A Framework for Building Verified Partial Evaluators

Anonymous Author(s)

Abstract

Partial evaluation is a classic technique for generating lean, customized code from libraries that start with more bells and whistles. It is also an attractive approach to creation of *formally verified* systems, where theorems can be proved about libraries, yielding correctness of all specializations "for free." However, it can be challenging to make library specialization both performant and trustworthy. We present a new approach, prototyped in the Coq proof assistant, which supports specialization at the speed of native-code execution, without adding to the trusted code base. Our extensible engine, which combines the traditional concepts of tailored term reduction and automatic rewriting from hint databases, is also of interest to replace these ingredients in proof assistants' proof checkers and tactic engines, at the same time as it supports extraction to standalone compilers from library parameters to specialized code.

1 Introduction

Mechanized proof is gaining in importance for development of critical software infrastructure. Oft-cited examples include the CompCert verified C compiler [17] and the seL4 verified operating-system microkernel [16]. Here we have very flexible systems that are ready to adapt to varieties of workloads, be they C source programs for CompCert or application binaries for seL4. For a verified operating system, such adaptation takes place at *runtime*, when we launch the application. However, some important bits of software infrastructure commonly do adaptation at *compile time*, such that the fully general infrastructure software is not even installed in a deployed system.

Of course, compilers are a natural example of that pattern, as we would not expect CompCert itself to be installed on an embedded system whose application code was compiled with it. The problem is that writing a compiler is rather labor-intensive, with its crafting of syntax-tree types for source, target, and intermediate languages, its fine-tuning of code for transformation passes that manipulate syntax trees explicitly, and so on. An appealing alternative is *partial evaluation* [15], which relies on reusable compiler facilities to specialize library code to parameters, with no need to write that library code in terms of syntax-tree manipulations. Cutting-edge tools in this tradition even make it possible to use high-level functional languages to generate performance-competitive low-level code, as in Scala's Lightweight Modular Staging [22].

It is natural to try to port this approach to construction of systems with mechanized proofs. On one hand, the typed functional languages in popular proof assistants' logics make excellent hosts for flexible libraries, which can often be specialized through means as simple as partial application of curried functions. Term-reduction systems built into the proof assistants can then generate the lean residual programs. On the other hand, it is surprisingly difficult to realize the last sentence with good performance. The challenge is that we are not just implementing algorithms; we also want a proof to be checked by a small proof checker, and there is tension in designing such a checker, as fancier reduction strategies grow the trusted code base. It would seem like an abandonment of the spirit of proof assistants to bake in a reduction strategy per library, yet effective partial evaluation tends to be rather fine-tuned in this way. Performance tuning matters when generated code is thousands of lines long.

In this paper, we present an approach to verified partial evaluation in proof assistants, which requires no changes to proof checkers. To make the relevance concrete, we use the example of Fiat Cryptography [11], 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 about half of all HTTPS connections from browsers are now initiated using code generated (with proof) by Fiat Cryptography. However, Fiat Cryptography was only used successfully to build C code for the two most widely used curves (P-256 and Curve25519). Their method of partial evaluation timed out trying to compile code for the third most widely used curve (P-384). Additionally, to achieve acceptable reduction performance, the library code had to be written manually in continuation-passing style. We will demonstrate a new Coq library that corrects both weaknesses, while maintaining the generality afforded by allowing rewrite rules to be mixed with partial evaluation.

1.1 A Motivating Example

We are interested in partial-evaluation examples that mix higher-order functions, inductive datatypes, and arithmetic simplification. For instance, consider the following Coq code.

```
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, ls''') := acc_ls''' in
```

Conference'17, July 2017, Washington, DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
    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]
```

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 partial evaluator, we prove the algebraic laws that it should use for simplification, starting with basic arithmetic identities.

```
Lemma zero_plus  : forall n, 0 + n = n.
Lemma plus_zero  : forall n, n + 0 = n.
Lemma times_zero : forall n, n * 0 = 0.
Lemma times_one  : forall 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 use a special apostrophe marker to indicate a quantified variable that may only match with *compile-time constants*. We also use a further marker `ident.eagerly` to ask the reducer 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 rewrite procedure 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 an extra bottom-up pass 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 righthand 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
 | forall a b c d, f a b c d = prefixSums [a;b;c;d]} :=
  ltac:(eexists; Rewrite_rhs_for rewriter; reflexivity).
```

1.2 Concerns of Trusted-Code-Base Size

Crafting a reduction strategy is challenging enough in a standalone tool. A large part of the difficulty in a proof assistant is reducing in a way that leaves a proof trail that can be checked efficiently by a small kernel. Most proof assistants present user-friendly surface tactic languages that generate proof traces in terms of more elementary tactic steps. The trusted proof checker only needs to know about the elementary steps, and there is pressure to be sure that these steps are indeed elementary, not requiring excessive amounts of kernel code. However, hardcoding a new reduction strategy in the kernel can bring dramatic performance improvements. Generating thousands of lines of code with partial evaluation would be intractable if we were outputting sequences of primitive rewrite steps justifying every little term manipulation, so we must take advantage of the time-honored feature of type-theoretic proof assistants that reductions included in the definitional equality need not be requested explicitly.

Which kernel-level reductions *does* Coq support today? Currently, the trusted code base knows about four different kinds of reduction: left-to-right conversion, right-to-left conversion, a virtual machine (VM) written in C based on the OCaml compiler, and a compiler to native code. Furthermore, the first two are parameterized on an arbitrary user-specified ordering of which constants to unfold when, in addition to internal heuristics about what to do when the user has not specified an unfolding order for given constants. Recently, native support for 63-bit integers has been added to the VM and native machines. A recent pull request proposes adding support for native IEEE 754-2008 binary64 floats [21], and support for native arrays is in the works [10].

To summarize, there has been quite a lot of "complexity creep" in the Coq trusted base, to support efficient reduction, and yet realistic partial evaluation has *still* been rather challenging. Even the additional three reduction mechanisms outside Coq's kernel (`cbn`, `simpl`, `cbv`) are not at first glance sufficient for verified partial evaluation.

1.3 Our Solution

Aehlig et al. [1] presented a very relevant solution to a related problem, using *normalization by evaluation (NbE)* [4] 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. We have adapted their approach in a new Coq library embodying **the first partial-evaluation approach to satisfy the following criteria**.

- It integrates with a general-purpose, foundational proof assistant, **without growing the trusted 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 **preserves sharing** of common subterms.
- It also allows **extraction of standalone partial evaluators**.

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 [20]).

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 [18]). 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 blowup. 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)* [8] 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 partial evaluation in a proof assistant, covering all elliptic curves from the published Fiat Cryptography experiments, along with microbenchmarks.

This paper proceeds through explanations of the trust stories behind our approach and earlier ones (section 2), the core structure of our engine (section 3), the additional scaling challenges we faced (section 4), performance experiments (section 5), and related work (section 6) and conclusions. Our implementation is included as an anonymous supplement.

2 Trust, Reduction, and Rewriting

Since much of the narrative behind our design process depends on tradeoffs between performance and trustworthiness, we start by reviewing the general situation in proof assistants.

Across a variety of proof assistants, simplification of functional programs is a workhorse operation. Proof assistants like Coq that are based on type theory typically build in *definitional equality* relations, identifying terms up to reductions like β-reduction and unfolding of named identifiers. What looks like a single "obvious" step in an on-paper equational proof may require many of these reductions, so it is handy to have built-in support for checking a claimed reduction. Figure 1a diagrams how such steps work in a system like Coq, where the system implementation is divided between a trusted *kernel*, for checking *proof terms* in a minimal language, and additional untrusted support, like a *tactic* engine evaluating a language of higher-level proof steps, in the process generating proof terms out of simpler building blocks. It is standard to include a primitive proof step that validates any reduction compatible with the definitional equality, as the latter is decidable. The figure shows a tactic that simplifies a goal using that facility.

In proof goals containing free variables, executing subterms can get stuck before reaching normal forms. However, we can often achieve further simplification by using equational rules that we prove explicitly, rather than just relying on the rules built into the definitional equality and its decidable equivalence checker. Coq's `autorewrite` tactic, as diagrammed in Figure 1b, is a good example: it takes in a database of quantified equalities and applies them repeatedly to rewrite in a goal. It is important that Coq's kernel does not trust the `autorewrite` tactic. Instead, the tactic must output a proof term that, in some sense, is the moral equivalent of a line-by-line equational proof. It can be challenging to keep these proof terms small enough, as naive rewrite-by-rewrite versions repeatedly copy large parts of proof goals, justifying a rewrite like $C[e_1] = C[e_2]$ for some context C given a proof of $e_1 = e_2$, with the full value of C replicated in the proof term for that single rewrite. Overcoming these challenges while retaining decidability of proof checking is tricky, since we may use `autorewrite` with rule sets that do not always lead to terminating reduction. Coq includes more experimental alternatives like `rewrite_strat`, which use bottom-up construction of multi-rewrite proofs, with sharing of common contexts. Still, as section 5 will show, these methods that generate substantial proof terms are at significant performance disadvantages.

Now we summarize how Aehlig et al. [1] provide flexible and fast interleaving of standard λ-calculus reduction and use of proved equalities (the next section will go into more detail). Figure 1c demonstrates a workflow based on *a deep embedding of a core ML-like language.* That is, 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. The basic strategy is, for a particular set of rewrite rules and a particular term to simplify, to *generate a (deeply embedded) ML program that, if it terminates, produces a syntax tree for the simplified term.* Their tactic uses *reification* to created ML versions of rule sets and terms. They also wrote a reduction function in ML and proved it sound once and for all, against the ML operational semantics. Combining that proof with proofs generated by reification, we conclude that an application of the reduction function to the reified rules and term is indeed an ML term that generates correct answers. The tactic then "throws the ML term over the wall," using a general code-generation framework for Isabelle/HOL [14]. 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

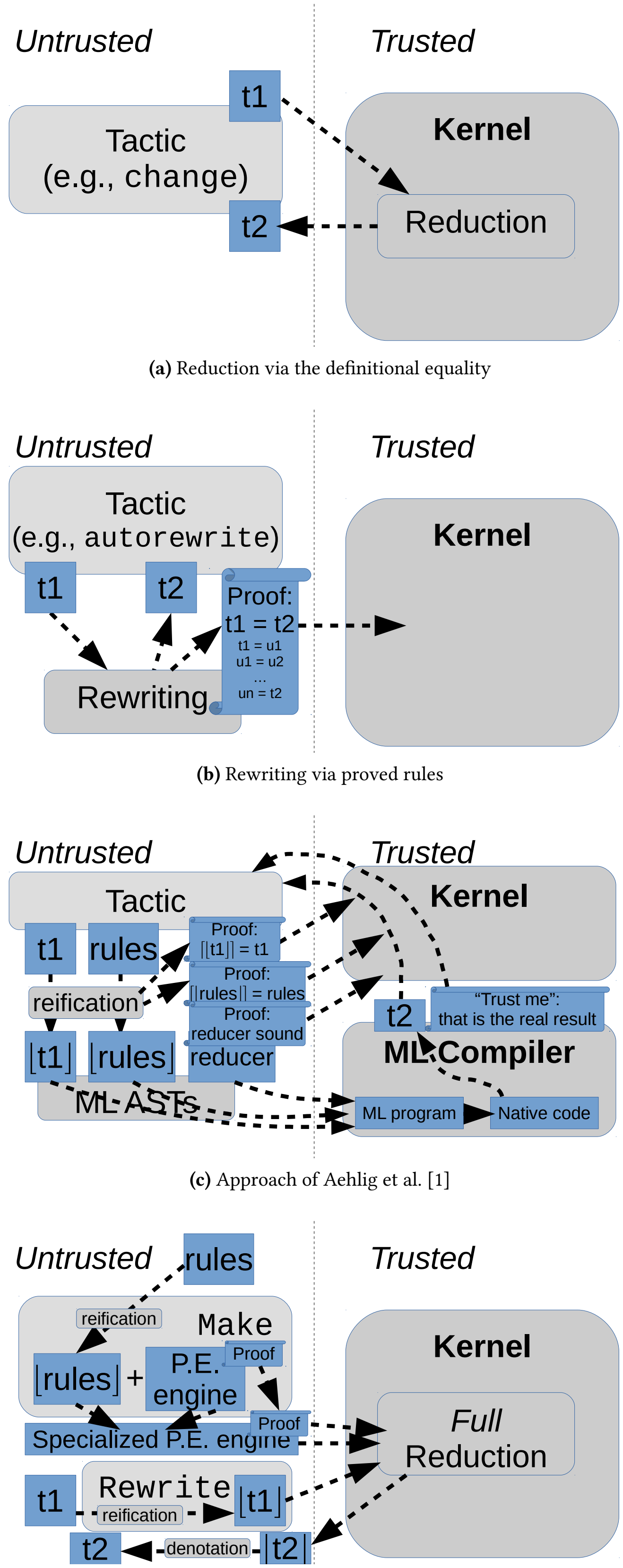


Figure 1. Different approaches to reduction and rewriting

is then passed back over to the tactic, in terms of an axiomatic assertion that the ML semantics really yields that answer.

As Aehlig et al. [1] argue, their use of external compilation and evaluation of ML code adds no real complexity on top of that required by the proof assistant – after all, the proof assistant itself must be compiled and executed somehow. However, the perceived increase of trusted code base is not spurious: it is one thing to trust that the toolchain and execution environment used by the proof assistant and the partial evaluator are well-behaved, and another to rely on two descriptions of ML (one deeply embedded in the proof assistant and another implied by the compiler) to agree on every detail of the semantics. Furthermore, there still is new trusted code to translate from the deeply embedded ML subset into the concrete syntax of the full-scale ML language. The vast majority of proof-assistant developments today rely on no such embeddings with associated mechanized semantics, so need we really add one to a proof-checking kernel to support efficient partial evaluation?

Our answer, diagrammed in Figure 1d, shows a different way. We still reify terms and rules into a deeply embedded language. However, *the reduction engine is implemented directly in the logic*, rather than as a deeply embedded syntax tree of an ML program. As a result, the kernel's own reduction engine is prepared to execute our reduction engine for us – using an operation that would be included in a type-theoretic proof assistant in any case, with no special support for a language deep embedding. We also stage the process for performance reasons. First, the **Make** command creates a rewriter out of a list of rewrite rules, by specializing a generic partial-evaluation engine, which has a generic proof that applies to any set of proved rewrite rules. We perform partial evaluation on the specialized partial evaluator, using Coq's normal reduction mechanisms, under the theory that we can afford to pay performance costs at this stage because we only need to create new rewriters relatively infrequently. Then individual rewritings involve reifying terms, asking the kernel to execute the specialized evaluator on them, and simplifying an application of an interpretation function to the result (this last step must be done using Coq's normal reduction, and it is the bottleneck for outputs with enormous numbers of nested binders as discussed in section 5.1).

2.1 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.
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 we prove soundness and syntactic-well-formedness lemmas 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.
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.

3 The Structure of a Rewriter

We now simultaneously review the approach of Aehlig et al. [1] and introduce some notable differences in our own approach, noting similarities to the reflective rewriter of Malecha and Bengtson [18] where applicable.

First, let us describe the language of terms we support rewriting in. Note that, while we support rewriting in full-scale Coq proofs, where the metalanguage is dependently typed, the object language of our rewriter is nearly simply typed, with limited support for calling polymorphic functions. However, we still support identifiers whose definitions use dependent types, since our reducer does not need to look into definitions.

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

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.

3.1 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. 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 [18], 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 [20], 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 lefthand 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`, 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`.

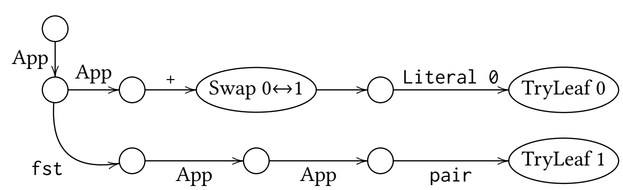
Consider the encoding of two simple example rewrite rules, where we follow Coq's $\mathcal{L}_{tac}$ language in prefacing pattern variables with question marks.

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

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 non-swap node implicitly has a "default" case arrow to `Failure`.

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 meta-compilation.

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.
```

3.2 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) [4] 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

$$?n + 0 \longrightarrow n$$

Aehlig et al. [1] also use NbE, and we begin by reviewing its 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.

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

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 dependent type family `expr`.

Now the core of NbE, shown in Figure 2, 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

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

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

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

$$\begin{aligned} \mathrm{NbE} &: \mathrm{expr}(t) \to \mathrm{expr}(t) \\ \mathrm{NbE}(e) &= \mathrm{reify}(\mathrm{reduce}(e)) \end{aligned}$$

Figure 2. Implementation of normalization by evaluation

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:

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

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. [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 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. [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 [9]. 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, which is itself performance-tuned with some crucial pieces of imperative code. Parametric higher-order abstract syntax (PHOAS) [8] 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
| 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, 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)
     | 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.

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

Even before getting to the correctness theorem, we needed to convince Coq that the function terminates. While for Aehlig et al. [1], 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. 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 that the Coq kernel's built-in reduction 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, while 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 may ask the Coq kernel to simplify **Rewrite**(E) by *full reduction via compilation to native code*, since we carefully designed **Rewrite**(E) and its dependencies to produce closed syntax trees. Finally, where E' is the result of that reduction, we simplify $[\![E']\!]$ with standard reduction, producing a normal-looking Coq term.

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 [11] 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 5.2) 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. 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

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

Our approach is to 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 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. One simple case is supporting *nonlinear* patterns, where a pattern variable appears multiple times. We can encode nonlinearity on top of linear patterns via side conditions.

$$?n_1 + ?m - ?n_2 \longrightarrow m \text{ if } n_1 = n_2$$

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 [18] 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, synthesizing one Boolean expression of our deeply embedded term language, standing 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.

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 partial evaluation. 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 infinite-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 $?n + 0 \longrightarrow n$. When we get to reducing fixed-precision-integer terms, we must be legalistic:

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

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:

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

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.

5 Evaluation

Our implementation, attached to this submission as an anonymized supplement with a roadmap in Appendix D, 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 experiments to isolate performance benefits for rewriting under binders and reducing higher-order structure.

5.1 Microbenchmarks

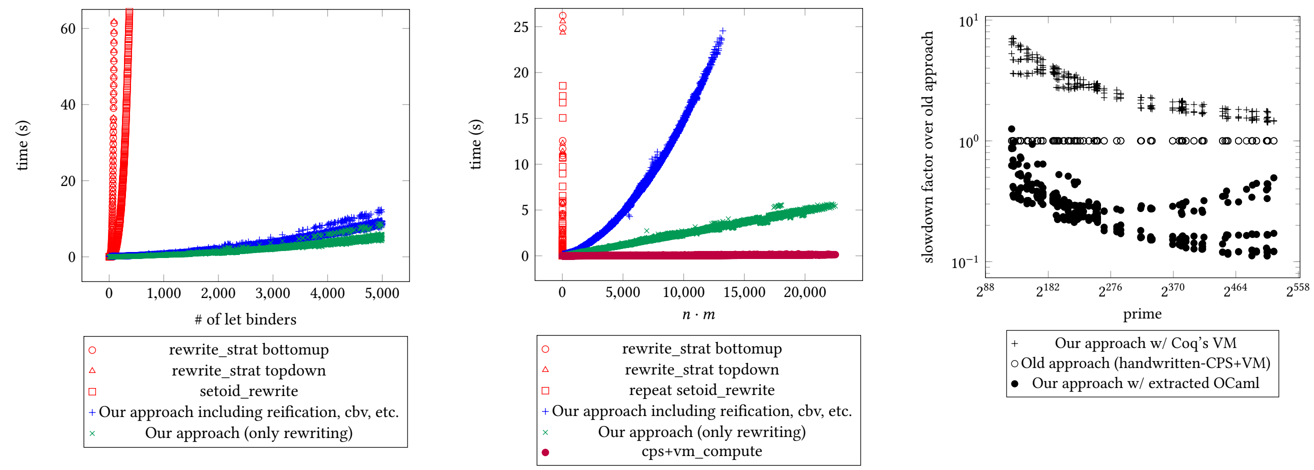
We start with microbenchmarks focusing attention on particular aspects of reduction and rewriting, with Appendix A going into more detail.

5.1.1 Rewriting Under Binders

Consider

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

We want to remove all of the $+\,0$s. We can start from this expression directly, in which case reification alone takes as



(a) Nested binders **(b)** Binders and recursive functions **(c)** Fiat Cryptography

Figure 3. Timing of different partial-evaluation implementations

much time as `setoid_rewrite`. As the reification method was not especially optimized, and there exist fast reification methods [13], we instead start from a call to a recursive function that generates such a sequence of **`let`** bindings.

Figure 3a shows the results. The comparison points are Coq's `setoid_rewrite` and `rewrite_strat`. The former performs one rewrite at a time, taking minimal advantage of commonalities across them and thus generating quite large, redundant proof terms. The latter 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. 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.

As detailed in subsection A.2, we ran a variant of this experiment with inlining of **`let`**s, forcing terms to grow quite large. Specifically, we generate n nested **`let`**s, each repeatedly adding a designated free variable into a sum, m times. Holding m fixed at a small value and letting n scale, we continue dominating the methods described above, though Coq's `rewrite!` tactic (to rewrite with one lemma many times) does better for $m < 2$. Holding n fixed and letting m scale, all other approaches quickly spike upward to timeouts, while ours holds steady even for $m = 1000$.

5.1.2 Binders and Recursive Functions

The next experiment uses the following example.

$$\mathrm{map\_dbl}(\ell) = \begin{cases} [] & \text{if } \ell = [] \\ \texttt{let } y := h + h \texttt{ in } y :: \mathrm{map\_dbl}(t) & \text{if } \ell = h :: t \end{cases}$$

$$\mathrm{make}(n, m, v) = \begin{cases} \underbrace{[v, \ldots, v]}_{n} & \text{if } m = 0 \\ \mathrm{map\_dbl}(\mathrm{make}(n, m-1, v)) & \text{if } m > 0 \end{cases}$$

$$\mathrm{example}_{n,m} = \forall v,\ \mathrm{make}(n, m, v) = []$$

Note that the `let ··· in ···` binding blocks further reduction of map_dbl, which we iterate m times, and so we need to take care to preserve sharing when reducing here.

Figure 3b 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 CPSed 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. See subsection A.3 for more on this microbenchmark and subsection A.4 for an even more extreme example of full reduction with a Sieve of Eratosthenes as in the experiments of Aehlig et al. [1] (ours 10s, VM 0.3s).

5.2 Macrobenchmark: Fiat Cryptography

Finally, we consider an experiment (described in more detail in Appendix B) replicating the generation of performance-competitive finite-field-arithmetic code for all popular elliptic curves by Erbsen et al. [11]. 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 methods for Fiat Cryptography, as the prime modulus of arithmetic scales up. Times are normalized to the performance of the original method, which relied entirely on standard Coq reduction. Actually, in the course of running this experiment, we found a way to improve the old approach for a fairer comparison. It had relied on Coq's configurable `cbv` tactic to perform reduction with selected rules of the definitional equality, which the Fiat Cryptography developers had applied to blacklist identifiers that should be left for compile-time execution. By instead hiding those identifiers behind opaque module-signature ascription, we were able to run Coq's more-optimized virtual-machine-based reducer.

As the figure shows, our approach running partial evaluation inside Coq's kernel begins with about a 10× performance disadvantage vs. the original method. With log scale on both axes, we see that this disadvantage narrows to become nearly negligible for the largest primes, of around 500 bits. (We used the same set of prime moduli as in the experiments run by Erbsen et al. [11], which were chosen based on searching the archives of an elliptic-curves mailing list for all prime numbers.) It makes sense that execution inside Coq leaves our new approach at a disadvantage, as we are essentially running an interpreter (our normalizer) within an interpreter (Coq's kernel), while the old approach ran just the latter directly. 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 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. By the time we reach middle-of-the-pack prime size around 300 bits, the extracted version is running about 10× as quickly as the baseline.

6 Related Work

We have already discussed the work of Aehlig et al. [1], 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 $\mathcal{R}_{tac}$ [18], which implements an experimental reflective version of `rewrite_strat` supporting arbitrary setoid relations, unification variables, and arbitrary semi-decidable side conditions solvable by other reflective tactics, using de Bruijn indexing to manage binders. We were unfortunately unable to get the rewriter to work with Coq 8.10 and were also not able to determine from the paper how to repurpose the rewriter to handle our benchmarks.

Our implementation builds on fast full reduction in Coq's kernel, via a virtual machine [12] or compilation to native code [5]. 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 suitable to bootstrap on top of reduction support in a kernel providing simpler reduction.

A variety of forms of pragmatic partial evaluation have been demonstrated, with Lightweight Modular Staging [22] 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 [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, justifying investment in verified partial evaluators.

7 Future Work

There are 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 proving equalities on 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

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

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

[3] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Proc. ITP*.

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

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

[6] Barry Bond, Chris Hawblitzel, Manos Kapritsos, Rustan Leino, Jay Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proc. USENIX Security*. http://www.cs.cornell.edu/~laurejt/papers/vale-2017.pdf

[7] Samuel Boutin. 1997. Using reflection to build efficient and certified decision procedures. In *Proc. TACS*.

[8] 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*. http://adam.chlipala.net/papers/PhoasICFP08/

[9] 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.

[10] Maxime Dénès. 2013. Towards primitive data types for COQ. In *The Coq Workshop 2013* (2013-04-06). https://coq.inria.fr/files/coq5_submission_2.pdf

[11] 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*. http://adam.chlipala.net/papers/FiatCryptoSP19/

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

[13] 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/

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

[15] N.D. Jones, C.K. Gomard, and P. Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. SOSP*. ACM, 207–220.

[17] 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

[18] 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

[19] Gregory Michael Malecha. 2014. *Extensible Proof Engineering in Intensional Type Theory.* Ph.D. Dissertation. Harvard University. http://gmalecha.github.io/publication/2015/02/01/extensible-proof-engineering-in-intensional-type-theory.html

[20] 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

[21] Erik Martin-Dorel. 2018. Implementing primitive floats (binary64 floating-point numbers) - Issue #8276 - coq/coq. https://github.com/coq/coq/issues/8276

[22] 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

A Additional Information on Microbenchmarks

We performed all benchmarks on a 3.5 GHz Core i7 running Linux and Coq 8.10.0. We name the subsections here with the names that show up in the code supplement.

A.1 UnderLetsPlus0

We provide more detail on the "nested binders" microbenchmark of subsubsection 5.1.1 displayed in Figure 3a.

Recall that we are removing all of the + 0s from

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

The code used to define this microbenchmark is

```
Definition make_lets_def (n:nat) (v acc : Z) :=
 @nat_rect
   (fun _ => Z * Z -> Z)
   (fun '(v, acc) => acc + acc + v)
   (fun _ rec '(v, acc) =>
     dlet acc := acc + acc + v in rec (v, acc))
   n
   (v, acc).
```

We note some details of the rewriting framework that were glossed over in the main body of the paper, which are useful for using the code: Although the rewriting framework does not support dependently typed constants, we can automatically preprocess uses of eliminators like `nat_rect` and `list_rect` into non-dependent 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, we must internally 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 the motive. Even though we will eventually plug in 0 for v, we jump through some extra hoops to ensure that our rewriter cannot cheat by rewriting away the $+0$ before reducing the recursion on n.

We can reduce this expression in three ways.

A.1.1 Our Rewriter

One lemma is required for rewriting with our rewriter:

```
Lemma Z.add_0_r : forall z, z + 0 = z.
```

Creating the rewriter takes about 12 seconds on the machine we used for running the performance experiments:

```
Make myrew := Rewriter For
 (Z.add_0_r, eval_rect nat, eval_rect prod).
```

Recall from subsection 1.1 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.

A.1.2 `setoid_rewrite` and `rewrite_strat`

To give as many advantages as we can to the preexisting work on rewriting, we pre-reduce the recursion on `nat`s using `cbv` before performing `setoid_rewrite`. (Note that `setoid_rewrite` cannot itself perform reduction without generating large proof terms, and `rewrite_strat` is not currently capable of sequencing reduction with rewriting internally due to bugs such as #10923.) Rewriting itself is easy; we may use any of `repeat setoid_rewrite Z.add_0_r`, `rewrite_strat topdown Z.add_0_r`, or `rewrite_strat bottomup Z.add_0_r`.

A.2 Plus0Tree

This is a version of subsection A.1 without any let binders, discussed in subsubsection 5.1.1 but not displayed in Figure 3.

We use two definitions for this microbenchmark:

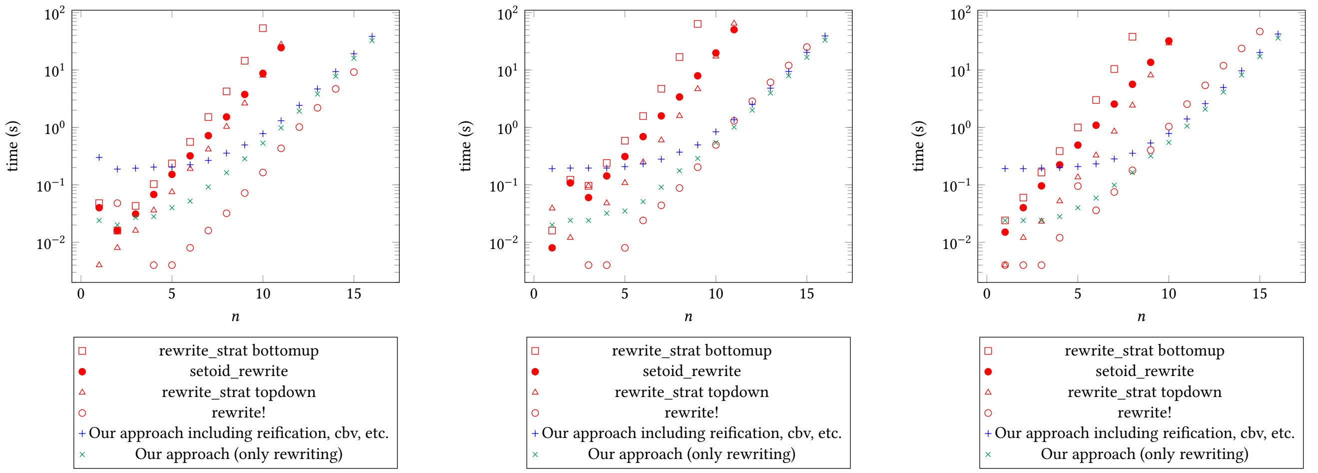
```
Definition iter (m : nat) (acc v : Z) :=
  @nat_rect
    (fun _ => Z -> Z)
    (fun acc => acc)
    (fun _ rec acc => rec (acc + v))
    m
    acc.

Definition make_tree (n m : nat) (v acc : Z) :=
 Eval cbv [iter] in
  @nat_rect
    (fun _ => Z * Z -> Z)
    (fun '(v, acc) => iter m (acc + acc) v)
    (fun _ rec '(v, acc) =>
      iter m (rec (v, acc) + rec (v, acc)) v)
    n
    (v, acc).
```

We can see from the graphs in Figure 4 and Figure 5 that (a) we incur constant overhead over most of the other methods which dominates on small examples; (b) when the term is quite large and there are few opportunities for rewriting relative to the term-size (i.e., $m \leq 2$), we are worse than `rewrite !Z.add_0_r`, but still better than the other methods; and (c) when there are many opportunities for rewriting relative to the term-size ($m > 2$), we thoroughly dominate the other methods.

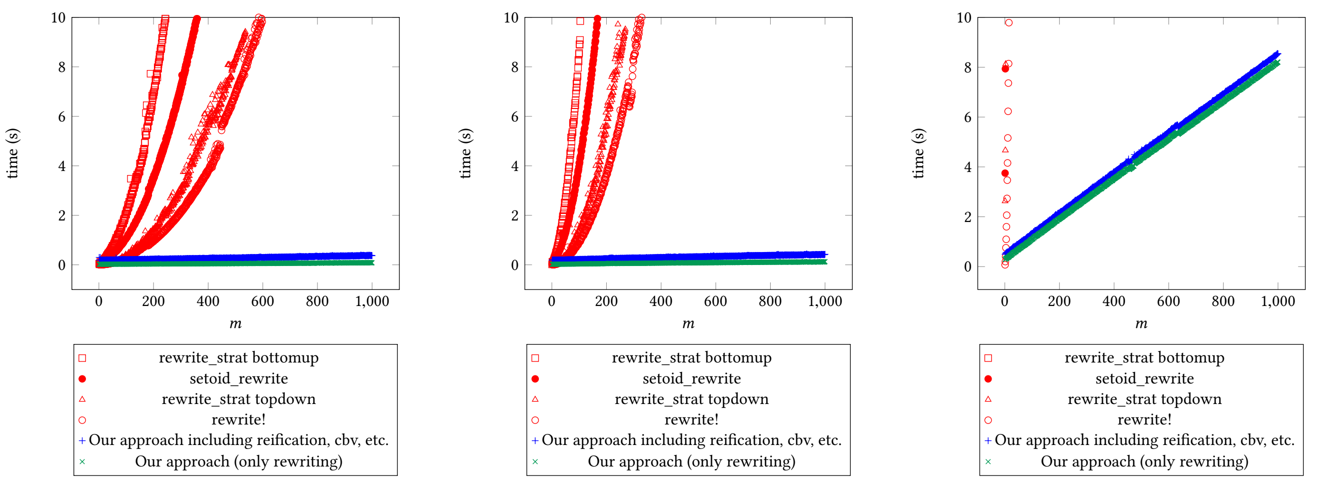
A.3 LiftLetsMap

We now discuss in more detail the "binders and recursive functions" example from subsubsection 5.1.2.



(a) Timing of Plus0Tree ($m = 1$) **(b)** Timing of Plus0Tree ($m = 2$) **(c)** Timing of Plus0Tree ($m = 3$)

Figure 4. Timing of different partial-evaluation implementations for Plus0Tree for fixed m. Note that we have a logarithmic time scale, because term size is proportional to 2^n.



(a) Timing of Plus0Tree ($n = 1$) **(b)** Timing of Plus0Tree ($n = 2$) **(c)** Timing of Plus0Tree ($n = 9$)

Figure 5. Timing of different partial-evaluation implementations for Plus0Tree for fixed n (1, 2, and then we jump to 9)

The expression we want to get out at the end looks like:

$$
\begin{array}{l}
\texttt{let } v_{1,1} := v + v \texttt{ in} \\
\vdots \\
\texttt{let } v_{1,n} := v + v \texttt{ in} \\
\texttt{let } v_{2,1} := v_{1,1} + v_{1,1} \texttt{ in} \\
\vdots \\
\texttt{let } v_{2,n} := v_{1,n} + v_{1,n} \texttt{ in} \\
\vdots \\
[v_{m,1}, \ldots, v_{m,n}]
\end{array}
$$

Recall that we make this example with the code

```
Definition map_double (ls : list Z) :=
  list_rect
    _
    []
    (λ x xs rec, let y := x + x in y :: rec)
    ls.

Definition make (n : nat) (m : nat) (v : Z) :=
  nat_rect
    _
    (List.repeat v n)
    (λ _ rec, map_double rec)
    m.
```

We can perform this rewriting in four ways; see Figure 3b.

Note that `rewrite_strat` grows quite quickly, hitting a minute when the total number of rewrites ($n \cdot m$) is in the mid-40s. Our method performs much better, but the fact that we have to perform cbv at the end costs us; 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. This is due to the unfortunate fact that reduction in Coq is quadratic in the number of nested binders present; see Coq bug #11151. Finally, and unsurprisingly, `vm_compute` outperforms us.

A.3.1 Our Rewriter

One lemma is required for rewriting with our rewriter:

```
Lemma eval_repeat A x n :
  @List.repeat A x ('n)
  = ident.eagerly nat_rect _
      []
      (λ k repeat_k, x :: repeat_k)
      ('n).
```

Recall that the apostrophe marker (') is explained in subsection 1.1. Recall again from subsection 1.1 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 construct our rewriter with

```
Make myrew := Rewriter For
  (eval_repeat, eval_rect list, eval_rect nat)
  (with extra idents (Z.add)).
```

On the machine we used for running all our performance experiments, this command takes about 13 seconds to run. Note that all identifiers which appear in any goal to be rewritten must either appear in the type of one of the rewrite rules or in the tuple passed to `with extra idents`.

Rewriting is relatively simple, now. Simply invoke the tactic **`Rewrite_for`** `myrew`. We support rewriting on only the left-hand-side and on only the right-hand-side using either the tactic **`Rewrite_lhs_for`** `myrew` or else the tactic **`Rewrite_rhs_for`** `myrew`, respectively.

A.3.2 `rewrite_strat`

To reduce adequately using `rewrite_strat`, we need the following two lemmas:

```
Lemma lift_let_list_rect T A P N C (v : A) fls
: @list_rect T P N C (Let_In v fls)
  = Let_In v (fun v => @list_rect T P N C (fls v)).
Lemma lift_let_cons T A x (v : A) f
: @cons T x (Let_In v f)
  = Let_In v (fun v => @cons T x (f v)).
```

Note that **`Let_In`** is the constant we use for writing `let ··· in ···` expressions that do not reduce under ζ. 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 may cause problems when converting the interpretation of the reified term with the pre-reified term, as Coq's conversion does not allow fine-tuning of when to inline or unfold **`let`**s.

To rewrite, we start with `cbv [example make map_dbl]` to expose the underlying term to rewriting. One would hope that one could just add these two hints to a database `db` and then write `rewrite_strat (repeat (eval cbn [list_rect]; try bottomup hints db))`, but unfortunately this does not work due to a number of bugs in Coq: #10934, #10923, #4175, #10955, and the potential to hit #10972. Instead, we must put the two lemmas in separate databases, and then write `repeat (cbn [list_rect]; (rewrite_strat (try repeat bottomup hints db1)); (rewrite_strat (try repeat bottomup hints db2)))`. Note that the rewriting with `lift_let_cons` can be done either top-down or bottom-up, but `rewrite_strat` breaks if the rewriting with `lift_let_list_rect` is done top-down.

A.3.3 CPS and the VM

If we want to use Coq's built-in VM reduction without our rewriter, to achieve the prior state-of-the-art performance, we can do so on this example, because it only involves partial reduction and not equational rewriting. However, we must (a) module-opacify the constants which are not to be unfolded, and (b) rewrite all of our code in CPS.

Then we are looking at

$$\mathrm{map\_dbl\_cps}(\ell, k) = \begin{cases} k([]) & \text{if } \ell = [] \\ \texttt{let } y := h +_{\mathrm{ax}} h \texttt{ in} & \text{if } \ell = h :: t \\ \mathrm{map\_dbl\_cps}(t, & \\ \qquad (\lambda ys, k(y :: ys))) & \end{cases}$$

$$\mathrm{make\_cps}(n, m, v, k) = \begin{cases} k([\underbrace{v, \ldots, v}_{n}]) & \text{if } m = 0 \\ \mathrm{make\_cps}(n, m-1, v, & \text{if } m > 0 \\ \quad (\lambda \ell, \mathrm{map\_dbl\_cps}(\ell, k)) & \end{cases}$$

$$\mathrm{example\_cps}_{n,m} = \forall v,\ \mathrm{make\_cps}(n, m, v, \lambda x.\, x) = []$$

Then we can just run `vm_compute`. Note that this strategy, while quite fast, results in a stack overflow when $n \cdot m$ is larger than approximately $2.5 \cdot 10^4$. This is unsurprising, as we are generating quite large terms. Our framework can handle terms of this size but stack-overflows on only slightly larger terms.

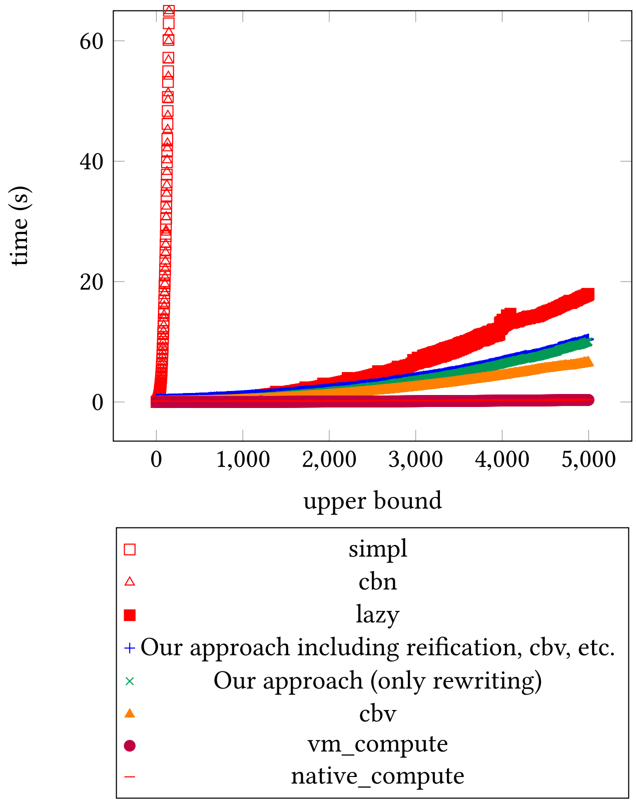


Figure 6. Timing of different full-evaluation implementations for SieveOfEratosthenes

A.3.4 Takeaway

From this example, we conclude that `rewrite_strat` is unsuitable for computations involving large terms with many binders, especially in cases where reduction and rewriting need to be interwoven, and that the many bugs in `rewrite_strat` result in confusing gymnastics required for success. The prior state of the art—writing code in CPS—suitably tweaked by using module pacity to allow `vm_compute`, remains the best performer here, though the cost of rewriting everything is CPS may be prohibitive. Our method soundly beats `rewrite_strat`. We are additionally bottlenecked on cbv, which is used to unfold the goal post-rewriting and costs about a minute on the largest of terms; see Coq bug #11151 for a discussion on what is wrong with Coq's reduction here.

A.4 SieveOfEratosthenes

To benchmark how much overhead we add when we are reducing fully, we compute the Sieve of Eratosthenes, taking inspiration on benchmark choice from Aehlig et al. [1]. 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.

We define the sieve using `PositiveMap.t` and `list Z`:

```
Definition sieve' (fuel : nat) (max : Z) :=
 List.rev
  (fst
   (@nat_rect
    (λ _, list Z (* primes *) *
     PositiveSet.t (* composites *) *
     positive (* np (next_prime) *) ->
     list Z (* primes *) *
     PositiveSet.t (* composites *))
    (λ '(primes, composites, next_prime),
     (primes, composites))
    (λ _ rec '(primes, composites, np),
      rec
       (if (PositiveSet.mem np composites ||
            (Z.pos np >? max))%bool%Z
        then
         (primes, composites, Pos.succ np)
        else
         (Z.pos np :: primes,
          List.fold_right
           PositiveSet.add
           composites
           (List.map
             (λ n, Pos.mul (Pos.of_nat (S n)) np)
             (List.seq 0 (Z.to_nat(max/Z.pos np)))),
          Pos.succ np)))
    fuel
    (nil, PositiveSet.empty, 2%positive))).

Definition sieve (n : Z)
  := Eval cbv [sieve'] in sieve' (Z.to_nat n) n.
```

We need four lemmas and an additional instance to create the rewriter:

```
Lemma eval_fold_right A B f x ls :
@List.fold_right A B f x ls
= ident.eagerly list_rect _ _
    x
    (λ l ls fold_right_ls, f l fold_right_ls)
    ls.

Lemma eval_app A xs ys :
xs ++ ys
= ident.eagerly list_rect A _
    ys
    (λ x xs app_xs_ys, x :: app_xs_ys)
    xs.

Lemma eval_map A B f ls :
@List.map A B f ls
= ident.eagerly list_rect _ _
    []
    (λ l ls map_ls, f l :: map_ls)
    ls.

Lemma eval_rev A xs :
@List.rev A xs
= (@list_rect _ (fun _ => _))
    []
```

```
    (λ x xs rev_xs, rev_xs ++ [x])%list
    xs.

Scheme Equality for PositiveSet.tree.

Definition PositiveSet_t_beq
   : PositiveSet.t -> PositiveSet.t -> bool
  := tree_beq.

Global Instance PositiveSet_reflect_eqb
 : reflect_rel (@eq PositiveSet.t) PositiveSet_t_beq
 := reflect_of_brel
      internal_tree_dec_bl internal_tree_dec_lb.
```

We then create the rewriter with

```
Make myrew := Rewriter For
  (eval_rect nat, eval_rect prod, eval_fold_right,
   eval_map, do_again eval_rev, eval_rect bool,
   @fst_pair, eval_rect list, eval_app)
   (with extra idents (Z.eqb, orb, Z.gtb,
    PositiveSet.elements, @fst, @snd,
    PositiveSet.mem, Pos.succ, PositiveSet.add,
    List.fold_right, List.map, List.seq, Pos.mul,
    S, Pos.of_nat, Z.to_nat, Z.div, Z.pos, O,
    PositiveSet.empty))
  (with delta).
```

To get cbn and simpl to unfold our term fully, we emit

```
Global Arguments Pos.to_nat !_ / .
```

B Additional Information on Fiat Cryptography Benchmarks

It may also be useful to see performance results with absolute times, rather than normalized execution ratios vs. the original Fiat Cryptography implementation. Furthermore, the benchmarks fit into four quite different groupings: elements of the cross product of two algorithms (unsaturated Solinas and word-by-word Montgomery) and bitwidths of target architectures (32-bit or 64-bit). Here we provide absolute-time graphs by grouping in Figure 7.

C Experience vs. Lean and `setoid_rewrite`

Although all of our toy examples work with `setoid_rewrite` or `rewrite_strat` (until the terms get too big), even the smallest of examples in Fiat Cryptography fell over using these tactics. When attempting to use `rewrite_strat` for partial evaluation and rewriting on unsaturated Solinas with 1 limb on small primes (such as 29), we were able to get `rewrite_strat` to finish after about 90 seconds. The bugs in `rewrite_strat` made finding the right magic invocation quite painful, nonetheless; the invocation we settled on involved *sixteen* consecutive calls to `rewrite_strat` with varying arguments and strategies. Trying to synthesize code for two limbs on slightly larger primes (such as 113, which needs two limbs on a 64-bit machine) took about three hours. The widely used primes tend to have around five to ten limbs; we leave extrapolating this slowdown to the reader.

We have attached this experiment using `rewrite_strat` as `fiat_crypto_via_rewrite_strat.v`, which is meant to be run in emacs/PG from inside the `fiat-crypto` directory, or in coqc by setting `COQPATH` to the value emitted by `make printenv` in `fiat-crypto` and then invoking the command `coqc -q -R /path/to/fiat-crypto/src Crypto /path/to/fiat_crypto_via_rewrite_strat.v`. To test with the two-limb prime 113, change `of_string "2^5-3" 8` in the definition of `p` to `of_string "2^7-15" 64`.

We also tried Lean, in the hopes that rewriting in Lean, specifically optimized for performance, would be up to the challenge. Although Lean performed about 30% better than Coq on the 1-limb example, taking a bit under a minute, it did not complete on the two-limb example even after four hours (after which we stopped trying), and a five-limb example was still going after 40 hours.

We have attached our experiments with running `rewrite` in Lean on the Fiat Cryptography code as a supplement as well. We used Lean version 3.4.2, commit cbd2b6686ddb, Release. Run `make` in `fiat-crypto-lean` to run the one-limb example; change `open ex` to `open ex2` to try the two-limb example, or to `open ex5` to try the five-limb example.

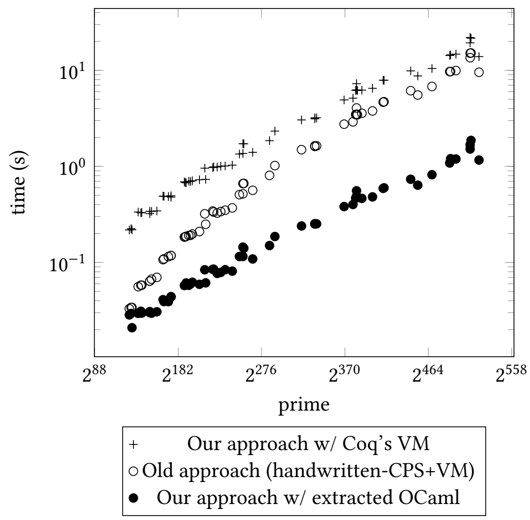
D Reading the Code Supplement

We have attached both the code for implementing the rewriter, as well as a copy of Fiat Cryptography adapted to use the rewriting framework. Both code supplements build with Coq 8.9 and Coq 8.10, and they require that whichever OCaml was used to build Coq be installed on the system to permit building plugins. (If Coq was installed via opam, then the correct version of OCaml will automatically be available.) Both code bases can be built by running `make` in the top-level directory.

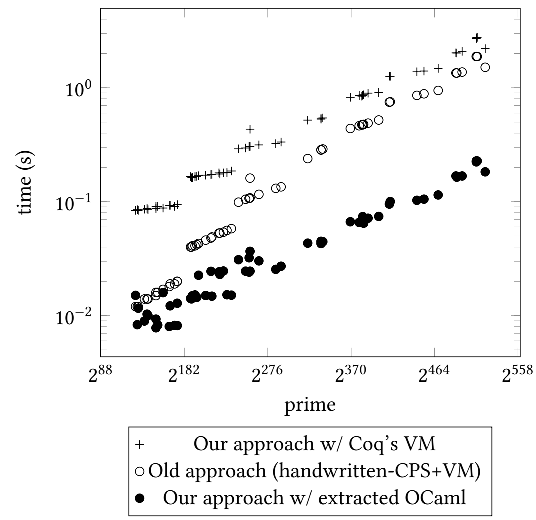
The performance data for both repositories are included at the top level as `.txt` and `.csv` files.

The performance data for the microbenchmarks can be rebuilt using `make perf-SuperFast perf-Fast perf-Medium` followed by `make perf-csv` to get the `.txt` and `.csv` files. The microbenchmarks should run in about 24 hours when run with `-j5` on a 3.5 GHz machine. There also exist targets `perf-Slow` and `perf-VerySlow`, but these take significantly longer.

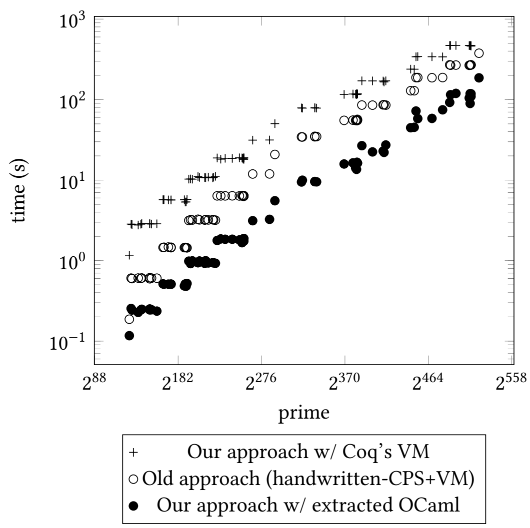
The performance data for the macrobenchmark can be rebuilt from the Fiat Cryptography copy included by running `make perf -k`. We ran this with `PERF_MAX_TIME=3600` to allow each benchmark to run for up to an hour; the default is 10 minutes per benchmark. Expect the benchmarks to take over a week of time with an hour timeout and five cores. Some tests are expected to fail, making `-k` a necessary flag. Again, the `perf-csv` target will aggregate the logs and turn them into `.txt` and `.csv` files.



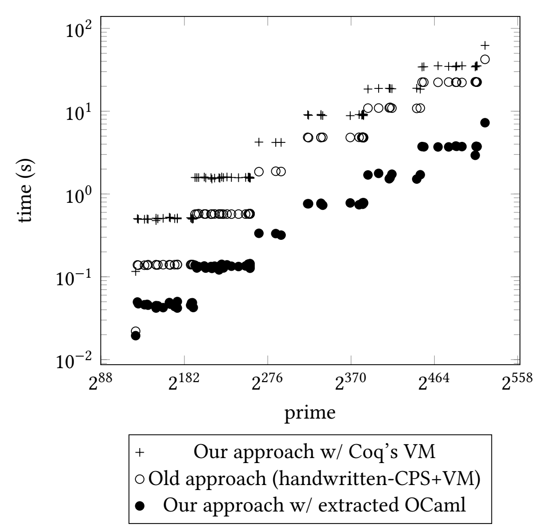
(a) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only unsaturated Solinas x32)



(b) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only unsaturated Solinas x64)



(c) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only word-by-word Montgomery x32)



(d) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only word-by-word Montgomery x64)

Figure 7. Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows

The entry point for the rewriter is the Coq source file `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v`.

The rewrite rules used in Fiat Cryptography are defined in `fiat-crypto/src/Rewriter/Rules.v` and proven in `fiat-crypto/src/Rewriter/RulesProofs.v`. Note that the Fiat Cryptography copy uses `COQPATH` for dependency management, and `.dir-locals.el` to set `COQPATH` in emacs/PG; you must accept the setting when opening a file in the directory for interactive compilation to work. Thus interactive editing either requires ProofGeneral or manual setting of `COQPATH`. The correct value of `COQPATH` can be found by running `make printenv`.

We will now go through this paper and describe where to find each reference in the code base.

D.1 Code from section 1, Introduction

D.1.1 Code from subsection 1.1, A Motivating Example

The `prefixSums` example appears in the Coq source file `rewriter/src/Rewriter/Rewriter/Examples/PrefixSums.v`.

Note that we use `dlet` rather than **`let`** in binding `acc'` so that we can preserve the **`let`** binder even under ι reduction, which much of Coq's infrastructure performs eagerly. Because we attempt to isolate the dependency on the axiom of functional extensionality as much as possible, we also in practice require `Proper` instances for each higher-order identifier saying that each constant respects function extensionality. We hope to remove the dependency on function extensionality altogether in the future. Although we glossed over this detail in the body of this paper, we also prove

```
Global Instance: forall A B,
 Proper ((eq ==> eq ==> eq) ==> eq ==> eq ==> eq)
        (@fold_left A B).
```

The **`Make`** command is exposed in the file `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v` and defined in the OCaml file `rewriter/src/Rewriter/Util/plugins/rewriter_build_plugin.mlg`. Note that one must run `make` to create this latter file; it is copied over from a version-specific file at the beginning of the build.

The `do_again`, `eval_rect`, and `ident.eagerly` constants are defined at the bottom of module `RewriteRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`.

D.1.2 Code from subsection 1.2, Concerns of Trusted-Code-Base Size

There is no code mentioned in this section.

D.1.3 Code from subsection 1.3, Our Solution

We claimed that our solution meets five criteria. We briefly justify each criterion with a sentence or a pointer to code:

- We claimed that we **did not grow the trusted base** (excepting the axiom of functional extensionality). In any example file (of which a couple can be found in `rewriter/src/Rewriter/Rewriter/Examples/`), the **`Make`** command creates a rewriter package. Running `Print Assumptions` on this new constant (often named `rewriter` or `myrew`) should demonstrate a lack of axioms other than functional extensionality. `Print Assumptions` may also be run on the proof that results from using the rewriter.
- We claimed **fast** partial evaluation with reasonable memory use; we assume that the performance graphs stand on their own to support this claim. Note that memory usage can be observed by making the benchmarks while passing `TIMED=1` to `make`.
- We claimed to allow reduction that **mixes** *rules of the definitional equality* with *equalities proven explicitly as theorems*; the "rules of the definitional equality" are, for example, β reduction, and we assert that it should be self-evident that our rewriter supports this.
- We claimed common-subterm **sharing preservation**. This is implemented by supporting the use of the `dlet` notation which is defined in `rewriter/src/Rewriter/Util/LetIn.v` via the `Let_In` constant. We will come back to the infrastructure that supports this.
- We claimed **extraction of standalone partial evaluators**. The extraction is performed in the Coq source file `perf_unsaturated_solinas.v`, in the source file `perf_word_by_word_montgomery.v`, and in the source files `saturated_solinas.v`, `unsaturated_solinas.v`, and `word_by_word_montgomery.v`, all in the directory `fiat-crypto/src/ExtractionOCaml/`. The OCaml code can be extracted and built using the target `make standalone-ocaml` (or `make perf-standalone` for the `perf_` binaries). There may be some issues with building these binaries on Windows as some versions of `ocamlopt` on Windows seem not to support outputting binaries without the `.exe` extension.

The P-384 curve is mentioned. This is the curve with prime modulus $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, and the benchmarks for this curve can be found in the files matching the glob `fiat-crypto/src/Rewriter/PerfTesting/Specific/generated/p2384m2128m296p232m1__*__word_by_word_montgomery_*`. While the `.log` files are included in the tarball, the `.v` and `.sh` files are automatically generated in the course of running `make perf -k`.

We mention integration with abstract interpretation; the abstract-interpretation pass is implemented in `fiat-crypto/src/AbstractInterpretation/`.

D.2 Code from section 2, Trust, Reduction, and Rewriting

The individual rewritings mentioned are implemented via the `Rewrite_*` tactics exported at the top of `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v`. These tactics bottom out in tactics defined at the bottom of `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

D.2.1 Code from subsection 2.1, Our Approach in Nine Steps

We match the nine steps with functions from the source code:

1. The given lemma statements are scraped for which named functions and types the rewriter package will support. This is performed by `rewriter_scrape_data` in the file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml` which invokes the tactic named `make_scrape_data` in a submodule in `rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v` on a goal headed by the constant we provide under the name `Pre.ScrapedData.t_with_args` in `rewriter/src/Rewriter/Language/PreCommon.v`.
2. Inductive types enumerating all available primitive types and functions are emitted. This step is performed by `rewriter_emit_inductives` in file `rewriter/src/`

Rewriter/Util/plugins/rewriter_build.ml invoking tactics, like make_base_elim in rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v, on goals headed by constants from rewriter/src/Rewriter/Language/IdentifiersBasicLibrary.v, including base_elim_with_args for example, to turn scraped data into eliminators for the inductives. The actual emitting of inductives is performed by code in the file rewriter/src/Rewriter/Util/plugins/inductive_from_elim.ml.

3. Tactics generate all of the necessary definitions and prove all of the necessary lemmas for dealing with this particular set of inductive codes. This step is performed by make_rewriter_of_scraped_and_ind in the source file rewriter/src/Rewriter/Util/plugins/rewriter_build.ml which invokes make_rewriter_all defined in the file rewriter/src/Rewriter/Rewriter/AllTactics.v on a goal headed by the provided constant VerifiedRewriter_with_ind_args defined in rewriter/src/Rewriter/Rewriter/ProofsCommon.v. The definitions emitted can be found by looking at the tactic Build_Rewriter in rewriter/src/Rewriter/Rewriter/AllTactics.v, the tactics build_package in the source file rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v and also in the Coq source file found in rewriter/src/Rewriter/Language/IdentifiersGenerate.v (there is a different tactic named build_package in each of these files), and the tactic prove_package_proofs_via which can be found in the Coq source file rewriter/src/Rewriter/Language/IdentifiersGenerateProofs.v.
4. The statements of rewrite rules are reified, and we prove soundness and syntactic-well-formedness lemmas about each of them. This step is performed as part of the previous step, when the tactic make_rewriter_all transitively calls Build_Rewriter from rewriter/src/Rewriter/Rewriter/AllTactics.v. Reification is handled by the tactic Build_RewriterT in rewriter/src/Rewriter/Rewriter/Reify.v, while soundness and syntactic-well-formedness are handled by the tactics prove_interp_good and prove_good respectively, both in the source file rewriter/src/Rewriter/Rewriter/ProofsCommonTactics.v.
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. This step is also performed by make_rewriter_of_scraped_and_ind in the source file rewriter/src/Rewriter/Util/plugins/rewriter_build.ml.

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

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). Note that it is not actually an equality between two functions but rather an equiv between two functions, where equiv is a custom relation we define indexed over type codes that is equality up to function extensionality. This step is performed by the tactic generalize_hyps_for_rewriting in rewriter/src/Rewriter/Rewriter/AllTactics.v.
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. This step is performed by the tactic do_reify_rhs_with in rewriter/src/Rewriter/Rewriter/AllTactics.v.
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. This step is performed by the tactic do_rewrite_with in rewriter/src/Rewriter/Rewriter/AllTactics.v.
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. This step is performed by the tactic do_final_cbv in rewriter/src/Rewriter/Rewriter/AllTactics.v.

These steps are put together in the tactic Rewrite_for_gen in rewriter/src/Rewriter/Rewriter/AllTactics.v.

D.2.2 Our Approach in More Than Nine Steps

As the nine steps of subsection 2.1 do not exactly match the code, we describe here a more accurate version of what is going on. For ease of readability, we do not clutter this description with references to the code supplement, instead allowing the reader to match up the steps here with the more coarse-grained ones in subsection 2.1 or subsubsection D.2.1.

In order to allow easy invocation of our rewriter, a great deal of code (about 6500 lines) needed to be written. Some of this code is about reifying rewrite rules into a form that the rewriter can deal with them in. Other code is about proving that the reified rewrite rules preserve interpretation and are well-formed. We wrote some plugin code to automatically generate the inductive type of base-type codes and identifier codes, as well as the two variants of the identifier-code inductive used internally in the rewriter. One interesting bit of code that resulted was a plugin that can emit an inductive declaration given the Church encoding (or eliminator) of the inductive type to be defined. We wrote a great deal of tactic code to prove basic properties about these inductive types, from the fact that one can unify two identifier codes and extract constraints on their type variables from this unification, to the fact that type codes have decidable equality. Additional plugin code was written to invoke the tactics that construct these definitions and prove these properties, so that we could generate an entire rewriter from a single command, rather than having the user separately invoke multiple commands in sequence.

In order to build the precomputed rewriter, the following actions are performed:

1. The terms and types to be supported by the rewriter package are scraped from the given lemmas.
2. An inductive type of codes for the types is emitted, and then three different versions of inductive codes for the identifiers are emitted (one with type arguments, one with type arguments supporting pattern type variables, and one without any type arguments, to be used internally in pattern-matching compilation).
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 cover categories like "Boolean equality on type codes" and "how to extract the pattern type variables from a given identifier code," and lemma categories include "type codes have decidable equality" and "the types being coded for have decidable equality" and "the identifiers all respect function extensionality."
4. The rewrite rules are reified, and we prove interpretation-correctness and well-formedness lemmas about each of them.
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.
6. The denotation functions for type and identifier codes are marked for early expansion in the kernel via the `Strategy` command; this is necessary for conversion at `Qed`-time to perform reasonably on enormous goals.

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

1. We use `etransitivity` to allow rewriting separately on the left- and right-hand-sides of an equality. Note that we do not currently support rewriting in non-equality goals, but this is easily worked around using `let v := open_constr:(_) in replace <some term> with v` and then rewriting in the second goal.
2. We revert all hypotheses mentioned in the goal, and change the form of the goal from a universally quantified statement about equality into a statement that two functions are extensionally equal. Note that this step will fail if any hypotheses are functions not known to respect function extensionality via typeclass search.
3. We reify the side of the goal that is not an existential variable using the inductive codes in the specified package; the resulting goal equates the denotation of the newly reified term with the original evar.
4. We use a lemma stating that rewriting preserves denotations of well-formed terms to replace the goal with the rewriter applied to our reified term. We use `vm_compute` to prove the well-formedness side condition reflectively. We use `vm_compute` again to reduce the application of the rewriter to the reified term.
5. Finally, we run `cbv` to unfold the denotation function, and we instantiate the evar with the resulting rewritten term.

There are a couple of steps that contribute to the trusted base. We must trust that the rewriter package we generate from the rewrite rules in fact matches the rewrite rules we want to rewrite with. This involves partially trusting the scraper, the reifier, and the glue code. We must also trust the VM we use for reduction at various points in rewriting. Otherwise, everything is checked by Coq. We do, however, depend on the axiom of function extensionality in one place in the rewriter proof; after spending a couple of hours trying to remove this axiom, we temporarily gave up.

D.3 Code from section 3, The Structure of a Rewriter

The expression language e corresponds to the inductive `expr` type defined in module `Compilers.expr` in `rewriter/src/Rewriter/Language/Language.v`.

D.3.1 Code from subsection 3.1, Pattern-Matching Compilation and Evaluation

The pattern -atching compilation step is done by the tactic `CompileRewrites` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`, which just invokes the Gallina definition named `compile_rewrites` with ever-increasing amounts of fuel until it succeeds. (It should never fail for reasons other than insufficient fuel, unless there is a bug in the code.) The workhorse function of this code is `compile_rewrites_step`.

The decision-tree evaluation step is done by the definition `eval_rewrite_rules`, also in the file `rewriter/src/Rewriter/Rewriter/Rewriter.v`. The correctness lemmas are `eval_rewrite_rules_correct` in the file `rewriter/src/Rewriter/Rewriter/InterpProofs.v` and the theorem `wf_eval_rewrite_rules` in `rewriter/src/Rewriter/Rewriter/Wf.v`. Note that the second of these lemmas, not mentioned in the paper, is effectively saying that for two related syntax trees, `eval_rewrite_rules` picks the same rewrite rule for both. (We actually prove a slightly weaker lemma, which is a bit harder to state in English.)

The third step of rewriting with a given rule is performed by the definition `rewrite_with_rule` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`. The correctness proof is `interp_rewrite_with_rule` in `rewriter/src/Rewriter/Rewriter/InterpProofs.v`. Note that the well-formedness-preservation proof for this definition in inlined into the proof `wf_eval_rewrite_rules` mentioned above.

The inductive description of decision trees is `decision_tree` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`.

The pattern language is defined as the inductive `pattern` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`. Note that we have a `Raw` version and a typed version; the pattern-matching compilation and decision-tree evaluation of Aehlig et al. [1] is an algorithm on untyped patterns and untyped terms. We found that trying to maintain typing constraints led to headaches with dependent types. Therefore when doing the actual decision-tree evaluation, we wrap all of our expressions in the dynamically typed `rawexpr` type and all of our patterns in the dynamically typed `Raw.pattern` type. We also emit separate inductives of identifier codes for each of the `expr`, `pattern`, and `Raw.pattern` type families.

We partially evaluate the partial evaluator defined by `eval_rewrite_rules` in the tactic `make_rewrite_head` in `rewriter/src/Rewriter/Rewriter/Reify.v`.

D.3.2 Code from subsection 3.2, Adding Higher-Order Features

The type NbE_t mentioned in this paper is not actually used in the code; the version we have is described in subsection 4.2 as the definition `value'` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`.

The functions `reify` and `reflect` are defined in `rewriter/src/Rewriter/Rewriter/Rewriter.v` and share names with the functions in the paper. The function `reduce` is named `rewrite_bottomup` in the code, and the closest match to NbE is `rewrite`.

D.4 Code from section 4, Scaling Challenges

D.4.1 Code from subsection 4.1, Variable Environments Will Be Large

The inductives `type`, `base_type` (actually the inductive type `base.type.type` in the supplemental code), and `expr`, as well as the definition `Expr`, are all defined in `rewriter/src/Rewriter/Language/Language.v`. The definition `denoteT` is the fixpoint `type.interp` (the fixpoint `interp` in the module `type`) in `rewriter/src/Rewriter/Language/Language.v`. The definition `denoteE` is `expr.interp`, and `DenoteE` is the fixpoint `expr.Interp`.

As mentioned above, `nbeT` does not actually exist as stated but is close to `value'` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`. The functions `reify` and `reflect` are defined in `rewriter/src/Rewriter/Rewriter/Rewriter.v` and share names with the functions in the paper. The actual code is somewhat more complicated than the version presented in the paper, due to needing to deal with converting well-typed-by-construction expressions to dynamically typed expressions for use in decision-tree evaluation and also due to the need to support early partial evaluation against a concrete decision tree. Thus the version of `reflect` that actually invokes rewriting at base types is a separate definition `assemble_identifier_rewriters`, while `reify` invokes a version of `reflect` (named `reflect`) that does not call rewriting. The function named `reduce` is what we call `rewrite_bottomup` in the code; the name `Rewrite` is shared between this paper and the code. Note that we eventually instantiate the argument `rewrite_head` of `rewrite_bottomup` with a partially evaluated version of the definition named `assemble_identifier_rewriters`. Note also that we use fuel to support `do_again`, and this is used in the definition `repeat_rewrite` that calls `rewrite_bottomup`.

The correctness theorems are `InterpRewrite` in `rewriter/src/Rewriter/Rewriter/InterpProofs.v` and `Wf_Rewrite` in `rewriter/src/Rewriter/Rewriter/Wf.v`.

Packages containing rewriters and their correctness theorems are in the record `VerifiedRewriter` in `rewriter/src/Rewriter/Rewriter/ProofsCommon.v`; a package of this type is then passed to the tactic `Rewrite_for_gen` from `rewriter/src/Rewriter/Rewriter/AllTactics.v` to perform the actual rewriting. The correspondence of the code to the various steps in rewriting is described in the second list of subsubsection D.2.1.

D.4.2 Code from subsection 4.2, Subterm Sharing is Crucial

To run the P-256 example in the copy of Fiat Cryptography attached as a code supplement, after building the library, run the code

```
Require Import Crypto.Rewriter.PerfTesting.Core.
Require Import Crypto.Util.Option.

Import WordByWordMontgomery.
Import Core.RuntimeDefinitions.

Definition p : params
  := Eval compute in invert_Some
       (of_string "2^256-2^224+2^192+2^96-1" 64).

Goal True.
  (* Successful run: *)
  Time let v := (eval cbv
    -[Let_In
      runtime_nth_default
      runtime_add
      runtime_sub
      runtime_mul
      runtime_opp
      runtime_div
      runtime_modulo
      RT_Z.add_get_carry_full
```

```
        RT_Z.add_with_get_carry_full
        RT_Z.mul_split]
     in (GallinaDefOf p)) in
     idtac.
   (* Unsuccessful OOM run: *)
   Time let v := (eval cbv
     -[(*Let_In*)
        runtime_nth_default
        runtime_add
        runtime_sub
        runtime_mul
        runtime_opp
        runtime_div
        runtime_modulo
        RT_Z.add_get_carry_full
        RT_Z.add_with_get_carry_full
        RT_Z.mul_split]
     in (GallinaDefOf p)) in
     idtac.
Abort.
```

The UnderLets monad is defined in the file `rewriter/src/Rewriter/Language/UnderLets.v`.

The definitions `nbeT'`, `nbeT`, and `nbeT_with_lets` are in `rewriter/src/Rewriter/Rewriter/Rewriter.v` and are named `value'`, `value`, and `value_with_lets`, respectively.

D.4.3 Code from subsection 4.3, Rules Need Side Conditions

The "variant of pattern variable that only matches constants" is actually special support for the reification of `ident.literal` (defined in the module `RewriteRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`) threaded throughout the rewriter. The apostrophe notation `'` is also introduced in the module `RewriteRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`. The support for side conditions is handled by permitting rewrite-rule-replacement expressions to return `option expr` instead of `expr`, allowing the function `expr_to_pattern_and_replacement` in the file `rewriter/src/Rewriter/Rewriter/Reify.v` to fold the side conditions into a choice of whether to return `Some` or `None`.

D.4.4 Code from subsection 4.4, Side Conditions Need Abstract Interpretation

The abstract-interpretation pass is defined in `fiat-crypto/src/AbstractInterpretation/`, and the rewrite rules handling abstract-interpretation results are the Gallina definitions `arith_with_casts_rewrite_rulesT`, in addition to `strip_literal_casts_rewrite_rulesT`, in addition to `fancy_with_casts_rewrite_rulesT`, and finally in addition to `mul_split_rewrite_rulesT`, all defined in `fiat-crypto/src/Rewriter/Rules.v`.

The `clip` function is the definition `ident.cast` in `fiat-crypto/src/Language/PreExtra.v`.

D.5 Code from section 5, Evaluation

D.5.1 Code from subsection 5.1, Microbenchmarks

This code is found in the files in `rewriter/src/Rewriter/Rewriter/Examples/`. We ran the microbenchmarks using the code in `rewriter/src/Rewriter/Rewriter/Examples/PerfTesting/Harness.v` together with some `Makefile` cleverness. The file names correspond to the section titles in Appendix A.

D.5.2 Code from subsection 5.2, Macrobenchmark: Fiat Cryptography

The rewrite rules are defined in `fiat-crypto/src/Rewriter/Rules.v` and proven in the file `fiat-crypto/src/Rewriter/RulesProofs.v`. They are turned into rewriters in the various files in `fiat-crypto/src/Rewriter/Passes/`. The shared inductives and definitions are defined in the Coq source files `fiat-crypto/src/Language/IdentifiersBasicGENERATED.v`, `fiat-crypto/src/Language/IdentifiersGENERATED.v`, and `fiat-crypto/src/Language/IdentifiersGENERATEDProofs.v`. Note that we invoke the subtactics of the **Make** command manually to increase parallelism in the build and to allow a shared language across multiple rewriter packages.