Computable Refinements by Quotients and Parametricity*

Cyril Cohen[1], Maxime Dénès[2] and Anders Mörtberg[1]

[1] Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
{cyrilc,mortberg}@chalmers.se
[2] INRIA Sophia Antipolis – Méditerranée
Maxime.Denes@inria.fr

It is commonly conceived that computationally well-behaved programs and data structures are usually more difficult to study formally than naive ones. Rich formalisms like the Calculus of Inductive Constructions, on which the Coq [2] proof assistant relies, allow several different representations of the same object so that users can choose the one suiting their needs.

Even simple objects like natural numbers may have both a unary representation which features a very straightforward induction scheme and a binary one which is exponentially more compact, but usually entails more involved proofs. Their respective incarnations in the standard library of Coq are the two inductive types `nat` and `N` along with two isomorphisms `N.of_nat : nat -> N` and `N.to_nat : N -> nat`. Operators and proofs used to be duplicated over these two types, whereas recent versions of the library make use of functors and type classes to factor the code.

However, this traditional approach to abstraction, by defining an interface specifying the signature of operators and axiomatizing their properties, proving correctness from the axioms, and then instantiating to particular implementations, has at least two drawbacks in our context. First, it is not always obvious how to define the correct interface, and not even clear that a suitable one always exists. Second, having abstract axioms goes against the type-theoretic view of objects with computational content which means in practice that proof techniques like small scale reflection, as advocated by the SSReflect extension [4], are not applicable. Instead, the approach we propose involves proving the correctness of operators on data structures designed for proofs – as opposed to an abstract signature – and then transporting them to more efficient implementations. We distinguish two notions: *program refinements* and *data refinements*.

The first of these consists in transforming a program into a more efficient one computing the same thing, preserving the involved types. For example, standard matrix multiplication can be refined to a more efficient implementation like Strassen's fast matrix product. The correctness of this kind of refinements is often straightforward to state in Coq as, in many cases, it suffices to prove that the two algorithms are extensionally equal.

However, we consider program refinement as an input to our framework and we do not explain here how to prove them formally. Instead, we focus on data refinements, by which we mean linking two different datatypes representing the same mathematical objects, and lifting the associated operators from one to the other.

We rely on a previous work [3] which defines a framework refining *algebraic structures* in a similar way, while allowing a step-by-step approach to prove correctness of algorithms. The present work[1] improves generality by extending to previously unsupported data types (like

*The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

[1] Documentation and development is available at `http://www.maximedenes.fr/coqeal/`

the rational numbers), flexibility and modularity by refining each *operator* in isolation, and automation.

In practice our framework decomposes each library into three parts. The first part consists in defining the computation-oriented datastructures together with the (possibly efficient) algorithms operating on it. Both may be based on abstract types and operators for the underlying structures. For example, given an abstract type `A` equipped with a binary operator `mul : A -> A -> A` representing multiplication, rational numbers may be implemented by the type `A * A` of pairs of elements in `A`. It is then possible to define multiplication for this representation. In order to make programming easier we use operational typeclasses [7] to parametrize by operations such as `mul`.

In the second part, the correctness of the computation-oriented type and its operations is proved by linking them to their proof-oriented counterparts. For rational numbers this could for example mean that the representation as a pair would be linked to the type `rat` of rational numbers of SSReflect, which are pairs of coprime integers where the second component is nonzero. In this case `A * A` and `rat` would not be isomorphic, but the latter would be a partial quotient of the former [1].

The correctness of the implemented algorithms is proved by instantiating by proof-oriented types and is proved with respect to its proof-oriented counterpart. For rational numbers, this would lead us to change the abstract type `A` to a proof-oriented representation of integers and prove that the definition of multiplication computes the same thing as the one for `rat` (modulo normalization). These correctness proofs are stored in a database (using typeclass instances [6]) so that they can be instrumented for automated translation.

The third step is to substitute the proof-oriented parameters with computation-oriented ones. Applying it on rational numbers implies changing the proof-oriented representation of integers into a computation-oriented one, which refines the former. This step can be done by taking advantage of parametricity. However since parametricity is not internal in Coq [5], we rely on meta-programming.

This work is intended to be used as a new basis for CoqEAL – The Coq Effective Algebra Library. This is a library that we are currently developing, containing many formally verified program refinements such as Strassen's fast matrix product, Karatsuba's fast polynomial product and the Sasaki-Murao algorithm to compute the characteristic polynomial of a matrix.

References

[1] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory.* Phd thesis, Ecole Polytechnique X, November 2012.

[2] Coq development team. The Coq Proof Assistant Reference Manual, version 8.4, 2012.

[3] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A Refinement Based Approach to Computational Algebra in Coq. In *ITP*, volume 7406 of *LNCS*, pages 83–98, 2012.

[4] Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[5] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In *CSL - 26th International Workshop/21st Annual Conference of the EACSL - 2012*, volume 16, pages 381–395, 2012.

[6] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 278–293, 2008.

[7] Bas Spitters and Eelis van der Weegen. Type Classes for Mathematics in Type Theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.