Accelerating Verified-Compiler Development with a Verified Rewriting Engine

Jason Gross
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA
Machine Intelligence Research Institute, Berkeley, CA, USA

Andres Erbsen
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Jade Philipoom
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA
Google, London, UK

Miraya Poddar-Agrawal
Reed College, Portland, OR, USA

Adam Chlipala
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

Compilers are a prime target for formal verification, since compiler bugs invalidate higher-level correctness guarantees, but compiler changes may become more labor-intensive to implement, if they must come with proof patches. One appealing approach is to present compilers as sets of algebraic rewrite rules, which a generic engine can apply efficiently. Now each rewrite rule can be proved separately, with no need to revisit past proofs for other parts of the compiler. We present the first realization of this idea, in the form of a framework for the Coq proof assistant. Our new Coq command takes normal proved theorems and combines them automatically into fast compilers with proofs. We applied our framework to improve the Fiat Cryptography toolchain for generating cryptographic arithmetic, producing an extracted command-line compiler that is about 1000× faster while actually featuring simpler compiler-specific proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting; Software and its engineering → Compilers; Software and its engineering → Translator writing systems and compiler generators

Keywords and phrases compiler verification, rewriting engines, cryptography

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.17

Related Version *Full Version*: `https://arxiv.org/abs/2205.00862`

Supplementary Material *Software (Source Code)*: `https://github.com/mit-plv/rewriter/tree/ITP-2022-perf-data`; archived at `swh:1:rev:1787ab401a7e71afc9937010e2e155e4b1594ab5`
Software (Source Code): `https://github.com/mit-plv/fiat-crypto/tree/perf-testing-data-ITP-2022-rewriting`; archived at `swh:1:rev:72fe0dddee5e6dceeab0b8a2e6a745abf5287d3e`

Funding This work was supported in part by a Google Research Award, National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584, and the National Science Foundation Graduate Research Fellowship under Grant Nos. 1122374 and 1745302. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

Formally verified compilers like CompCert [13] and CakeML [12] are success stories for proof assistants, helping close a trust gap for one of the most important categories of software infrastructure. A popular compiler cannot afford to stay still; developers will add new backends, new language features, and better optimizations. Proofs must be adjusted as these improvements arrive. It makes sense that the author of a new piece of compiler code must prove its correctness, but ideally there would be no need to revisit old proofs. There has been limited work, though, on avoiding that kind of coupling. Tatlock and Lerner [17] demonstrated a streamlined way to extend CompCert with new verified optimizations driven by dataflow analysis, but we are not aware of past work that supports easy extension for compilers from functional languages to C code. We present our work targeting that style.

One strategy for writing compilers modularly is to exercise foresight in designing a core that will change very rarely, such that feature iteration happens outside the core. Specifically, phrasing the compiler in terms of rewrite rules allows clean abstractions and conceptual boundaries [11]. Then, most desired iteration on the compiler can be achieved through iteration on the rewrite rules.

It is surprisingly difficult to realize this modular approach with good performance. Verified compilers can either be proof-producing (certifying) or proven-correct (certified). Proof-producing compilers usually operate on the functional languages of the proof assistants that they are written in, and variable assignments are encoded as let binders. All existing proof-producing rewriting strategies scale at least quadratically in the number of binders. This performance scaling is inadequate for applications like Fiat Cryptography [8] where the generated code has 1000s of variables in a single function. Proven-correct compilers do not suffer from this asymptotic blowup in the number of binders.

In this paper, we present **the first proven-correct compiler-builder toolkit parameterized on rewrite rules**. Arbitrary sets of Coq theorems (quantified equalities) can be assembled by a single new Coq command into an extraction-ready verified compiler. We did not need to extend the trusted code base, so our compiler compiler need not be trusted. We achieve both good performance of compiler runs and good performance of generated code, via addressing a number of scale-up challenges vs. past work.

We evaluate our toolkit by replacing a key component of Fiat Cryptography [8], a Coq library that generates code for big-integer modular arithmetic at the heart of elliptic-curve-cryptography algorithms. Routines generated (with proof) with Fiat Cryptography now ship with all major Web browsers and all major mobile operating systems. With our improved compiler architecture, it became easy to add two new backends and a variety of new supported source-code features, and we were easily able to try out new optimizations.

Replacing Fiat Cryptography's original compiler with the compiler generated by our toolkit has two additional benefits. Fiat Cryptography was previously only used successfully to build C code for two of the three most widely used curves (P-256 and Curve25519). Our prior version's execution timed out trying to compile code for the third most widely used curve (P-384). Using our new toolkit has made it possible to generate compiler-synthesized code for P-384 while generating completely identical code for the primes handled by the previous version, about 1000× more quickly. Additionally, Fiat Cryptography previously required source code to be written in continuation-passing style, and our compiler has enabled a direct-style approach, which pays off in simplifying theorem statements and proofs.

1.1 Related Work

Assume our mission is to take libraries of purely functional combinators, apply them to compile-time parameters, and compile the results down to lean C code. Furthermore, we ask for machine-checked proofs that the C programs preserve the behavior of the higher-order functional programs we started with. What good ideas from the literature can we build on?

Hickey and Nogin [11] discuss at length how to build compilers around rewrite rules. "All program transformations, from parsing to code generation, are cleanly isolated and specified as term rewrites." While they note that the correctness of the compiler is thus reduced to the correctness of the rewrite rules, they did not prove correctness mechanically. Furthermore, it is not clear that they manage to avoid the asymptotic blow-up associated with proof-producing rewriting of deeply nested let-binders. They give no performance numbers, so it is hard to say whether or not their compiler performs at the scale necessary for Fiat Cryptography. Their rewrite-engine driver is unproven OCaml code, while we will produce custom drivers with Coq proofs.

$\mathcal{R}_{tac}$ [14] is a more general framework for verified proof tactics in Coq, including an experimental reflective version of `rewrite_strat` supporting arbitrary setoid relations, unification variables, and arbitrary semidecidable side conditions solvable by other verified tactics, using de Bruijn indexing to manage binders. We found that $\mathcal{R}_{tac}$ misses a critical feature for compiling large programs: preserving subterm sharing. As a result, our experiments with compiler construction yielded clear asymptotic slowdown vs. what we eventually accomplished. $\mathcal{R}_{tac}$ is also more heavyweight to use, for instance requiring that theorems be restated manually in a deep embedding to bring them into automation procedures. Furthermore, we are not aware of any past experiments driving verified compilers with $\mathcal{R}_{tac}$.

Aehlig et al. [1] came closest to a fitting approach, using *normalization by evaluation (NbE)* [3] to bootstrap reduction of open terms on top of full reduction, as built into a proof assistant. However, it was simultaneously true that they expanded the proof-assistant trusted code base in ways specific to their technique, and that they did not report any experiments actually using the tool for partial evaluation (just traditional full reduction), potentially hiding performance-scaling challenges or other practical issues. For instance, they also do not preserve subterm sharing explicitly, and they represent variable references as unary natural numbers (de Bruijn-style). They also require that rewrite rules be embodied in ML code, rather than stated as natural "native" lemmas of the proof assistant. We will follow their basic outline with important modifications.

Our implementation builds on fast full reduction in Coq's kernel, via a virtual machine [9] or compilation to native code [5] (neither verified). Especially the latter is similar in adopting NbE for full reduction, simplifying even under λs, on top of a more traditional implementation of OCaml that never executes preemptively under λs. Neither approach unifies support for rewriting with proved rules, and partial evaluation only applies in very limited cases.

A variety of forms of pragmatic partial evaluation have been demonstrated, with Lightweight Modular Staging [16] in Scala as one of the best-known current examples. The LMS-Verify system [2] can be used for formal verification of generated code after-the-fact. Typically LMS-Verify has been used with relatively shallow properties (though potentially applied to larger and more sophisticated code bases than we tackle), not scaling to the kinds of functional-correctness properties that concern us here.

So, overall, to our knowledge, no past compiler as a set of rewrite rules has come with a full proof of correctness as a standalone functional program. Related prior work with mechanized proofs suffered from both performance bottlenecks and usability problems, the latter in requiring that eligible rewrite rules be stated in special deep embeddings.

1.2 Our Solution

Our variant on the technique of Aehlig et al. [1] has these advantages:

- It integrates with a general-purpose, foundational proof assistant, **without growing the trusted code base**.
- For a wide variety of initial functional programs, it provides **fast** partial evaluation with reasonable memory use.
- It allows reduction that **mixes** *rules of the definitional equality* with *equalities proven explicitly as theorems*.
- It allows **rapid iteration** on rewrite rules with *minimal verification overhead*.
- It **preserves sharing** of common subterms.
- It also allows **extraction of standalone compilers**.

Our contributions include answers to a number of challenges that arise in scaling NbE-based partial evaluation in a proof assistant. First, we rework the approach of Aehlig et al. [1] to function *without extending a proof assistant's trusted code base*, which, among other challenges, requires us to prove termination of reduction and encode pattern matching explicitly (leading us to adopt the performance-tuned approach of Maranget [15]). We also improve on Coq-specific related work (e.g., of Malecha and Bengtson [14]) by allowing rewrites to be written in natural Coq form (not special embedded syntax-tree types), while supporting optimizations associated with past unverified engines (e.g., Boespflug [4]).

Second, using partial evaluation to generate residual terms thousands of lines long raises *new scaling challenges*:

- Output terms may contain so *many nested variable binders* that we expect it to be performance-prohibitive to perform bookkeeping operations on first-order-encoded terms (e.g., with de Bruijn indices, as is done in $\mathcal{R}_{tac}$ by Malecha and Bengtson [14]). For instance, while the reported performance experiments of Aehlig et al. [1] generate only closed terms with no binders, Fiat Cryptography may generate a single routine (e.g., multiplication for curve P-384) with nearly a thousand nested binders.
- Naive representation of terms without proper *sharing of common subterms* can lead to fatal term-size blow-up.
- Unconditional rewrite rules are in general insufficient, and we need *rules with side conditions*. E.g., Fiat Cryptography depends on checking lack-of-overflow conditions.
- However, it is also not reasonable to expect a general engine to discharge all side conditions on the spot. We need integration with *abstract interpretation*.

Briefly, our respective solutions to these problems are the *parametric higher-order abstract syntax (PHOAS)* [6] term encoding, a *let-lifting* transformation threaded throughout reduction, extension of rewrite rules with executable Boolean side conditions, and a design pattern that uses decorator function calls to include analysis results in a program.

Finally, we carry out the *first large-scale performance-scaling evaluation* of a verified rewrite-rule-based compiler, covering all elliptic curves from the published Fiat Cryptography experiments, along with microbenchmarks.

We pause to give a motivating example before presenting the core structure of our engine (Section 3), the additional scaling challenges we faced (Section 4), experiments (Section 5), and conclusions. Our implementation is attached.

2 A Motivating Example

Our compilation style involves source programs that mix higher-order functions and inductive types. We want to compile to C code, reducing away uses of fancier features while seizing opportunities for arithmetic simplification. Here is a small but illustrative example.

```
Definition prefixSums (ls : list nat) : list nat :=
  let ls' := combine ls (seq 0 (length ls)) in
  let ls'' := map (λ p, fst p * snd p) ls' in
  let '(_, ls''') := fold_left (λ '(acc, ls''') n,
    let acc' := acc + n in (acc', acc' :: ls''')) ls'' (0, []) in ls'''.
```

This function first computes list `ls'` that pairs each element of input list `ls` with its position, so, for instance, list $[a; b; c]$ becomes $[(a, 0); (b, 1); (c, 2)]$. Then we map over the list of pairs, multiplying the components at each position. Finally, we compute all prefix sums.

We would like to specialize this function to particular list lengths. That is, we know in advance how many list elements we will pass in, but we do not know the values of those elements. For a given length, we can construct a schematic list with one free variable per element. For example, to specialize to length four, we can apply the function to list `[a; b; c; d]`, and we expect this output:

```
let acc := b + c * 2 in let acc' := acc + d * 3 in [acc'; acc; b; 0]
```

We do not quite have C code yet, but, composing this code with another routine to consume the output list, we easily arrive at a form that looks almost like three-address code and is quite easy to translate to C and many other languages.

Notice how subterm sharing via `let`s is important. As list length grows, we avoid quadratic blowup in term size through sharing. Also notice how we simplified the first two multiplications with $a \cdot 0 = 0$ and $b \cdot 1 = b$ (each of which requires explicit proof in Coq), using other arithmetic identities to avoid introducing new variables for the first two prefix sums of `ls''`, as they are themselves constants or variables, after simplification.

To set up our compiler, we prove the algebraic laws that it should use for simplification, starting with basic arithmetic identities.

```
Lemma zero_plus : ∀ n, 0 + n = n.      Lemma times_zero : ∀ n, n * 0 = 0.
Lemma plus_zero : ∀ n, n + 0 = n.      Lemma times_one  : ∀ n, n * 1 = n.
```

Next, we prove a law for each list-related function, connecting it to the primitive-recursion combinator for some inductive type (natural numbers or lists, as appropriate). We also use a further marker `ident.eagerly` to ask the compiler to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree.

```
Lemma eval_map A B (f : A -> B) l
: map f l = ident.eagerly list_rect _ _ [] (λ x _ l', f x :: l') l.
Lemma eval_fold_left A B (f : A -> B -> A) l a
: fold_left f l a = ident.eagerly list_rect _ _ (λ a, a) (λ x _ r a, r (f a x)) l a.
Lemma eval_combine A B (la : list A) (lb : list B)
: combine la lb =
list_rect _ (λ _, []) (λ x _ r lb, list_case (λ _, _) [] (λ y ys, (x,y)::r ys) lb) la lb.
Lemma eval_length A (ls : list A)
: length ls = list_rect _ 0 (λ _ _ n, S n) ls.
```

With all the lemmas available, we can package them up into a rewriter, which triggers generation of a specialized compiler and its soundness proof. Our Coq plugin introduces a new command `Make` for building rewriters

```
Make rewriter := Rewriter For (zero_plus, plus_zero, times_zero, times_one, eval_map,
  eval_fold_left, do_again eval_length, do_again eval_combine,
  eval_rect nat, eval_rect list, eval_rect prod) (with delta) (with extra idents (seq)).
```

Most inputs to `Rewriter For` list quantified equalities to use for left-to-right rewriting. However, we also use options `do_again`, to request that some rules trigger extra bottom-up passes after being used for rewriting; `eval_rect`, to queue up eager evaluation of a call to a primitive-recursion combinator on a known recursive argument; `with delta`, to request evaluation of all monomorphic operations on concrete inputs; and `with extra idents`, to inform the engine of further permitted identifiers that do not appear directly in any of the rewrite rules.

Our plugin also provides new tactics like `Rewrite_rhs_for`, which applies a rewriter to the right-hand side of an equality goal. That last tactic is just what we need to synthesize a specialized `prefixSums` for list length four, along with its correctness proof.

```
Definition prefixSums4 :
{f:nat→nat→nat→nat→list nat | ∀ a b c d, f a b c d = prefixSums [a;b;c;d]}
  := ltac:(eexists; Rewrite_rhs_for rewriter; reflexivity).
```

That compiler execution ran inside of Coq, but an even more pragmatic approach is to *extract* the compiler as a standalone program in OCaml or Haskell. Such a translation is possible because the `Make` command produces a proved program in Gallina, Coq's logic. As a result, our reworking of Fiat Cryptography compilation culminated in extraction of a command-line OCaml program that developers in industry have been able to run without our help, where Fiat Cryptography previously required installing and running Coq, with an elaborate build process to capture its output. It is also true that the standalone program is about 10× as fast as execution within Coq, though the trusted code base is larger.

3 The Structure of a Rewriter

We are mostly guided by Aehlig et al. [1] but made a number of crucial changes. Let us review the basic idea of the approach of Aehlig et al. First, their supporting library contains:

1. Within the logic of the proof assistant (Isabelle/HOL, in their case), a type of syntax trees for ML programs is defined, with an associated (trusted) operational semantics.
2. They also wrote a reduction function in (deeply embedded) ML, parameterized on a function to choose the next rewrite, and proved it sound once-and-for-all.

Given a set of rewrite rules and a term to simplify, their main tactic must:

1. *Generate a (deeply embedded) ML program that decides which rewrite rule, if any, to apply at the top node of a syntax tree*, along with a proof of its soundness.
2. *Generate a (deeply embedded) ML term standing for the term we set out to simplify*, with a proof that it means the same as the original.
3. Combining the general proof of the rewrite engine with proofs generated by reification (the prior two steps), conclude that an application of the reduction function to the reified rules and term is indeed an ML term that generates correct answers.
4. "Throw the ML term over the wall," using a general code-generation framework for Isabelle/HOL [10]. Trusted code compiles the ML code into the concrete syntax of Standard ML, and compiles it, and runs it, asserting an axiom about the outcome.

Here is where our approach differs at that level of detail:

- Our reduction engine is written *as a normal Gallina functional program*, rather than within a deeply embedded language. As a result, we are able to prove its type-correctness and termination, and we are able to run it within Coq's kernel.
- We do *compile-time specialization of the reduction engine* to sets of rewrite rules, removing overheads of generality.

3.1 Our Approach in Ten Steps

Here is a bit more detail on the steps that go into applying our Coq plugin, many of which we expand on in the following sections. For `Make` to precompute a rewriter:

1. The given lemma statements are scraped for which named identifiers to encode.
2. Inductive types enumerating all available primitive types and functions are emitted. This allows us to achieve the performance gains attributed in Boespflug [4] to having native metalanguage constructors for all constants, without manual coding.
3. Tactics generate all of the necessary definitions and prove all of the necessary lemmas for dealing with this particular set of inductive codes. Definitions include operations like Boolean equality on type codes and lemmas like "all types have decidable equality."
4. The statements of rewrite rules are reified and soundness and syntactic-well-formedness lemmas are proven about each of them.
5. Definitions and lemmas needed to prove correctness are assembled into a single package.

Then, to rewrite in a goal, the following steps are performed:

1. Rearrange the goal into a single quantifier-free logical formula.
2. Reify a selected subterm and replace it with a call to our denotation function.
3. Rewrite with a theorem, into a form calling our rewriter.
4. Call Coq's built-in full reduction (`vm_compute`) to reduce this application.
5. Run standard call-by-value reduction to simplify away use of the denotation function.

The object language of our rewriter is nearly simply typed.

$$e ::= \texttt{App}\ e_1\ e_2 \mid \texttt{Let}\ v = e_1\ \texttt{In}\ e_2 \mid \texttt{Abs}\ (\lambda v.\, e) \mid \texttt{Var}\ v \mid \texttt{Ident}\ i$$

The `Ident` case is for identifiers, which are described by an enumeration specific to a use of our library. For example, the identifiers might be codes for $+$, $\cdot$, and literal constants. We write $[\![e]\!]$ for a standard denotational semantics.

3.2 Pattern-Matching Compilation and Evaluation

Aehlig et al. [1] feed a specific set of user-provided rewrite rules to their engine by generating code for an ML function, which takes in deeply embedded term syntax (actually *doubly* deeply embedded, within the syntax of the deeply embedded ML!) and uses ML pattern matching to decide which rule to apply at the top level. Thus, they delegate efficient implementation of pattern matching to the underlying ML implementation. As we instead build our rewriter in Coq's logic, we have no such option to defer to ML.

We could follow a naive strategy of repeatedly matching each subterm against a pattern for every rewrite rule, as in the rewriter of Malecha and Bengtson [14], but in that case we do a lot of duplicate work when rewrite rules use overlapping function symbols. Instead, we adopted the approach of Maranget [15], who describes compilation of pattern matches in OCaml to decision trees that eliminate needless repeated work (for example, decomposing an expression into $x + y + z$ only once even if two different rules match on that pattern).

There are three steps to turn a set of rewrite rules into a functional program that takes in an expression and reduces according to the rules. The first step is pattern-matching compilation: we must compile the left-hand sides of the rewrite rules to a decision tree that describes how and in what order to decompose the expression, as well as describing which rewrite rules to try at which steps of decomposition. Because the decision tree is merely a decomposition hint, we require no proofs about it to ensure soundness of our rewriter. The second step is decision-tree evaluation, during which we decompose the expression as per the decision tree, selecting which rewrite rules to attempt. The only correctness lemma needed for this stage is that any result it returns is equivalent to picking some rewrite rule and rewriting with it. The third and final step is to actually rewrite with the chosen rule. Here the correctness condition is that we must not change the semantics of the expression.

While pattern matching begins with comparing one pattern against one expression, Maranget's approach works with intermediate goals that check multiple patterns against multiple expressions. A decision tree describes how to match a vector (or list) of patterns against a vector of expressions. It is built from these constructors:

- `TryLeaf k onfailure`: Try the k^{th} rewrite rule; if it fails, keep going with `onfailure`.
- `Failure`: Abort; nothing left to try.
- `Switch icases app_case default`: With the first element of the vector, match on its kind; if it is an identifier matching something in `icases`, which is a list of pairs of identifiers and decision trees, remove the first element of the vector and run that decision tree; if it is an application and `app_case` is not `None`, try the `app_case` decision tree, replacing the first element of each vector with the two elements of the function and the argument it is applied to; otherwise, do not modify the vectors and use the `default`.
- `Swap i cont`: Swap the first element of the vector with the i^{th} element (0-indexed) and keep going with `cont`.

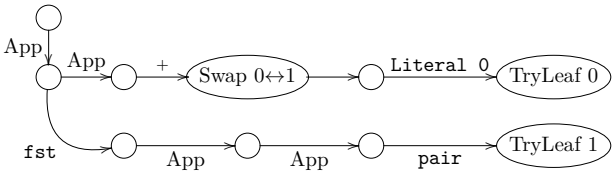
Consider the encoding of two simple example rewrite rules, where we follow Coq's $\mathcal{L}_{tac}$ language in prefacing pattern variables with question marks.

$?n + 0 \to n$ $\qquad\qquad$ $\texttt{fst}_{\mathbb{Z},\mathbb{Z}}(?x, ?y) \to x$

We embed them in an AST type for patterns, which largely follows our ASTs for expressions.

```
0. App (App (Ident +) Wildcard) (Ident (Literal 0))
1. App (Ident fst) (App (App (Ident pair) Wildcard) Wildcard)
```

The decision tree produced is



where every nonswap node implicitly has a "default" case arrow to `Failure` and circles represent `Switch` nodes.

We implement, in Coq's logic, an evaluator for these trees against terms. Note that we use Coq's normal partial evaluation to turn our general decision-tree evaluator into a specialized matcher to get reasonable efficiency. Although this partial evaluation of our partial evaluator is subject to the same performance challenges we highlighted in the introduction, it only has to be done once for each set of rewrite rules, and we are targeting cases where the time of per-goal reduction dominates this time of metacompilation.

For our running example of two rules, specializing gives us this match expression.

```
match e with
| App f y => match f with
  | Ident fst => match y with
    | App (App (Ident pair) x) y => x | _ => e end
  | App (Ident +) x => match y with
    | Ident (Literal 0) => x | _ => e end | _ => e end | _ => e end.
```

3.3 Adding Higher-Order Features

Fast rewriting at the top level of a term is the key ingredient for supporting customized algebraic simplification. However, not only do we want to rewrite throughout the structure of a term, but we also want to integrate with simplification of higher-order terms, in a way where we can prove to Coq that our syntax-simplification function always terminates. Normalization by evaluation (NbE) [3] is an elegant technique for adding the latter aspect, in a way where we avoid needing to implement our own λ-term reducer or prove it terminating.

To orient expectations: we would like to enable the following reduction

$$(\lambda f\ x\ y.\ f\ x\ y)\ (+)\ z\ 0 \rightsquigarrow z$$

using the rewrite rule

$$?n + 0 \to n$$

We begin by reviewing NbE's most classic variant, for performing full β-reduction in a simply typed term in a guaranteed-terminating way. Our simply typed λ-calculus syntax is:

$$t ::= t \to t \mid b \qquad\qquad e ::= \lambda v.\, e \mid e\ e \mid v \mid c$$

with v for variables, c for constants, and b for base types.

We can now define normalization by evaluation. First, we choose a "semantic" representation for each syntactic type, which serves as an interpreter's result type.

$$\mathrm{NbE}_t(t_1 \to t_2) = \mathrm{NbE}_t(t_1) \to \mathrm{NbE}_t(t_2) \qquad\qquad \mathrm{NbE}_t(b) = \mathtt{expr}(b)$$

Function types are handled as in a simple denotational semantics, while base types receive the perhaps-counterintuitive treatment that the result of "executing" one is a syntactic expression of the same type. We write $\mathtt{expr}(b)$ for the metalanguage type of object-language syntax trees of type b, relying on a type family `expr`.

Now the core of NbE, shown in Figure 1, is a pair of dual functions reify and reflect, for converting back and forth between syntax and semantics of the object language, defined by primitive recursion on type syntax. We split out analysis of term syntax in a separate function reduce, defined by primitive recursion on term syntax, when usually this functionality would be mixed in with reflect. The reason for this choice will become clear when we extend NbE.

We write v for object-language variables and x for metalanguage (Coq) variables, and we overload λ notation using the metavariable kind to signal whether we are building a host λ or a λ syntax tree for the embedded language. The crucial first clause for reduce replaces object-language variable v with fresh metalanguage variable x, and then we are somehow tracking that all free variables in an argument to reduce must have been replaced with metalanguage variables by the time we reach them. We reveal in Subsection 4.1 the encoding decisions that make all the above legitimate, but first let us see how to integrate

$$\begin{aligned}
\text{reify}_t &: \text{NbE}_t(t) \to \text{expr}(t) \\
\text{reify}_{t_1 \to t_2}(f) &= \lambda v.\, \text{reify}_{t_2}(f(\text{reflect}_{t_1}(v))) \\
\text{reify}_b(f) &= f \\
\\
\text{reflect}_t &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reflect}_{t_1 \to t_2}(e) &= \lambda x.\, \text{reflect}_{t_2}(e(\text{reify}_{t_1}(x)) \\
\text{reflect}_b(e) &= e
\end{aligned}$$

$$\begin{aligned}
\text{reduce} &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reduce}(\lambda v.\ e) &= \lambda x.\ \text{reduce}([x/v]e) \\
\text{reduce}(e_1\ e_2) &= (\text{reduce}(e_1))\,(\text{reduce}(e_2)) \\
\text{reduce}(x) &= x \\
\text{reduce}(c) &= \text{reflect}(c) \\
\\
\text{NbE} &: \text{expr}(t) \to \text{expr}(t) \\
\text{NbE}(e) &= \text{reify}(\text{reduce}(e))
\end{aligned}$$

Figure 1 Implementation of normalization by evaluation.

use of the rewriting operation from the previous section. To fuse NbE with rewriting, we only modify the constant case of `reduce`. First, we bind our specialized decision-tree engine (which rewrites *at the root of an AST only*) under the name rewrite-head.

In the constant case, we still reflect the constant, but underneath the binders introduced by full η-expansion, we perform one instance of rewriting. In other words, we change this one function-definition clause:

$$\text{reflect}_b(e) = \text{rewrite-head}(e)$$

It is important to note that a constant of function type will be η-expanded only once for each syntactic occurrence in the starting term, though the expanded function is effectively a thunk, waiting to perform rewriting again each time it is called. From first principles, it is not clear why such a strategy terminates on all possible input terms.

The details so far are essentially the same as in the approach of Aehlig et al. [1]. Recall that their rewriter was implemented in a deeply embedded ML, while ours is implemented in Coq's logic, which enforces termination of all functions. Aehlig et al. did not prove termination, which indeed does not hold for their rewriter in general, which works with untyped terms, not to mention the possibility of divergent rule-specific ML functions. In contrast, we need to convince Coq up-front that our interleaved λ-term normalization and algebraic simplification always terminate. Additionally, we must prove that rewriting preserves term denotations, which can easily devolve into tedious binder bookkeeping.

The next section introduces the techniques we use to avoid explicit termination proof or binder bookkeeping, in the context of a more general analysis of scaling challenges.

4 Scaling Challenges

Aehlig et al. [1] only evaluated their implementation against closed programs. What happens when we try to apply the approach to partial-evaluation problems that should generate thousands of lines of low-level code?

4.1 Variable Environments Will Be Large

We should think carefully about representation of ASTs, since many primitive operations on variables will run in the course of a single partial evaluation. For instance, Aehlig et al. [1] reported a significant performance improvement changing variable nodes from using strings to using de Bruijn indices [7]. However, de Bruijn indices and other first-order representations remain painful to work with. We often need to fix up indices in a term being

substituted in a new context. Even looking up a variable in an environment tends to incur linear time overhead, thanks to traversal of a list. Perhaps we can do better with some kind of balanced-tree data structure, but there is a fundamental performance gap versus the arrays that can be used in imperative implementations. Unfortunately, it is difficult to integrate arrays soundly in a logic. Also, even ignoring performance overheads, tedious binder bookkeeping complicates proofs.

Our strategy is to use a variable encoding that pushes all first-order bookkeeping off on Coq's kernel or the implementation of the language we extract to, which are themselves performance-tuned with some crucial pieces of imperative code. Parametric higher-order abstract syntax (PHOAS) [6] is a dependently typed encoding of syntax where binders are managed by the enclosing type system. It allows for relatively easy implementation and proof for NbE, so we adopted it for our framework.

Here is the actual inductive definition of term syntax for our object language, PHOAS-style. The characteristic oddity is that the core syntax type `expr` is parameterized on a dependent type family for representing variables. However, the final representation type `Expr` uses first-class polymorphism over choices of variable type, bootstrapping on the metalanguage's parametricity to ensure that a syntax tree is agnostic to variable type.

```
Inductive type := arrow (s d : type) | base (b : base_type).
Infix "→" := arrow.
Inductive expr (var : type -> Type) : type -> Type :=
| Var {t} (v : var t) : expr var t
| Abs {s d} (f : var s -> expr var d) : expr var (s → d)
| App {s d} (f : expr var (s → d)) (x : expr var s) : expr var d
| LetIn {a b} (x : expr var a) (f : var a -> expr var b) : expr var b
| Const {t} (c : const t) : expr var t.
Definition Expr (t : type) : Type := forall var, expr var t.
```

A good example of encoding adequacy is assigning a simple denotational semantics. First, a simple recursive function assigns meanings to types.

```
Fixpoint denoteT (t : type) : Type := match t with
  | arrow s d => denoteT s -> denoteT d
  | base b    => denote_base_type b  end.
```

Next we see the convenience of being able to *use* an expression by choosing how it should represent variables. Specifically, it is natural to choose *the type-denotation function itself* as variable representation. Especially note how this choice makes rigorous last section's convention (e.g., in the suspicious function-abstraction clause of reduce), where a recursive function enforces that values have always been substituted for variables early enough.

```
Fixpoint denoteE {t} (e : expr denoteT t) : denoteT t := match e with
  | Var v     => v
  | Abs f     => λ x, denoteE (f x)
  | App f x   => (denoteE f) (denoteE x)
  | LetIn x f => let xv := denoteE x in denoteE f xv
  | Ident c   => denoteI c  end.
Definition DenoteE {t} (E : Expr t) : denoteT t := denoteE (E denoteT).
```

It is now easy to follow the same script in making our rewriting-enabled NbE fully formal, in Figure 2. Note especially the first clause of `reduce`, where we avoid variable substitution precisely because we have chosen to represent variables with normalized semantic values. The subtlety there is that base-type semantic values are themselves expression syntax trees, which depend on a nested choice of variable representation, which we retain as a parameter throughout these recursive functions. The final definition λ-quantifies over that choice.

```
Fixpoint nbeT var (t : type) : Type :=
match t with
| arrow s d => nbeT var s -> nbeT var d
| base b    => expr var b
end.


Fixpoint reify {var t}
  : nbeT var t -> expr var t :=
match t with
| arrow s d => λ f, Abs (λ x,
    reify (f (reflect (Var x))))
| base b    => λ e, e            end
```

```
with reflect{var t}:expr var t->nbeT var t
  := match t with
| arrow s d => λ e, λ x,
  reflect (App e (reify x))
| base b    => rewrite_head      end.
Fixpoint reduce{var t}(e:expr (nbeT var) t)
  : nbeT var t := match e with
| Abs e     => λ x, reduce (e (Var x))
| App e1 e2 => (reduce e1) (reduce e2)
| Var x     => x
| Ident c   => reflect (Ident c) end.
Definition Rewrite {t} (E:Expr t) : Expr t
  := λ var, reify (reduce (E (nbeT var t))).
```

Figure 2 PHOAS implementation of normalization by evaluation.

One subtlety hidden in Figure 2 in implicit arguments is in the final clause of `reduce`, where the two applications of the `Ident` constructor use different variable representations. With all those details hashed out, we can prove a pleasingly simple correctness theorem, with a lemma for each main definition, with inductive structure mirroring recursive structure of the definition, also appealing to correctness of last section's pattern-compilation operations. (We now use syntax $[\![\cdot]\!]$ for calls to `DenoteE`.)

$$\forall t, E : \texttt{Expr t}.\ [\![\texttt{Rewrite}(E)]\!] = [\![E]\!]$$

To understand how we now apply the soundness theorem in a tactic, it is important to note how the Coq kernel builds in reduction strategies. These strategies have, to an extent, been tuned to work well to show equivalence between a simple denotational-semantics application and the semantic value it produces. In contrast, it is rather difficult to code up one reduction strategy that works well for all partial-evaluation tasks. Therefore, we should restrict ourselves to (1) running full reduction in the style of functional-language interpreters and (2) running normal reduction on "known-good" goals like correctness of evaluation of a denotational semantics on a concrete input.

Operationally, then, we apply our tactic in a goal containing a term e that we want to partially evaluate. In standard proof-by-reflection style, we *reify* e into some E where $[\![E]\!] = e$, replacing e accordingly, asking Coq's kernel to validate the equivalence via standard reduction. Now we use the `Rewrite` correctness theorem to replace $[\![E]\!]$ with $[\![\texttt{Rewrite}(E)]\!]$. Next we ask the Coq kernel to simplify `Rewrite`(E) by *full reduction via native compilation*. Finally, where E' is the result of that reduction, we simplify $[\![E']\!]$ with standard reduction.

We have been discussing representation of bound variables. Also important is representation of constants (e.g., library functions mentioned in rewrite rules). They could also be given some explicit first-order encoding, but dispatching on, say, strings or numbers for constants would be rather inefficient in our generated code. Instead, we chose to have our Coq plugin generate a custom inductive type of constant codes, for each rewriter that we ask it to build with `Make`. As a result, dispatching on a constant can happen in constant time, based on whatever pattern-matching is built into the execution language (either the Coq kernel or the target language of extraction). To our knowledge, no past verified reduction tool in a proof assistant has employed that optimization.

4.2 Subterm Sharing Is Crucial

For some large-scale partial-evaluation problems, it is important to represent output programs with sharing of common subterms. Redundantly inlining shared subterms can lead to exponential increase in space requirements. Consider the Fiat Cryptography [8] example

of generating a 64-bit implementation of field arithmetic for the P-256 elliptic curve. The library has been converted manually to continuation-passing style, allowing proper generation of `let` binders, whose variables are often mentioned multiple times. We ran that old code generator (actually just a subset of its functionality, but optimized by us a bit further, as explained in Subsection 5.3) on the P-256 example and found it took about 15 seconds to finish. Then we modified reduction to inline `let` binders instead of preserving them, at which point the job terminated with an out-of-memory error, on a machine with 64 GB of RAM.

We see a tension here between performance and niceness of library implementation. When we built the original Fiat Cryptography, we found it necessary to CPS-convert the code to coax Coq into adequate reduction performance. Then all of our correctness theorems were complicated by reasoning about continuations. In fact, the CPS reasoning was so painful that at one point most algorithms in the template library were defined twice, once in continuation-passing style and once in direct-style code, because it was easier to prove the two equivalent and work with the non-CPS version than to reason about the CPS version directly. It feels like a slippery slope on the path to implementing a domain-specific compiler, rather than taking advantage of the pleasing simplicity of partial evaluation on natural functional programs. Our reduction engine takes shared-subterm preservation seriously while applying to libraries in direct style.

Our approach is `let`-lifting: we lift `let`s to top level, so that applications of functions to `let`s are available for rewriting. For example, we can perform the rewriting

$$\texttt{map}\ (\lambda x.\, y + x)\ (\texttt{let}\ z := e\ \texttt{in}\ [0; 1; z + 1]) \rightsquigarrow \texttt{let}\ z := e\ \texttt{in}\ [y; y + 1; y + (z + 1)]$$

using the rules

$$\texttt{map}\ ?f\ [] \to []\qquad \texttt{map}\ ?f\ (?x :: ?xs) \to f\ x :: \texttt{map}\ f\ xs\qquad ?n + 0 \to n$$

We define a telescope-style type family called `UnderLets`:

```
Inductive UnderLets {var} (T : Type) := Base (v : T)
| UnderLet {A} (e : @expr var A) (f : var A -> UnderLets T).
```

A value of type `UnderLets T` is a series of `let` binders (where each expression `e` may mention earlier-bound variables) ending in a value of type `T`.

Recall that the NbE type interpretation mapped base types to expression syntax trees. We add flexibility, parameterizing by a Boolean declaring whether to introduce telescopes.

```
Fixpoint nbeT' {var} (with_lets : bool) (t : type) := match t with
  | base t => if with_lets then @UnderLets var (@expr var t) else @expr var t
  | arrow s d => nbeT' false s -> nbeT' true d  end.
Definition nbeT := nbeT' false.      Definition nbeT_with_lets := nbeT' true.
```

There are cases where naive preservation of `let` binders blocks later rewrites from triggering and leads to suboptimal performance, so we include some heuristics. For instance, when the expression being bound is a constant, we always inline. When the expression being bound is a series of list "cons" operations, we introduce a name for each individual list element, since such a list might be traversed multiple times in different ways.

4.3 Rules Need Side Conditions

Many useful algebraic simplifications require side conditions. For example, bit-shifting operations are faster than divisions, so we might want a rule such as

$$?n/?m \to n \gg \log_2 m \quad \text{if} \quad 2^{\lfloor \log_2 m \rfloor} = m$$

The trouble is how to support predictable solving of side conditions during partial evaluation, where we may be rewriting in open terms. We decided to sidestep this problem by allowing side conditions only as executable Boolean functions, to be applied only to variables that are confirmed as *compile-time constants*, unlike Malecha and Bengtson [14] who support general unification variables. We added a variant of pattern variable that only matches constants. Semantically, this variable style has no additional meaning, and in fact we implement it as a special identity function (notated as an apostrophe) that should be called in the right places within Coq lemma statements. Rather, use of this identity function triggers the right behavior in our tactic code that reifies lemma statements.

Our reification inspects the hypotheses of lemma statements, using type classes to find decidable realizations of the predicates that are used, thereby synthesizing one Boolean expression of our deeply embedded term language, which stands for a decision procedure for the hypotheses. The `Make` command fails if any such expression contains pattern variables not marked as constants.

Hence, we encode the above rule as $\forall n, m.\ 2^{\lfloor \log_2({}'m) \rfloor} = {}'m \to n / {}'m = n \gg {}'(\log_2 m)$.

4.4 Side Conditions Need Abstract Interpretation

With our limitation that side conditions are decided by executable Boolean procedures, we cannot yet handle directly some of the rewrites needed for realistic compilation. For instance, Fiat Cryptography reduces high-level functional to low-level code that only uses integer types available on the target hardware. The starting library code works with arbitrary-precision integers, while the generated low-level code should be careful to avoid unintended integer overflow. As a result, the setup may be too naive for our running example rule $?n + 0 \to n$. When we get to reducing fixed-precision-integer terms, we must be legalistic:

$$\texttt{add\_with\_carry}_{64}(?n, 0) \to (0, n) \quad \text{if} \quad 0 \le n < 2^{64}$$

We developed a design pattern to handle this kind of rule.

First, we introduce a family of functions $\texttt{clip}_{l,u}$, each of which forces its integer argument to respect lower bound l and upper bound u. Partial evaluation is proved with respect to unknown realizations of these functions, only requiring that $\texttt{clip}_{l,u}(n) = n$ when $l \le n < u$. Now, before we begin partial evaluation, we can run a verified abstract interpreter to find conservative bounds for each program variable. When bounds l and u are found for variable x, it is sound to replace x with $\texttt{clip}_{l,u}(x)$. Therefore, at the end of this phase, we assume all variable occurrences have been rewritten in this manner to record their proved bounds.

Second, we proceed with our example rule refactored:

$$\texttt{add\_with\_carry}_{64}(\texttt{clip}_{,?l,{}'?u}(?n), 0) \to (0, \texttt{clip}_{l,u}(n)) \quad \text{if} \quad u < 2^{64}$$

If the abstract interpreter did its job, then all lower and upper bounds are constants, and we can execute side conditions straightforwardly during pattern matching.

See Appendix F in the full version for discussion of some further twists in the implementation.

5 Evaluation

Our implementation, available on GitHub at `mit-plv/rewriter@ITP-2022-perf-data` and with a roadmap in Appendix G of the full version, includes a mix of Coq code for the proved core of rewriting, tactic code for setting up proper use of that core, and OCaml plugin code

for the manipulations beyond the tactic language's current capabilities. We report here on evidence that the tool is effective, first in terms of productivity by users and then in terms of compile-time performance.

5.1 Iteration on the Fiat Cryptography Compiler

We ported Fiat Cryptography's core compiler functionality to use our framework. The result is now used in production by a number of open-source projects. We were glad to retire the CPS versions of verified arithmetic functions, which had been present only to support predictable reduction with subterm sharing. More importantly, it became easy to experiment with new transformations via proving new rewrite theorems, directly in normal Coq syntax, including the following, all justified by demand from real users:

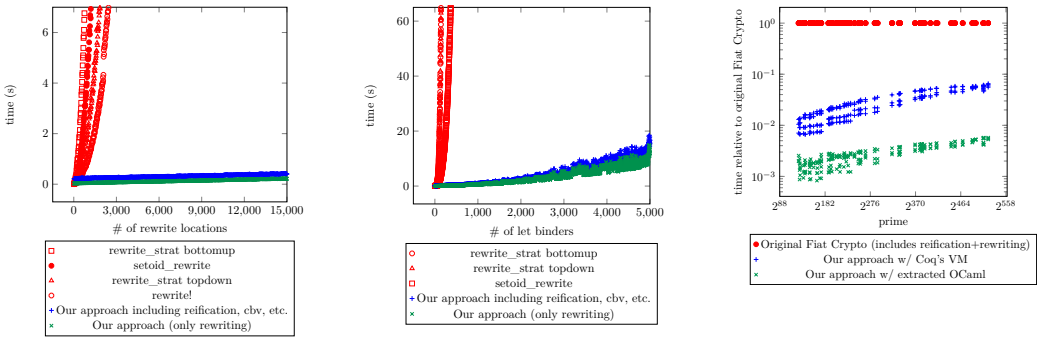
- Reassociating arithmetic to minimize the bitwidths of intermediate results
- Multiplication primitives that separately return high halves and low halves
- Strings and a "comment" function of type $\forall A.\ \texttt{string} \to A \to A$
- Support for bitwise exclusive-or
- A special marker to block C compilers from introducing conditional jumps in code that should be constant-time
- Eliding bitmask-with-constant operations that can be proved as no-ops
- Rules to introduce conditional moves (on supported platforms)
- New hardware backend, via rules that invoke special instructions of a cryptographic accelerator
- New hardware backend, with a requirement that all intermediate integers have the same bitwidth, via rules to break wider operations down into several narrower operations

5.2 Microbenchmarks

Now we turn to evaluating performance of generated compilers. We start with microbenchmarks focusing attention on particular aspects of reduction and rewriting, with Appendix C of the full version going into more detail, including on a few more benchmarks.

Our first example family, *nested binders*, has two integer parameters n and m. An expression tree is built with 2^n copies of an expression, which is itself a free variable with m "useless" additions of zero. We want to see all copies of this expression reduced to just the variable. Figure 3a on the following page shows the results for $n = 3$ as we scale m. The comparison points are Coq's `rewrite!`, `setoid_rewrite`, and `rewrite_strat`. The first two perform one rewrite at a time, taking minimal advantage of commonalities across them and thus generating quite large, redundant proof terms. The third makes top-down or bottom-up passes with combined generation of proof terms. For our own approach, we list both the total time and the time taken for core execution of a verified rewrite engine, without counting reification (converting goals to ASTs) or its inverse (interpreting results back to normal-looking goals). The comparison here is very favorable for our approach so long as $m > 2$. (See Appendix B.1 in the full version for more detailed plots.)

Now consider what happens when we use `let` binders to share subterms within repeated addition of zero, incorporating exponentially many additions with linearly sized terms. Figure 3b on the next page shows the results. The comparison here is again very favorable for our approach. The competing tactics spike upward toward timeouts at just a few hundred generated binders, while our engine is only taking about 10 seconds for examples with 5,000 nested binders.



(a) No binders. **(b)** Nested binders. **(c)** Fiat Cryptography.

Figure 3 Timing of different partial-evaluation implementations.

Although we have made our comparison against the built-in tactics `setoid_rewrite` and `rewrite_strat`, by analyzing the performance in detail, we can argue that these performance bottlenecks are likely to hold for any proof assistant designed like Coq. Detailed debugging reveals five performance bottlenecks in the existing tactics, discussed in Appendix A of the full version.

5.3 Macrobenchmark: Fiat Cryptography

Finally, we consider an experiment (described in more detail in Appendix B.2 of the full version) replicating the generation of performance-competitive finite-field-arithmetic code for all popular elliptic curves by Erbsen et al. [8]. In all cases, we generate essentially the same code as they did, so we only measure performance of the code-generation process. We stage partial evaluation with three different reduction engines (i.e., three `Make` invocations), respectively applying 85, 56, and 44 rewrite rules (with only 2 rules shared across engines), taking total time of about 5 minutes to generate all three engines. These engines support 95 distinct function symbols.

Figure 3c graphs running time of three different partial-evaluation and rewriting methods for Fiat Cryptography, as the prime modulus of arithmetic scales up. Times are normalized to the performance of the original method of Erbsen et al. [8], which relied on standard Coq reduction to evaluate code that had been manually written in CPS, followed by reification and a custom ad-hoc simplification and rewriting engine.

As the figure shows, our approach gives about a 10×–1000× speed-up over the original Fiat Cryptography pipeline. Inspection of the timing profiles of the original pipeline reveals that reification dominates the timing profile; since partial evaluation is performed by Coq's kernel, reification must happen *after* partial evaluation, and hence the size of the term being reified grows with the size of the output code. Also recall that the old approach required rewriting Fiat Cryptography's library of arithmetic functions in continuation-passing style, enduring this complexity in library correctness proofs, while our new approach applies to a direct-style library. Finally, the old approach included a custom reflection-based arithmetic simplifier for term syntax, run after traditional reduction, whereas now we are able to apply a generic engine that combines both, without requiring more than proving traditional rewrites.

The figure also confirms a clear performance advantage of running reduction in code extracted to OCaml, which is possible because our plugin produces verified code in Coq's functional language. The extracted version is about 10× faster than running in Coq's kernel.

6 Future Work

By far the biggest next step for our engine is to integrate abstract interpretation with rewriting and partial evaluation. We expect this would net us asymptotic performance gains as described in Appendix D of the full version. Additionally, it would allow us to simplify the phrasing of many of our post-abstract-interpretation rewrite rules, by relegating bounds information to side conditions rather than requiring that they appear in the syntactic form of the rule.

There are also a number of natural extensions to our engine. For instance, we do not yet allow pattern variables marked as "constants only" to apply to container datatypes; we limit the mixing of higher-order and polymorphic types, as well as limiting use of first-class polymorphism; we do not support rewriting with equalities of nonfully-applied functions; we only support decidable predicates as rule side conditions, and the predicates may only mention pattern variables restricted to matching constants; we have hardcoded support for a small set of container types and their eliminators; we support rewriting with equality and no other relations; and we require decidable equality for all types mentioned in rules.

References

1 Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In *Proc. TPHOLs*, 2008.

2 Nada Amin and Tiark Rompf. LMS-Verify: Abstraction without regret for verified systems programming. In *Proc. POPL*, 2017.

3 U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, July 1991. doi:10.1109/LICS.1991.151645.

4 Mathieu Boespflug. Efficient normalization by evaluation. In Olivier Danvy, editor, *Workshop on Normalization by Evaluation*, Los Angeles, United States, August 2009. URL: https://hal.inria.fr/inria-00434283.

5 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *Proc. CPP*, 2011.

6 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, British Columbia, Canada, September 2008. URL: http://adam.chlipala.net/papers/PhoasICFP08/.

7 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

8 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *IEEE Security & Privacy*, San Francisco, CA, USA, May 2019. URL: http://adam.chlipala.net/papers/FiatCryptoSP19/.

9 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proc. ICFP*, 2002.

10 Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In *Proc. TPHOLs*, 2007.

11 Jason Hickey and Aleksey Nogin. Formal compiler construction in a logical framework. *Higher-Order and Symbolic Computation*, 19(2):197–230, 2006. doi:10.1007/s10990-006-8746-6.

12 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, January 2014. URL: https://cakeml.org/popl14.pdf.

13 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. URL: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

14 Gregory Malecha and Jesper Bengtson. *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, chapter Extensible and Efficient Automation Through Reflective Tactics, pages 532–559. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49498-1_21.

15 Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46. ACM, 2008. URL: http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf.

16 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Proceedings of GPCE*, 2010. URL: https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf.

17 Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 111–121, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806596.1806611.