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN: the Constructive Coq Repository at Nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management, Third International Conference, MKM 2004*, volume 3119 of *LNCS*, pages 88–103. Springer–Verlag, 2004.

[7] 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, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag, 2003.

[8] H. Benl et al. Proof theory at work: Program development in the Minlog system. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*. Kluwer Academic Publishers, Dordrecht, 1998.

[9] H. Geuvers and M. Niqui. Constructive reals in Coq: Axioms and categoricity. In Callaghan et al. [4], pages 79–95.

[10] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the Fundamental Theorem of Algebra without using the rationals. In Callaghan et al. [4], pages 96–111.

[11] S. Hayashi and H. Nakano. PX, a Computational Logic. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1987.

[12] C. Kreitz. *The Nuprl Proof Development System, Version 5*. Cornell University, Ithaca, NY, 2002. Available at `http://www.nuprl.org`.

[13] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 200–219. Springer–Verlag, 2003.

[14] P. Letouzey. *Programmation functionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[15] M. Niqui and Y. Bertot. QArith: Coq formalisation of lazy rational arithmetic. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 309–323. Springer–Verlag, 2004.

[16] Aleksey Nogin. Writing constructive proofs yielding efficient extracted programs. In Didier Galmiche, editor, *Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics*, volume 37 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.

[17] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989.

[18] C. Paulin-Mohring and B. Werner. Synthesis of ml programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[19] H. Schwichtenberg. Constructive analysis with witnesses. Technical report, Ludwig-Maximilians-Universität, München, 2003. Proceedings of Marktoberdorf '03 Summer School.

[20] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*, February 2004. Available at `http://coq.inria.fr`.

[21] S. Wolfram. *The Mathematica Book*. Wolfram Media, 2003. Fifth Edition.