Towards primitive data types for Coq: 63-bits integers and persistent arrays*

Maxime Dénès

INRIA Sophia Antipolis - Méditerranée

Maxime.Denes@inria.fr

April 6, 2013

As formal methods are applied to an increasingly wide variety of areas of mathematics and program verification, the need for efficient computations inside proof assistants is becoming more present. Typical applications are proofs inherently relying on costly computations, like the four color theorem [Gon07], the Kepler conjecture [Hal05] or the certification of big prime numbers [GTW06]. But computational capabilities can also be used to enhance proof automation, like tactics deciding algebraic identities over rings [GM05] or Kleene algebras [BP10] or calling external solvers without trusting them [Arm+11; BCP11]. Other original applications may include importing proof objects from different proof systems [KW10] or emitting formally verified assembly code [JBK13].

Addressing this need, the Coq proof assistant has evolved to offer new features for efficient computations. Runtime environments for terms evaluation have been improved, a key step being definitely the introduction of a bytecode compiler along with a dedicated virtual machine [GL02]. This has been recently refined to evaluation by compilation to native code [BDG11]. However, another critical source of performance (or lack thereof) is the choice of data structures to represent the objects involved in the computation.

The case of numbers is symptomatic: the traditional unary representation for natural numbers quickly becomes intractable, even for simply parsing and storing them in memory. That is why Coq's standard library provides an alternative, binary representation, achieving logarithmic space and time complexity for basic operations like addition, which is already better but still does not scale to real-world computations.

The envelope was pushed further by the introduction of machine arithmetic inside the evaluation mechanism of Coq [Arm+10]. This approach consists in defining a data type and operators reflecting OCaml's standard 31 bits arithmetic. This data type behaves like a regular inductive type, except for evaluation and conversion tests, in which case it is substituted with actual OCaml integers. The *leitmotiv* of this approach is that the extension is local to the evaluation machinery, hence no change in the formalism and an easier implementation. However, there are two main drawbacks: first, when generalizing this methodology to other data types, it is not clear whether there always exists a suitable inductive type reflecting the same computational behavior. And second, objects (in our case, integers) are stored in compact form only during evaluation. In particular, a huge amount of memory is still required to allocate them for type checking, and to store them in proof terms.

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

We propose an alternative approach, initiated by Benjamin Grégoire after having hit memory limits when defining terms representing proof traces from SMT solvers, which consists in extending Coq's terms with primitive data types like integers and persistent arrays, along with operators, and axiomatizing their equational theory[1]. As usual, this is open to discussion since it lies at the heart of the tension between De Bruijn's criterion and the practical efficiency of proof systems.

As a side benefit of our ongoing work, we extend the current 31-bits arithmetic to 63-bits, which should lead to a significant performance improvement on 64-bits architectures for libraries relying on machine integers, like big numbers [GT06], floating-point arithmetic [Mel12] or fast exact reals [KS11]. We also design a minimalist axiomatization, for both integers and persistent arrays and give a suitable user view by deducing elementary properties.

References

[Arm+10] Michaël Armand et al. "Extending Coq with Imperative Features and its Application to SAT Verification". In: *ITP 2010, Edinburgh, Scotland*. 2010.

[Arm+11] Michaël Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *CPP*. 2011.

[BCP11] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. "Modular SMT Proofs for Fast Reflexive Checking Inside Coq". In: *CPP*. 2011.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full Reduction at Full Throttle". In: *CPP*. 2011.

[BP10] T. Braibant and D. Pous. "An efficient coq tactic for deciding Kleene algebras". In: *Interactive Theorem Proving* (2010).

[GL02] B. Grégoire and X. Leroy. "A compiled implementation of strong reduction". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (2002).

[GM05] B. Grégoire and A. Mahboubi. "Proving equalities in a commutative ring done right in coq". In: *Lecture notes in computer science* 3603 (2005).

[Gon07] G. Gonthier. "The Four Colour Theorem: Engineering of a Formal Proof". In: *ASCM*. 2007.

[GT06] Benjamin Grégoire and Laurent Théry. "A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers". In: *IJCAR*. 2006.

[GTW06] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. "A Computational Approach to Pocklington Certificates in Type Theory". In: *FLOPS*. 2006.

[Hal05] Thomas C. Hales. "Introduction to the Flyspeck Project". In: *Mathematics, Algorithms, Proofs*. 2005.

[JBK13] Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. "High-level separation logic for low-level code". In: *POPL*. 2013.

[KS11] Robbert Krebbers and Bas Spitters. "Type classes for efficient exact real arithmetic in Coq". In: *Logical Methods in Computer Science* 9.1 (2011).

[KW10] Chantal Keller and Benjamin Werner. "Importing HOL Light into Coq". In: *ITP*. 2010.

[Mel12] Guillaume Melquiond. "Floating-point arithmetic in the Coq system". In: *Inf. Comput.* 216 (2012).

[1] A prototype implementation can be found at `http://github.com/maximedenes/native-coq`