Efficient Finite-Domain Function Library for the Coq Proof Assistant*

Kazuhiko Sakaguchi† (Supervisor: Yukiyoshi Kameyama)

Department of Computer Science, University of Tsukuba, Japan

Summary Finiteness is an important concept in the computer science. In particular, finite-domain functions are a useful concept for representing various data structures such as finite graphs, finite automata and matrices, and used in quite a few programs.

We provide finite-domain function libraries in Coq [12], which improves the efficiency of code extracted from proofs without forcing one to rewrite the whole proofs which use existing libraries. The SSReflect/Mathematical Components [5, 8] of Coq provide the libraries to support finite types (fintype) and finite-domain functions (finfun), which allow one to drastically reduce the burden of writing proofs. While useful in proving, they have a serious problem in the performance of code extracted from proofs. In this study, we improve the fintype and finfun libraries, and show that OCaml code extracted from proofs using our libraries are much more efficient than those using the SSReflect libraries, and that existing proofs using the SSReflect libraries can be ported to the proofs using our libraries with very little modification. As concrete evidence for that, we provide a matrix-multiplication benchmark, whose time complexity has been improved from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$ by our libraries. We also demonstrate that the 170,000-line proof of Feit-Thompson theorem [6] has been successfully ported where we only have to rewrite less than 10 lines of the whole proof.

Our improved library, some case studies and benchmark scripts are available at [10].

Keywords interactive theorem proving, Coq, SSReflect, finite type, finite-domain function, program extraction, proof modularity

1 Our Approach

In the SSReflect library, finite types are characterized by nonduplicate enumeration of the elements, and a finite-domain function $f : T \to A$ is defined as a $|T|$-tuple of A. We call this tuple the *graph* of f. Application of the finite-domain function f to x is computed by (1) searching for the value x in the list $\mathrm{enum}(T)$, and let i be its index, and (2) taking the i-th element of the graph of f.

This procedure is very inefficient for computing because the first step of the procedure traverses $\mathrm{enum}(T)$. We solve this problem by redefining the finite types. In our library, finite types are characterized by encoding/decoding bijection to a *finite ordinal* type $'\mathrm{I}_n = \{0, \ldots, n-1\}$. Then, function application $f(x)$ can be computed by taking $\mathrm{enc}_T(x)$-th element of $\mathrm{graph}(f)$ directly. This new procedure is efficient if the encoding function of T is efficient.[1]

Proof Modularity and Compatibility As many existing proofs rely on the rich theories provided SSReflect, it is critically important that the users of our library need not rewrite their proofs using SSReflect, or at least the efforts of rewriting should be reasonably small. Hence, the key issues of this study are compatibility with SSReflect theories and what we call *proof modularity*.

Our library has achieved a high level compatibility with the corresponding libraries in SSReflect; we did it by: (1) giving modified definitions, but reproducing the same set of lemmas and theorems in the original library, and (2) hiding the modified definitions by the lock/unlock mechanism [5, §7.3] of the SSReflect.

Program Extraction Coq's *program extraction* mechanism [7, 9] is useful to obtain programs from formal proofs. This mechanism eliminates proof information which is useless for computation from terms, and transforms it to functional programs such as OCaml, Haskell, Scheme.

The n-tuple is defined as a dependent sum of list x and size refinement (proof of $|x| = n$) in SSReflect, and this definition is inefficient for computing. We replace n-tuples with OCaml's arrays, and redefine the application and construction of finite-domain functions with `Array.get` and `Array.init`. The first one is achieved by `Extract Inductive` command, and the second one is achieved by `Extract Constant` command.

2 Case Studies and Benchmarks

Matrix Multiplication The first case study is matrix multiplication. SSReflect provides a useful linear algebra library [4], and $m \times n$ matrices of A are defined as finite-domain functions $'\mathrm{I}_m \times {}'\mathrm{I}_n \to A$ in this library. Our improvement involves this part, and accelerates some operations on matrix such as addition and multiplication.

We compare the performance of extracted code by the $n \times n$ integer matrix multiplication benchmark. Figure 1

*This article is a extended abstract for IPSJ-PRO-2016-1-7 [11].

†`sakaguchi@coins.tsukuba.ac.jp`
`http://logic.cs.tsukuba.ac.jp/~sakaguchi/`
`https://github.com/pi8027`

[1] On the other hand, decoding functions are used for constructing graphs of finite-domain functions.

Table 1: Benchmark results of the decision procedures for Presburger arithmetic

#	decision procedure	formula	number of states	result	execution time (sec.) before	execution time (sec.) after
1	SAT	$2 \le x+y \wedge x \le 2 \wedge y \le 2 \wedge x+y = 1+4z$	35 (640)	UNSAT	0.016	0.008
2	SAT	$2 \le 2x+y \wedge 5x \le 4y \wedge 5y \le 4x+5$	70 (660)	UNSAT	0.008	0.004
3	SAT	$2 \le x+y \wedge 3x \le 6+y \wedge 3y \le 6+x \wedge x+y = 1+4z$	156 (4000)	UNSAT	0.068	0.028
4	SAT	$y \le 2x \wedge 12 \le 3x+4y \wedge 5x+y \le 15 \wedge y = 3z$	277 (10560)	SAT	0.116	0.056
5	SAT	$5 \mid x$	6 (2^8)	SAT	0.024	0.008
6	SAT	$10 \mid x$	8 (2^{13})	SAT	0.124	0.076
7	VALID	$2 \mid x \wedge 3 \mid x \Leftrightarrow 6 \mid x$	8 (2^{40})	VALID	N/A	0.036
8	VALID	$3a = b+c+d \wedge 3b = a+c+d \wedge 3c = a+b+d \wedge 3d = a+b+c \Leftrightarrow a = b \wedge a = c \wedge a = d$	262 (2^{36})	VALID	N/A	0.168

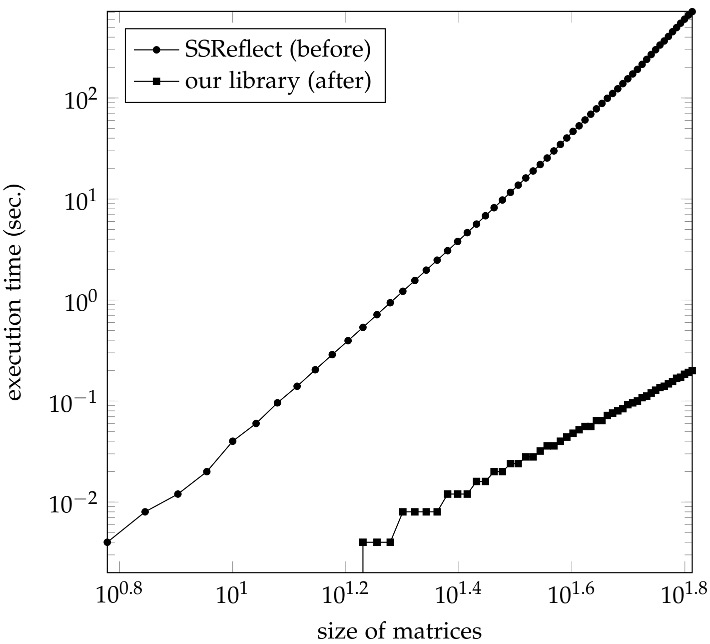

Figure 1: Benchmark results of matrix multiplication

shows the benchmark results, and it indicates the approximate time complexity has been improved from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$.

Decision Procedure for Presburger Arithmetic

Presburger arithmetic is the first-order theory on the natural numbers (or integers) with addition, but without multiplication. We formalize a proof of decidability of Presburger arithmetic by using Boudet's method [1, 3], and extract decision procedures from it. This method converts Presburger formulae to finite automata, and reduces some properties of formulae such as satisfiability and validity to a certain properties on formal languages. We use the Doczkal's regular language library [2] to formalize automata construction algorithms.

Table 1 shows the comparison of the extracted code. The cases #1 through #4, and #8 are quantifier-free formulae, and cases #5 through #7 contain quantifiers implicitly.[2] Our improved library better performed for the cases #1 through #6. In cases #7 and #8, code extracted with SSReflect fails by stack overflow. However, code extracted with our library runs successfully in all cases.

[2] Divisibility with a constant divisor $n|x$ is an abbreviation of the $\exists y.\, ny = x$.

3 Future Work

In this study, we provide an efficient finite-domain function library for Coq. It can be regarded as a library for immutable arrays with an arbitrary finite index-set. Now, we would like to extend our work to *mutable* arrays by using a restricted state monad. We expect that some graph algorithms such as Dijkstra, Warshall–Floyd and Tarjan's algorithm are good application of it.

References

[1] Alexandre Boudet and Hubert Comon. "Diophantine equations, Presburger arithmetic and finite automata". In: *Trees in Algebra and Programming — CAAP '96*. Vol. 1059. LNCS. Springer, 1996, pp. 30–43.

[2] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. "A Constructive Theory of Regular Languages in Coq". In: *Certified Programs and Proofs*. Vol. 8307. LNCS. Springer, 2013, pp. 82–97.

[3] Javier Esparza. *Automata Theory: An Algorithmic Approach*. 2016. URL: https://www7.in.tum.de/~esparza/automatanotes.html.

[4] Georges Gonthier. "Point-Free, Set-Free Concrete Linear Algebra". In: *Interactive Theorem Proving*. Vol. 6898. LNCS. Springer, 2011, pp. 103–118.

[5] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report. INRIA, 2015. URL: https://hal.inria.fr/inria-00258384v16.

[6] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving*. Vol. 7998. LNCS. Springer, 2013, pp. 163–179.

[7] Pierre Letouzey. "A New Extraction for Coq". In: *Types for Proofs and Programs*. Vol. 2646. LNCS. Springer, 2003, pp. 200–219.

[8] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. 2016. URL: https://math-comp.github.io/mcb/book.pdf.

[9] Christine Paulin-Mohring. "Extracting F_ω's programs from proofs in the Calculus of Constructions". In: *Sixteenth Annual ACM Symposium on Principles of Programming Languages*. Austin: ACM, Jan. 1989, pp. 89–104.

[10] Kazuhiko Sakaguchi. *The Coq development accompanying this paper*. URL: http://logic.cs.tsukuba.ac.jp/~sakaguchi/src/efficient-finfun-ipsj-pro-2016-1.tar.gz.

[11] Kazuhiko Sakaguchi and Yukiyoshi Kameyama. "Efficient Finite-Domain Function Library for the Coq Proof Assistant". Japanese. In: *IPSJ Journal of Programming* 10.1 (2016), pp. 14–28. URL: http://logic.cs.tsukuba.ac.jp/~sakaguchi/papers/ipsj-pro-2016-1-7.pdf.

[12] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.5beta3. 2015. URL: https://coq.inria.fr/distrib/8.5beta3/refman/.