Accelerating Verified-Compiler Development with a Verified Rewriting Engine

ANONYMOUS AUTHOR(S)

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.

1 INTRODUCTION

Formally verified compilers like CompCert [Leroy 2009] and CakeML [Kumar et al. 2014] 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 [2010] 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 new work targeting that sort of compilation.

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 [Hickey and Nogin 2006]. 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 [Erbsen et al. 2019] 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, proving 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 in any way. 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, including efficient handling of intermediate code with thousands of nested variable binders and critical subterm sharing.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

We evaluate our builder toolkit by replacing a key component of Fiat Cryptography, a Coq library that generates code for big-integer modular arithmetic at the heart of elliptic-curve-cryptography algorithms. This domain-specific compiler has been adopted, for instance, in the Chrome Web browser, such that the majority of all HTTPS connections from browsers are now initiated using code generated (with proof) by Fiat Cryptography. 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 and later replace unsound optimizations with equivalent sound 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 the two most widely used curves (P-256 and Curve25519). Their execution timed out trying to compile code for the third most widely used curve (P-384). Using our 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 the theorem statement and proof for every arithmetic combinator.

1.1 Related Work, Take One

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 input-output behavior of the higher-order functional programs we started with. What good ideas from the literature can we build on?

Hickey and Nogin [2006] 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. This suggests also that they cannot use rewrite rules which are only sound in the presence of side conditions, as such side-condition checking could not happen in the engine driving the rewrites. Overall, their work complements ours; we focus on how to build a scalable engine for building a proven-correct compiler around shallowly embedded rewrite rules in a proof assistant like Coq; they focus on what rewrite rules are necessary for implementing a general-purpose compiler.

$\mathcal{R}_{tac}$ [Malecha and Bengtson 2016] 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 $\mathcal{R}_{tac}$, and we expect there would be other bottlenecks working with large, deeply nested terms.

Aehlig et al. [2008] came closest to a fitting approach, using *normalization by evaluation (NbE)* [Berger and Schwichtenberg 1991] 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.

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. We will demonstrate all of these pieces working together well enough to be adopted by several high-profile open-source projects.

1.2 Our Solution

Our variant on the technique of Aehlig et al. [2008] 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. [2008] 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 [2008]). We also improve on Coq-specific related work (e.g., of Malecha and Bengtson [2016]) 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 [2009]).

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 [2016]). For instance, while the reported performance experiments of Aehlig et al. [2008] 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. Fiat Cryptography's arithmetic routines rely on significant sharing of this kind.
- Unconditional rewrite rules are in general insufficient, and we need *rules with side conditions*. For instance, in Fiat Cryptography, some rules for simplifying modular arithmetic depend on proofs that operations in subterms do not overflow.

- 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* that can analyze whole programs to support reduction.

Briefly, our respective solutions to these problems are the *parametric higher-order abstract syntax (PHOAS)* [Chlipala 2008] 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 related work (section 6) and conclusions. Our implementation is included as an anonymous supplement.

2 A MOTIVATING EXAMPLE

Our style of compilation involves source programs that mix higher-order functions and inductive datatypes. We want to compile them to C code, reducing away all uses of fancier features while seizing opportunities for arithmetic simplification. Let us walk through a small example that nonetheless illustrates several key challenges.

```
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 traverse that list, building up a list of 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. That actual translation we leave to other small compiler phases outside the scope of this paper.

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 a proof of its equivalence to the original function.

```
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 even more pragmatic 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 that doubles as a dependently typed functional language. 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 for the extracted version, since extraction itself is not proved.

3 THE STRUCTURE OF A REWRITER

We are mostly guided by Aehlig et al. [2008] 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 operational semantics. This operational semantics is trusted, much as Coq extraction is trusted.
(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, against the ML operational semantics.

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 with respect to ML semantics.
(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) Combine 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 [Haftmann and Nipkow 2007]. Trusted code compiles the ML code into the concrete syntax of a mainstream ML language, Standard ML in their case, and compiles it with an off-the-shelf compiler. The output of that compiled program is then passed back over to the tactic, in terms of an axiomatic assertion that the ML semantics really yields that answer.

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, rather than through a plugin.
- We do *compile-time specialization of the reduction engine* to sets of rewrite rules, removing overheads of generality.

3.1 Our Approach in Nine 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. In order to build a precomputed rewriter with the `Make` command, the following actions are performed:

(1) The given lemma statements are scraped for which named functions and types the rewriter package will support.
(2) Inductive types enumerating all available primitive types and functions are emitted. This allows us to achieve the performance gains attributed in Boespflug [2009] to having native metalanguage constructors for each constant without needing to manually code an enumeration.
(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 representable primitive 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. Each instance of the former involves wrapping the user-provided proof with the right adapter to apply to the reified version. Automating this step allows rewrite rules to be proven in terms of their shallow embedding, which drastically accelerates iteration on the set of rewrite rules.

(5) The definitions needed to perform reification and rewriting and the lemmas needed to prove correctness are assembled into a single package that can be passed by name to the rewriting tactic.

When we want to rewrite with a rewriter package in a goal, the following steps are performed:

(1) We rearrange the goal into a single logical formula: all free-variable quantification in the proof context is replaced by changing the equality goal into an equality between two functions (taking the free variables as inputs).
(2) We reify the side of the goal we want to simplify, using the inductive codes in the specified package. That side of the goal is then replaced with a call to a denotation function on the reified version.
(3) We use a theorem stating that rewriting preserves denotations of well-formed terms to replace the denotation subterm with the denotation of the rewriter applied to the same reified term. We use Coq's built-in full reduction (**`vm_compute`**) to reduce the application of the rewriter to the reified term.
(4) Finally, we run **`cbv`** (a standard call-by-value reducer) to simplify away the invocation of the denotation function on the concrete syntax tree from rewriting.

The object language of our rewriter is nearly simply typed, with limited support for calling polymorphic functions.

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

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 +, ·, and literal constants. We write $[\![e]\!]$ for a standard denotational semantics.

3.2 Pattern-Matching Compilation and Evaluation

Aehlig et al. [2008] 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. Indeed, Coq's logic only includes primitive pattern-matching constructs to match one constructor at a time.

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 [2016], 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 [2008], 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). We have not yet implemented any of the optimizations described therein for finding *minimal* decision trees.

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. Said another way, any rewrite-rule replacement expression must match the semantics of the rewrite-rule pattern.

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` decision tree.
- `Swap i cont`: Swap the first element of the vector with the i^{th} element (0-indexed) and keep going with `cont`.

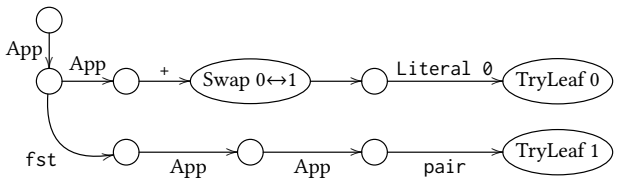
Consider the encoding of two simple example rewrite rules, where we follow Coq's $\mathcal{L}_{tac}$ language in prefacing pattern variables with question marks.

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

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) [Berger and Schwichtenberg 1991] 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. The simply typed λ-calculus syntax we use 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 the result type of an intermediate interpreter.

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

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 to handle our full problem domain.

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 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 under the name rewrite-head. Recall that this function only tries to apply rewrite rules at the top level of its input.

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:

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

$$
\begin{aligned}
\text{reify}_t &: \text{NbE}_t(t) \to \text{expr}(t) \\
\text{reify}_{t_1 \to t_2}(f) &= \lambda v.\, \text{reify}_{t_2}(f(\text{reflect}_{t_1}(v))) \\
\text{reify}_b(f) &= f \\
\\
\text{reflect}_t &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reflect}_{t_1 \to t_2}(e) &= \lambda x.\, \text{reflect}_{t_2}(e(\text{reify}_{t_1}(x)) \\
\text{reflect}_b(e) &= e \\
\\
\text{reduce} &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reduce}(\lambda v.\ e) &= \lambda x.\ \text{reduce}([x/v]e) \\
\text{reduce}(e_1\ e_2) &= (\text{reduce}(e_1))\ (\text{reduce}(e_2)) \\
\text{reduce}(x) &= x \\
\text{reduce}(c) &= \text{reflect}(c) \\
\\
\text{NbE} &: \text{expr}(t) \to \text{expr}(t) \\
\text{NbE}(e) &= \text{reify}(\text{reduce}(e))
\end{aligned}
$$

Fig. 1. Implementation of normalization by evaluation

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, though we work up to convincing Coq of that fact.

The details so far are essentially the same as in the approach of Aehlig et al. [2008]. 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 rule-specific ML functions that diverge themselves. In contrast, we need to convince Coq up-front that our interleaved λ-term normalization and algebraic simplification always terminate. Additionally, we need to prove that our rewriter preserves denotations of terms, which can easily devolve into tedious binder bookkeeping, depending on encoding.

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. [2008] 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. [2008] reported a significant performance improvement changing variable nodes from using strings

to using de Bruijn indices [De Bruijn 1972]. 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) [Chlipala 2008] 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 the variable representation. Especially note how this choice makes rigorous the convention we followed in the prior section (e.g., in the suspicious function-abstraction clause of function 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. 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))).
```

One subtlety hidden above 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`.)

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

Even before getting to the correctness theorem, we needed to convince Coq that the function terminates. While for Aehlig et al. [2008], a termination proof would have been a whole separate enterprise, it turns out that PHOAS and NbE line up so well that Coq accepts the above code with no additional termination proof; each key function is obviously structurally recursive on either a type or an expression. As a result, the Coq kernel is ready to run our **`Rewrite`** procedure during checking.

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 $[\![\textbf{Rewrite}(E)]\!]$. Next we may ask the Coq kernel to simplify $\textbf{Rewrite}(E)$ by *full reduction via compilation to native code*, since we carefully designed $\textbf{Rewrite}(E)$ and its dependencies to produce closed syntax trees, so that reduction will not get stuck pattern-matching on free variables. Finally, where E' is the result of that reduction, we simplify $[\![E']\!]$ with standard reduction, producing a normal-looking Coq term.

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 [Erbsen et al. 2019] 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 their 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 reduction job terminated with an out-of-memory error, on a machine with 64 GB of RAM. (The successful run uses under 2 GB.)

We see a tension here between performance and niceness of library implementation. The Fiat Cryptography authors found it necessary to CPS-convert their code to coax Coq into adequate reduction performance. Then all of their correctness theorems were complicated by reasoning about continuations. In fact, the CPS reasoning was so painful that at one point most algorithms in their 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

$$\begin{aligned} &\texttt{map}\ (\lambda x.\, y + x)\ (\texttt{let}\ z := e\ \texttt{in}\ [0; 1; 2; z; z + 1]) \\ \rightsquigarrow\ &\texttt{let}\ z := e\ \texttt{in}\ [y; y + 1; y + 2; y + z; y + (z + 1)] \end{aligned}$$

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`. It is easy to build various "smart constructors" working with this type, for instance to construct a function application by lifting the `lets` of both function and argument to a common top level.

Such constructors are used to implement an NbE strategy that outputs `UnderLets` telescopes. Recall that the NbE type interpretation mapped base types to expression syntax trees. We now parameterize that type interpretation by a Boolean declaring whether we want 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

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

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 [2016] 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 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. We introduce a notation where a prefixed apostrophe signals a call to the "constants only" function.

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. Therefore, matching of rules can safely run side conditions, knowing that Coq's full-reduction engine can determine their truth efficiently.

Hence, we encode the above rule as

$$\forall n, m.\ 2^{\lfloor \log_2 ('m) \rfloor} = 'm \longrightarrow n/'m = n \gg '(\log_2 m)$$

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:

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

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:

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

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.

4.5 Limitations and Preprocessing

We now note some details of the rewriting framework that were previously glossed over, which are useful for using the code or implementing something similar, but which do not add fundamental capabilities to the approach. Although the rewriting framework does not support dependently typed constants, we can automatically preprocess uses of eliminators like `nat_rect` and `list_rect` into nondependent versions. The tactic that does this preprocessing is extensible via $\mathcal{L}_{tac}$'s reassignment feature. Since pattern-matching compilation mixed with NbE requires knowing how many arguments a constant can be applied to, internally we must use a version of the recursion principle whose type arguments do not contain arrows; current preprocessing can handle recursion principles with either no arrows or one arrow in motives.

Recall from section 2 that `eval_rect` is a definition provided by our framework for eagerly evaluating recursion associated with certain types. It functions by triggering typeclass resolution for the lemmas reducing the recursion principle associated to the given type. We provide instances for `nat`, `prod`, `list`, `option`, and `bool`. Users may add more instances if they desire.

Recall again from section 2 that we use `ident.eagerly` to ask the reducer to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree. Our current version only allows a limited, hard-coded set of eliminators with `ident.eagerly` (`nat_rect` on return types with either zero or one arrows, `list_rect` on return types with either zero or one arrows, and `List.nth_default`), but nothing in principle prevents automatic generation of the necessary code.

We define a constant `Let_In` which we use for writing `let ⋯ in ⋯` expressions that do not reduce under ζ (Coq's reduction rule for **`let`**-inlining). Throughout most of this paper, anywhere that `let ⋯ in ⋯` appears, we have actually used `Let_In` in the code. It would alternatively be possible to extend the reification preprocessor to automatically convert `let ⋯ in ⋯` to

`Let_In`, but this strategy may cause problems when converting the interpretation of the reified term with the prereified term, as Coq's conversion does not allow fine-tuning of when to inline or unfold **`let`**s.

5 EVALUATION

Our implementation, attached to this submission as an anonymized supplement with a roadmap in Appendix E, 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 current capabilities of the tactic language. 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 continuation-passing-style 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 and proof style.

5.1.1 Reassociation of Arithmetic. One of the simplest and most minor examples of iterating on the Fiat Cryptography compiler is that of reassociating multiplication under certain conditions. In particular, if $y \cdot z$ fits in 64 bits, but $x \cdot y$ and $x \cdot y \cdot z$ both take 128 bits, then we want to emit $x \cdot (y \cdot z)$ rather than $(x \cdot y) \cdot z$. This way, the C compiler will emit one single-width (64-bit) multiplication and one double-width (128-bit) multiplication, rather than two double-width multiplications. The rule we use for this is roughly

$$\forall x, y, c.\ |\texttt{'}c| \leq 2^8 \rightarrow (x \cdot y) \cdot \texttt{'}c = x \cdot (y \cdot \texttt{'}c)$$

Recall from subsection 4.3 that the prefix `'` is used to indicate that some pattern must be a compile-time constant. Since this is just a notation for a specially named identity function, we can prove this rule just by applying associativity of multiplication in the integers.

5.1.2 New Constructs in the Frontend. Adding support for new constructs in the code being compiled is quite simple: just add the types and constants to the list used to generate the rewriter package. The original Fiat Cryptography compiler proof required making changes in several places and modifying several proofs per new identifier. We added support for the following constructs in a purely local, modular way:

- multiplication primitives that separately return the high half and low half of a double-width multiplication
- strings and a "comment" function of type $\forall A.\ \texttt{string} \rightarrow A \rightarrow A$
- support for bitwise exclusive-or
- an identity function which prints in the backend as a call to some inline assembly defined in the header which prevents C compilers from introducing conditional jumps into code that ought to be constant-time

5.1.3 New Middle-End Variations. Performance-debugging runs of our generated compilers led us to various ideas for improvement, which were quite easy to add locally. For instance, some bitmasking operations (applying bitwise "and" with constants) can be proved to be no-ops, based on properties of the terms being masked. This realization led us to add several new rules.

On our way to code generation for different target languages, we often make different decisions based on which broad families those targets belong to. For instance, only some platforms support conditional-move instructions. We added easy compiler configuration of whether to try to use conditional moves. Based on such a flag, we choose different generated rewriters, which have been described with largely the same rewrite rules, just including or omitting conditional-move-oriented rules as appropriate.

5.1.4 New Backends. Two new backends were added after this framework was integrated into Fiat Cryptography. A backend is one set of rewrite rules, motivated by a specific target machine, that can be composed with the frontend rules that are independent of target. Each such combination can be used to generate a separate (proved) command-line tool.

The first new backend involved a separate set of identifiers for an unusual hardware platform with special instructions to accelerate cryptographic arithmetic. The rewrite rules we added merely translated the constructs in our IR to the corresponding identifiers in this output language.

The other backend we added required all arithmetic operations to operate on a single bitwidth, rather than mixing, say, 64-bit and 128-bit integers in the same function. To perform this translation, we add rewrite rules for all operations that produce integers of different bitwidths than they consume. These rewrite rules replace such an operation with a couple of variable assignments in terms of operations on the smaller bitwidth, together with a special function `Z.combine_at_bitwidth` which is defined to combine two smaller-bitwidth integers into a larger-bitwidth integer. We also added rewrite rules about how to push various operations on the larger bitwidth through this `Z.combine_at_bitwidth` construct (for example, a rule about how to turn a multiplication of two outputs of `Z.combine_at_bitwidth` into `Z.combine_at_bitwidth` applied to multiplications of the high and low bits of the inputs). By designating `Z.combine_at_bitwidth` as an identifier that should be inlined rather than let-bound during the rewriting, we automatically get a complete compilation of bitwidth-splitting without having to run the rewrite rules repeatedly.

5.1.5 Moving Rules Involving Carries. We originally had rules like "adding 0 to x produces just x, with no carry." This rule is not true in general, because we must encode carrying as happening at a particular bitwidth, while we have no guarantee that x fits within that bitwidth. The correct rule has a precondition: that x is between 0 and 2^{64}.

We discovered this issue when trying to prove our rewrite rules correct. As a result, we had to move this sort of rule from happening before abstract interpretation to happening after abstract interpretation. Since the passes are just defined as lists of rewrite rules, moving the rules was quite simple. (Rephrasing them to talk about the outputs of abstract interpretation was somewhat painful, though, because the proof assistant did not enforce the conventions we were using for where to store abstract-interpretation information. We hypothesize that a more uniform approach to integrating abstract interpretation with rewriting and partial evaluation would solve this problem.)

5.1.6 Fusing Compiler Passes. When we moved the aforementioned constant-folding rules from before abstract interpretation to after it, the performance of our compiler on Word-by-Word Montgomery code synthesis decreased significantly. (The generated code did not change.) We discovered that the number of variable assignments in our intermediate code was quartic in the number of bits in the prime, while the number of variable assignments in the generated code is only quadratic. The performance numbers we measured supported this theory: the overall running time of synthesizing code for a prime near 2^k jumped from $\Theta(k^2)$ to $\Theta(k^4)$ when we made this change. We believe that fusing abstract interpretation with rewriting and partial evaluation would allow us to fix this asymptotic-complexity issue.

To make this situation more concrete, consider the following example: Fiat Cryptography uses abstract interpretation to perform bounds analysis; each expression is associated with a range that describes the lower and upper bounds of values that expression might take on. Abstract interpretation on addition works as follows: if we have that $x_\ell \le x \le x_u$ and $y_\ell \le y \le y_u$, then we have that $x_\ell + y_\ell \le x + y \le x_u + y_u$. Performing bounds analysis on + requires two additions. We might have an arithmetic simplification that says that $x + y = x$ whenever we know that $0 \le y \le 0$. If we perform the abstract interpretation and then the arithmetic simplification, we perform two additions (for the bounds analysis) and then two comparisons (to test the lower and upper bounds of y for equality with 0). We cannot perform the arithmetic simplification before abstract interpretation, because we will not know the bounds of y. However, if we perform the arithmetic simplification for each expression after performing bounds analysis on its *subexpressions* and only after this perform abstract interpretation on the resulting expression, then we need not use any additions to compute the bounds of $x + y$ when $0 \le y \le 0$, since the expression will just become x.

Another essential pass to fuse with rewriting and partial evaluation is let-lifting. Unless all of the code is CPS-converted ahead of time, attempting to do let-lifting via rewriting, as must be done when using **`setoid_rewrite`**, `rewrite_strat`, or $\mathcal{R}_{tac}$, results in slower asymptotics. This pattern is already apparent in the `LiftLetsMap` / "Binders and Recursive Functions" example in subsubsection 5.2.4. We achieve linear performance in $n \cdot m$ when ignoring the final **`cbv`**, while **`setoid_rewrite`** and `rewrite_strat` are both cubic. The rewriter in $\mathcal{R}_{tac}$ cannot possibly achieve better than $O\left(n \cdot m^2\right)$ unless it can be sublinear in the number of rewrites, because our rewriter gets away with a constant number of rewrites (four), plus evaluating recursion principles for a total amount of work $O(n \cdot m)$. But without primitive support for let-lifting, it is instead necessary to lift the lets by rewrite rules, which requires $O\left(n \cdot m^2\right)$ rewrites just to lift the lets. The analysis is thus: running `make` simply gives us m nested applications of `map_dbl` to a length-n list. To reduce a given call to `map_dbl`, all existing let-binders must first be lifted (there are $n \cdot k$ of them on the k-innermost-call) across `map_dbl`, one-at-a-time. Then the `map_dbl` adds another n let binders, so we end up doing $\sum_{k=0}^{m} n \cdot k$ lifts, i.e., $n \cdot m(m+1)/2$ rewrites just to lift the lets.

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 going into more detail.

5.2.1 Rewriting Without Binders. Consider the code defined by the expression $\text{tree}_{n,m}(v)$ in Figure 2. We want to remove all of the + 0s. There are $\Theta(m \cdot 2^n)$ such rewriting locations. We can start from this expression directly, in which case reification alone takes as much time as Coq's `rewrite`. As the reification method was not especially optimized, and there exist fast reification methods [Gross et al. 2018], we instead start from a call to a recursive function that generates such an expression.

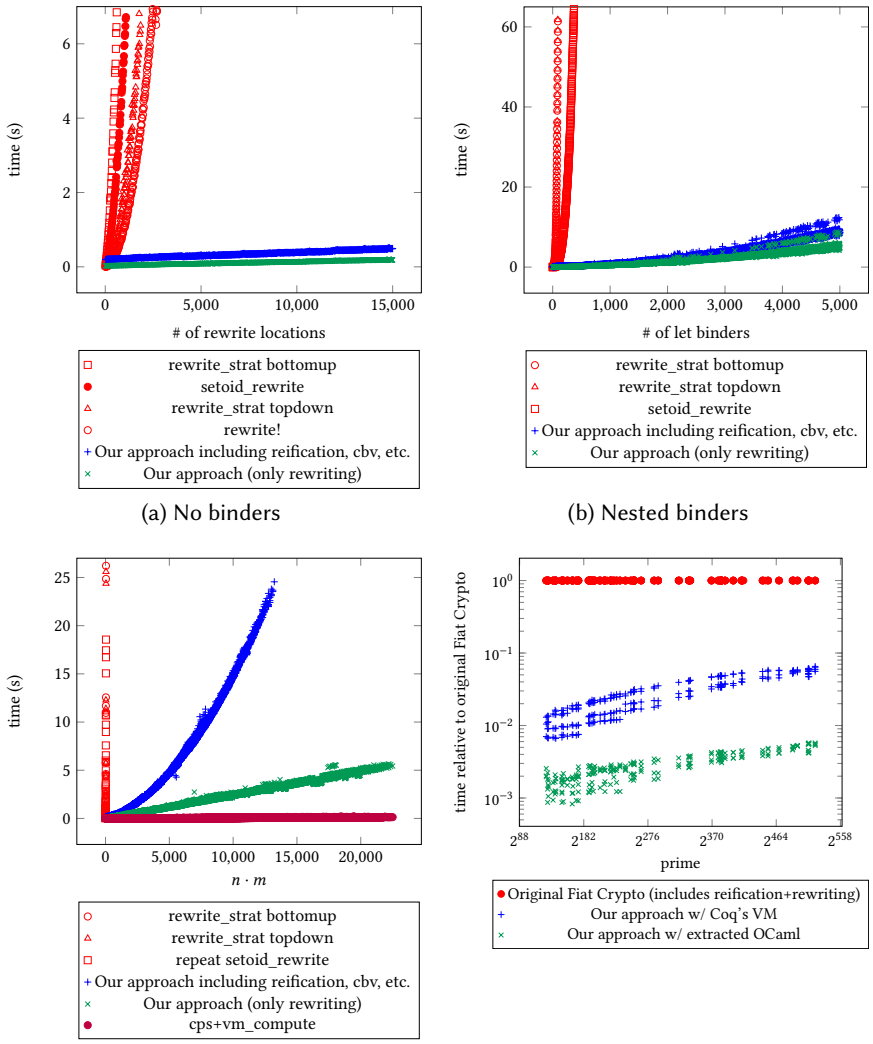
$$\text{iter}_m(v) = v + \underbrace{0 + 0 + \cdots + 0}_{m}$$
$$\text{tree}_{0,m}(v) = \text{iter}_m(v + v)$$
$$\text{tree}_{n+1,m}(v) = \text{iter}_m(\text{tree}_{n,m}(v) + \text{tree}_{n,m}(v))$$

Fig. 2. Expressions computing initial code

Figure 3a on the facing 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 $m > 2$. The competing tactics spike upward toward timeouts at just around a thousand rewrite locations, while our engine is still under two seconds for examples with tens of thousands of rewrite locations. When $m < 2$, Coq's `rewrite!` tactic does a little bit better than our engine, corresponding roughly to the overhead incurred by our term representation (which, for example, stores the types at every application node) when most of the term is in fact unchanged by rewriting. See subsection B.1for more detailed plots.



(a) No binders (b) Nested binders

(c) Binders and recursive functions (d) Fiat Cryptography

Fig. 3. Timing of different partial-evaluation implementations

5.2.2 Rewriting Under Binders. Consider now the code in Figure 4, which is a version of the code above where redundant expressions are shared via **`let`** bindings.

Figure 3b on the previous 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.

$$\begin{array}{l}\texttt{let } v_1 := v_0 + v_0 + 0 \texttt{ in} \\ \vdots \\ \texttt{let } v_n := v_{n-1} + v_{n-1} + 0 \texttt{ in} \\ v_n + v_n + 0\end{array}$$

Fig. 4. Initial code

5.2.3 Performance Bottlenecks of Proof-Producing Rewriting. 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 rewriting tactics, which we discuss in Appendix A.[1]

5.2.4 Binders and Recursive Functions. The next experiment uses the code in Figure 5. Note that the `let ⋯ in ⋯` binding blocks further reduction of map_dbl when we iterate it m times in make, and so we need to take care to preserve sharing when reducing here.

Figure 3c compares performance between our approach, `repeat setoid_rewrite`, and two variants of `rewrite_strat`. Additionally, we consider another option, which was adopted by Fiat Cryptography at a larger scale: rewrite our functions to improve reduction behavior. Specifically, both functions are rewritten in continuation-passing style, which makes them harder to read and reason about but allows standard VM-based reduction to achieve good performance. The figure shows that `rewrite_strat` variants are essentially unusable for this example, with `setoid_rewrite` performing only marginally better, while our approach applied to the original, more readable definitions loses ground steadily to VM-based reduction on CPS'd code. On the largest terms ($n \cdot m > 20,000$), the gap is 6s vs. 0.1s of compilation time, which should often be acceptable in return for simplified coding and proofs, plus the ability to mix proved rewrite rules with built-in reductions. Note that about 99% of the difference between the full time of our method and just the rewriting is spent in the final **`cbv`** at the end, used to denote our output term from reified syntax. We blame this performance on the unfortunate fact that reduction in Coq is quadratic in the number of nested binders present; see Coq bug #11151. This bug has since been fixed, as of Coq 8.14; see Coq PR #13537. See subsection C.3 for more on this microbenchmark.

$$\begin{aligned}\text{map\_dbl}(\ell) &= \begin{cases}[] & \text{if } \ell = [] \\ \texttt{let } y := h + h \texttt{ in} \\ y :: \text{map\_dbl}(t) & \text{if } \ell = h :: t\end{cases} \\ \text{make}(n, m, v) &= \begin{cases}\underbrace{[v, \ldots, v]}_{n} & \text{if } m = 0 \\ \text{map\_dbl}(\text{make}(n, m-1, v)) & \text{if } m > 0\end{cases} \\ \text{example}_{n,m} &= \forall v,\ \text{make}(n, m, v) = []\end{aligned}$$

Fig. 5. Initial code for binders and recursive functions

5.2.5 Full Reduction. The final experiment involves full reduction in computing the Sieve of Eratosthenes, taking inspiration on benchmark choice from Aehlig et al. [2008]. We find in Figure 6 that we are slower than **`vm_compute`**, **`native_compute`**, and **`cbv`**, but faster than **`lazy`**, and of course much faster than **`simpl`** and **`cbn`**, which are quite slow.

5.3 Macrobenchmark: Fiat Cryptography

[1] This forward reference goes to an appendix included within the main submission page limit, to avoid interrupting the flow in presenting the most important results.

Finally, we consider an experiment (described in more detail in subsection B.2) replicating the generation of performance-competitive finite-field-arithmetic code for all popular elliptic curves by Erbsen et al. [2019]. 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 3d on page 19 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. [2019], 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.

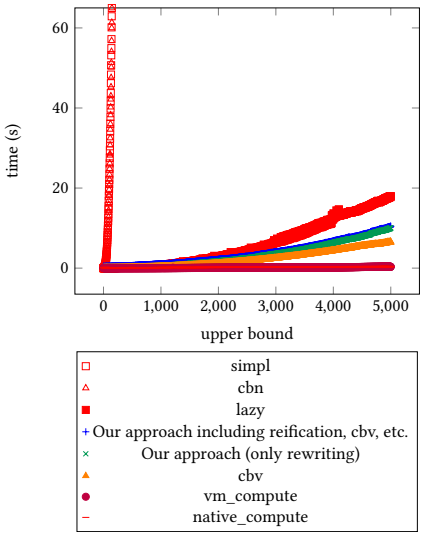


Fig. 6. Full evaluation, Sieve of Eratosthenes

As the figure shows, our approach gives about a 10×–1000× speed-up over the original Fiat Cryptography pipeline. (We used the same set of prime moduli as in the experiments run by Erbsen et al. [2019], which were chosen based on searching the archives of an elliptic-curves mailing list for all prime numbers.) 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 rewrite rules.

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 RELATED WORK, TAKE TWO

We have already discussed the work of Aehlig et al. [2008], which introduced the basic structure that our engine shares, but which required a substantially larger trusted code base, did not tackle certain challenges in scaling to large partial-evaluation problems, and did not report any performance experiments in partial evaluation.

We have also mentioned $\mathcal{R}_{tac}$ [Malecha and Bengtson 2016], which provides tactics as proved functions. Its rewrite tactic does not support preservation of subterm sharing, which is performance-prohibitive for the kind of compilation that interested us. There is also no support (in $\mathcal{R}_{tac}$ or the standard Coq tactics) for rewriting in normalization-by-evaluation order, which turns out to be perfect for good performance scaling in these compilers.

Our implementation builds on fast full reduction in Coq's kernel, via a virtual machine [Grégoire and Leroy 2002] or compilation to native code [Boespflug et al. 2011]. Especially the latter is similar

in adopting an NbE style 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, where functions that should not be evaluated at compile time must have properly opaque definitions that the evaluator will not consult. Neither implementation involved a machine-checked proof.

Boespflug [2009] discusses optimizations on (untyped) NbE for use in proof assistants. They mention the reuse of metalanguage pattern-matching facilities on datatypes as an optimization, wherein using metalanguage constructors for constants breaks modularity of the reduction engine. We achieve this optimization while avoiding this issue by automatically creating on-the-fly a datatype of constants as a sort of early compilation phase. This provides a way to have constructors in the object language be represented as constructors in the metalanguage, in a typed NbE setting, without breaking modularity nor incurring extra work for the user, for a minor upfront time cost.

A variety of forms of pragmatic partial evaluation have been demonstrated, with Lightweight Modular Staging [Rompf and Odersky 2010] in Scala as one of the best-known current examples. A kind of type-based overloading for staging annotations is used to smooth the rough edges in writing code that manipulates syntax trees. The LMS-Verify system [Amin and Rompf 2017] 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, justifying investment in verified partial evaluators.

7 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 subsubsection 5.1.6. 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 (e.g., subset inclusion); and we require decidable equality for all types mentioned in rules. It may be helpful to design an engine that lifts some or all of these limitations, building on the basic structure that we present here.

REFERENCES

Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. 2008. A Compiled Implementation of Normalization by Evaluation. In *Proc. TPHOLs*.

Nada Amin and Tiark Rompf. 2017. LMS-Verify: Abstraction without Regret for Verified Systems Programming. In *Proc. POPL*.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proc. POPL*. 3–15. https://www.cis.upenn.edu/~bcpierce/papers/binders.pdf

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

Mathieu Boespflug. 2009. Efficient normalization by evaluation. In *Workshop on Normalization by Evaluation* (2009-08), Olivier Danvy (Ed.). Los Angeles, United States. https://hal.inria.fr/inria-00434283

Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. 2011. Full Reduction at Full Throttle. In *Proc. CPP*.

Adam Chlipala. 2008. 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. http://adam.chlipala.net/papers/PhoasICFP08/

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

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises. In *IEEE Security & Privacy*. San Francisco, CA, USA. http://adam.chlipala.net/papers/FiatCryptoSP19/

Jason Gross, Andres Erbsen, and Adam Chlipala. 2018. Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac. In *Proc. ITP*. http://adam.chlipala.net/papers/ReificationITP18/

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

Florian Haftmann and Tobias Nipkow. 2007. A Code Generator Framework for Isabelle/HOL. In *Proc. TPHOLs*.

Jason Hickey and Aleksey Nogin. 2006. Formal Compiler Construction in a Logical Framework. *Higher-Order and Symbolic Computation* 19, 2 (2006), 197–230. https://doi.org/10.1007/s10990-006-8746-6

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014-01). ACM Press, 179–191. https://cakeml.org/popl14.pdf

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf

Gregory Malecha and Jesper Bengtson. 2016. *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*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Extensible and Efficient Automation Through Reflective Tactics, 532–559. https://doi.org/10.1007/978-3-662-49498-1_21

Luc Maranget. 2008. Compiling Pattern Matching to Good Decision Trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM, 35–46. http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf

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

Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada, 2010) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/1806596.1806611

A PERFORMANCE BOTTLENECKS OF PROOF-PRODUCING REWRITING

Although we have made our performance comparison against the built-in Coq 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 rewriting tactics.

A.1 Bad performance scaling in sizes of existential-variable contexts

We found that even when there are no occurrences fully matching the rule, **`setoid_rewrite`** can still be *cubic* in the number of binders (or, more accurately, quadratic in the number of binders with an additional multiplicative linear factor of the number of head-symbol matches). Rewriting without any successful matches takes nearly as much time as **`setoid_rewrite`** in this microbenchmark; by the time we are looking at goals with 400 binders, the difference is less than 5%.

We posit that this overhead comes from **`setoid_rewrite`** looking for head-symbol matches and then creating evars (existential variables) to instantiate the arguments of the lemmas for each head-symbol-match location; hence even if there are no matches of the rule as a whole, there may still be head-symbol matches. Since Coq uses a locally nameless representation [Aydemir et al. 2008] for its terms, evar contexts are necessarily represented as *named* contexts. Representing a substitution between named contexts takes linear space, even when the substitution is trivial, and hence each evar incurs overhead linear in the number of binders above it. Furthermore, fresh-name generation in Coq is quadratic in the size of the context, and since evar-context creation uses fresh-name generation, the additional multiplicative factor likely comes from fresh-name generation. (Note, though, that this pattern suggests that the true performance is quartic rather than merely cubic. However, doing a linear regression on a log-log of the data suggests that the performance is genuinely cubic rather than quartic.)

Note that this overhead is inherent to the use of a locally nameless term representation. To fix it, Coq would likely have to represent identity evar contexts using a compact representation, which is only naturally available for de Bruijn representations. Any rewriting system that uses unification variables with a locally nameless (or named) context will incur at least quadratic overhead on this benchmark.

Note that `rewrite_strat` uses exactly the same rewriting engine as **`setoid_rewrite`**, just with a different strategy. We found that **`setoid_rewrite`** and `rewrite_strat` have identical performance when there are no matches and generate identical proof terms when there are matches. Hence we can conclude that the difference in performance between `rewrite_strat` and **`setoid_rewrite`** is entirely due to an increased number of failed rewrite attempts.

A.2 Proof-term size

Setting aside the performance bottleneck in constructing the matches in the first place, we can ask the question: how much cost is associated to the proof terms? One way to ask this question in Coq is to see how long it takes to run **`Qed`**. While **`Qed`** time is asymptotically better, it is still quadratic in the number of binders. This outcome is unsurprising, because the proof-term size is quadratic in the number of binders. On this microbenchmark, we found that **`Qed`** time hits one second at about 250 binders, and using the best-fit quadratic line suggests that it would hit 10 seconds at about 800 binders and 100 seconds at about 2 500 binders. While this may be reasonable for the microbenchmarks, which only contain as many rewrite occurrences as there are binders, it would become unwieldy to try to build and typecheck such a proof with a rule for every primitive reduction step, which would be required if we want to avoid manually CPS-converting the code in Fiat Cryptography.

The quadratic factor in the proof term comes because we repeat subterms of the goal linearly in the number of rewrites. For example, if we want to rewrite `f (f x)` into `g (g x)` by the equation `∀ x, f x = g x`, then we will first rewrite `f x` into `g x`, and then rewrite `f (g x)` into `g (g x)`. Note that `g x` occurs three times (and will continue to occur in every subsequent step).

A.3 Poor subterm sharing

How easy is it to share subterms and create a linearly sized proof? While it is relatively straightforward to share subterms using **`let`** binders when the rewrite locations are not under any binders, it is not at all obvious how to share subterms when the terms occur under different binders. Hence any rewriting algorithm that does not find a way to share subterms across different contexts will incur a quadratic factor in proof-building and proof-checking time, and we expect this factor will be significant enough to make applications to projects as large as Fiat Crypto infeasible.

A.4 Overhead from the `let` typing rule

Suppose we had a proof-producing rewriting algorithm that shared subterms even under binders. Would it be enough? It turns out that even when the proof size is linear in the number of binders, the cost to typecheck it in Coq is still quadratic! The reason is that when checking that `f : T` in a context `x := v`, to check that **`let`** `x := v` **`in`** `f` has type `T` (assuming that `x` does not occur in `T`), Coq will substitute `v` for `x` in `T`. So if a proof term has n **`let`** binders (e.g., used for sharing subterms), Coq will perform n substitutions on the type of the proof term, even if none of the **`let`** binders are used. If the number of **`let`** binders is linear in the size of the type, there is quadratic overhead in proof-checking time, even when the proof-term size is linear.

We performed a microbenchmark on a rewriting goal with no binders (because there is an obvious algorithm for sharing subterms in that case) and found that the proof-checking time reached about one second at about 2 000 binders and reached 10 seconds at about 7 000 binders. While these results might seem good enough for Fiat Cryptography, we expect that there are hundreds of thousands of primitive reduction/rewriting steps even when there are only a few hundred binders in the output term, and we would need **`let`** binders for each of them. Furthermore, we expect that getting such an algorithm correct would be quite tricky.

Fixing this quadratic bottleneck would, as far as we can tell, require deep changes in how Coq is implemented; it would either require reworking all of Coq to operate on some efficient representation of delayed substitutions paired with unsubstituted terms, or else it would require changing the typing rules of the type theory itself to remove this substitution from the typing rule for **`let`**. Note that there is a similar issue that crops up for function application and abstraction.

A.5 Inherent advantages of reflection

Finally, even if this quadratic bottleneck were fixed, Aehlig et al. [2008] reported a 10×–100× speed-up over the *simp* tactic in Isabelle, which performs all of the intermediate rewriting steps via the kernel API. Their results suggest that even if all of the superlinear bottlenecks were fixed—no small undertaking—rewriting and partial evaluation via reflection might still be orders of magnitude faster than any proof-term-generating tactic.