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 \(O(n^5)\) to \(O(n^3)\) 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 \(f : T \rightarrow A\) 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 \(\text{enum}(T)\), 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 \(\text{enum}(T)\). 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 \(\text{int}_{n} = \{0, \ldots, n-1\}\). Then, function application \(f(x)\) can be computed by taking \(\text{enc}_{T}(x)\)-th element of graph \(f\) directly. This new procedure is efficient if the encoding function of \(T\) is efficient.

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 \text{Array.get} and \text{Array.init}. The first one is achieved by \text{Extract Inductive} command, and the second one is achieved by \text{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 \(\text{I}_{m} \times \text{I}_{n} \rightarrow A\) 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] on the other hand, decoding functions are used for constructing graphs of finite-domain functions.
and cases #5 through #7 contain quantifiers implicitly. cases #1 through #4, and #8 are quantifier-free formulae, algorithms. lar language library [2] to formalize automata construction properties on formal languages. We use the Doczkal’s regu-
cication. We formalize a proof of decidability of Presburger numbers (or integers) with addition, but without multipli-
imate time complexity has been improved from $O(n^5)$ to $O(n^3)$.

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. 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.

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

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

<table>
<thead>
<tr>
<th>#</th>
<th>decision procedure</th>
<th>formula</th>
<th>number of states</th>
<th>result</th>
<th>execution time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SAT</td>
<td>$2 \leq x + y \land x \leq 2 \land y \leq 2 \land x + y = 1 + 4z$</td>
<td>35 (640)</td>
<td>UNSAT</td>
<td>0.016</td>
</tr>
<tr>
<td>2</td>
<td>SAT</td>
<td>$2 \leq 2x + y \land 5x \leq 4y \land 5y \leq 4x + 5$</td>
<td>70 (660)</td>
<td>UNSAT</td>
<td>0.008</td>
</tr>
<tr>
<td>3</td>
<td>SAT</td>
<td>$2 \leq x + y \land 3x \leq 6 + y \land 3y \leq 6 + x \land x + y = 1 + 4z$</td>
<td>156 (4000)</td>
<td>UNSAT</td>
<td>0.068</td>
</tr>
<tr>
<td>4</td>
<td>SAT</td>
<td>$y \leq 2x \land 12 \leq 3x + 4y \land 5x + y \leq 15 \land y = 3z$</td>
<td>277 (10560)</td>
<td>SAT</td>
<td>0.116</td>
</tr>
<tr>
<td>5</td>
<td>SAT</td>
<td>$5 \mid x$</td>
<td>6 (25)</td>
<td>SAT</td>
<td>0.024</td>
</tr>
<tr>
<td>6</td>
<td>SAT</td>
<td>$10 \mid x$</td>
<td>8 (213)</td>
<td>SAT</td>
<td>0.124</td>
</tr>
<tr>
<td>7</td>
<td>VALID</td>
<td>$2 \mid x \land 3 \mid x \iff 6 \mid x$</td>
<td>8 (206)</td>
<td>VALID</td>
<td>N/A</td>
</tr>
<tr>
<td>8</td>
<td>VALID</td>
<td>$3a = b \mid c + d \land 3b = a + c + d \land 3c = a + b + d \land 3d = a + b + c \iff a = b \land a = c \land a = d$</td>
<td>262 (256)</td>
<td>VALID</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 1: Benchmark results of matrix multiplication
shows the benchmark results, and it indicates the approxi-
mate time complexity has been improved from $O(n^5)$ to $O(n^3)$.

Figure 1: Benchmark results of matrix multiplication shows the benchmark results, and it indicates the approxi-
mate time complexity has been improved from $O(n^5)$ to $O(n^3)$.