A Framework for Building Verified Partial Evaluators

ANONYMOUS AUTHOR(S)

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 [Leroy 2009] and the seL4 verified operating-system microkernel [Klein et al. 2009]. 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* [Jones et al. 1993], 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 [Rompf and Odersky 2010].

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 every step of 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 [Erbsen et al. 2019], a Coq library that generates code for big-integer modular

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

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 partial-evaluation method 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 also being easily applied to new partial-evaluation settings.

1.1 Motivating Example: Cryptographic Arithmetic

First, why would partial evaluation make sense for big-integer arithmetic? Traditional libraries like the GNU Multi Precision Arithmetic Library[1] handle this domain with dynamic allocation of data structures representing integers. However, it turns out that, when the large prime modulus of arithmetic is known, as it is for Internet-standard crypto algorithms like in HTTPS/TLS, it is possible to do much better: C code customized to a modulus can perform in the neighborhood of 10X better (and handwritten assembly ekes out more performance still). Thus, the state of practice has been that crypto experts *handwrite complete arithmetic implementations from scratch in C or assembly for each new prime modulus and target machine-word size.* The method for doing so is fairly systematic, but it had not been formalized until Fiat Cryptography codified it in a library of functional programs. Partial evaluation specializes these programs to first-order code that is easily pretty-printed as C code with essentially the same algorithmic content as what the experts were writing. Fiat Cryptography now generates the fastest-known C code for all elliptic curves.

Let us make the example more concrete with a simplified excerpt of a library function for multiplication specialized to a representation of big integers using five machine-word-sized digits, as shown in Figure 1. The most interesting parameter to a Fiat Cryptography library function is a choice of representation for integers. Definition `weights` encodes that choice in the figure, as a list of weights. For instance, with a weight list $[w_1; w_2]$, the digit list $[d_1; d_2]$ would be interpreted as the number $d_1 \cdot w_1 + d_2 \cdot w_2$. We will not delve into more detail on the code, just noting that the final definition `mulmod` is for the modular-multiplication routine we are looking for, and some intermediate definitions convert into and out of a novel intermediate number representation called "associational." We have also simplified the code a bit for readability by doing some partial evaluation already, inlining the representation list `weights` into other definitions, while the real library takes `weights` as a parameter to `mulmod` and several other functions.

We would like to reduce the code

$$\texttt{mulmod}\ [f_1; f_2; f_3; f_4; f_5]\ [g_1; g_2; g_3; g_4; g_5]$$

to the "flat" code shown in Figure 2. This code computes the multiplication result when the input digits (known only at runtime) are f_i and g_i.

Ideally, we would also eliminate the $1\cdot$ and $+0$, but reduction (based on Coq's *definitional equality*) alone cannot accomplish that effect on binary integers (which should be used for good reduction performance). Also, in general it is important to retain `let ··· in ···` expressions from the library in the specialized code, to avoid term-size blow-up by losing natural sharing. Our example would work out fine if we inlined all `let`-bound variables, but let us still trace through the challenges of sharing preservation, for didactic reasons. We must go through fairly painful acrobatics. We must (a) CPS-transform `from_associational` so that we can (b) let-bind the second projection of

[1] https://gmplib.org/

$$
\begin{aligned}
&\texttt{weights}\ n &&= \texttt{map}\ (\lambda i.\, 2^{51\cdot i})\ (\texttt{seq}\ 0\ n)\\
&\texttt{to\_associational}\ n\ f &&= \texttt{combine}\ (\texttt{weights}\ n)\ f\\
&\texttt{assoc\_mul}\ f\ g &&= \texttt{flat\_map}\ (\lambda x.\, \texttt{map}\ (\lambda y.\, (x_1 \cdot y_1, x_2 \cdot y_2))\ g)\ f\\
&\texttt{zeros}\ n &&= \texttt{repeat}\ 0\ n\\
\\
&\texttt{place}\ (w, v)\ 0 &&= (0, w \cdot v)\\
&\texttt{place}\ (w, v)\ i &&= \texttt{if}\ w \bmod 2^{51\cdot i} = 0\ \texttt{then}\ (i, (w/2^{51\cdot i}) \cdot v)\ \texttt{else place}\ (w, v)\ (i-1)\\
\\
&\texttt{from\_associational}\ n\ f &&= \texttt{fold\_right}\\
& &&\qquad (\lambda t\ ls.\, \texttt{let}\ p = \texttt{place}\ t\ (n-1)\ \texttt{in}\ \texttt{add\_to\_nth}\ p_1\ p_2\ ls)\\
& &&\qquad (\texttt{zeros}\ n)\\
& &&\qquad f\\
&\texttt{split}\ s\ p &&= \texttt{let}\ (\texttt{hi}, \texttt{lo}) = \texttt{partition}\ (\lambda t.\, t_1 \bmod s = 0)\ p\ \texttt{in}\\
& &&\quad (\texttt{lo}, \texttt{map}\ (\lambda t.\, (t_1/s, t_2))\ hi)\\
&\texttt{reduce}\ s\ c\ p &&= \texttt{let}\ (\texttt{lo}, \texttt{hi}) = \texttt{split}\ s\ p\ \texttt{in lo} \mathbin{++} \texttt{assoc\_mul}\ c\ \texttt{hi}\\
\\
&\texttt{mul}\ f\ g &&= \texttt{assoc\_mul}\ (\texttt{to\_associational}\ 5\ f)\ (\texttt{to\_associational}\ 5\ g)\\
&\texttt{mulmod}\ f\ g &&= \texttt{from\_associational}\ 5\ \left(\texttt{reduce}\ 2^{255}\ [(1, 19)]\ (\texttt{mul}\ f\ g)\right)
\end{aligned}
$$

Fig. 1. Code to specialize for multiplication mod $2^{255} - 19$ on 64-bit processors

$$
\begin{array}{ll}
\texttt{let}\ p2 = 1 \cdot (f1 \cdot g1)\ \texttt{in} & \texttt{let}\ p3 = 1 \cdot (f1 \cdot g2)\ \texttt{in}\\
\texttt{let}\ p4 = 1 \cdot (f1 \cdot g3)\ \texttt{in} & \texttt{let}\ p5 = 1 \cdot (f1 \cdot g4)\ \texttt{in}\\
\texttt{let}\ p6 = 1 \cdot (f1 \cdot g5)\ \texttt{in} & \texttt{let}\ p7 = 1 \cdot (f2 \cdot g1)\ \texttt{in}\\
\texttt{let}\ p8 = 1 \cdot (f2 \cdot g2)\ \texttt{in} & \texttt{let}\ p9 = 1 \cdot (f2 \cdot g3)\ \texttt{in}\\
\texttt{let}\ p10 = 1 \cdot (f2 \cdot g4)\ \texttt{in} & \texttt{let}\ p11 = 1 \cdot (f3 \cdot g1)\ \texttt{in}\\
\texttt{let}\ p12 = 1 \cdot (f3 \cdot g2)\ \texttt{in} & \texttt{let}\ p13 = 1 \cdot (f3 \cdot g3)\ \texttt{in}\\
\texttt{let}\ p14 = 1 \cdot (f4 \cdot g1)\ \texttt{in} & \texttt{let}\ p15 = 1 \cdot (f4 \cdot g2)\ \texttt{in}\\
\texttt{let}\ p16 = 1 \cdot (f5 \cdot g1)\ \texttt{in} & \texttt{let}\ p17 = 1 \cdot (19 \cdot (f2 \cdot g5))\ \texttt{in}\\
\texttt{let}\ p18 = 1 \cdot (19 \cdot (f3 \cdot g4))\ \texttt{in} & \texttt{let}\ p19 = 1 \cdot (19 \cdot (f3 \cdot g5))\ \texttt{in}\\
\texttt{let}\ p20 = 1 \cdot (19 \cdot (f4 \cdot g3))\ \texttt{in} & \texttt{let}\ p21 = 1 \cdot (19 \cdot (f4 \cdot g4))\ \texttt{in}\\
\texttt{let}\ p22 = 1 \cdot (19 \cdot (f4 \cdot g5))\ \texttt{in} & \texttt{let}\ p23 = 1 \cdot (19 \cdot (f5 \cdot g2))\ \texttt{in}\\
\texttt{let}\ p24 = 1 \cdot (19 \cdot (f5 \cdot g3))\ \texttt{in} & \texttt{let}\ p25 = 1 \cdot (19 \cdot (f5 \cdot g4))\ \texttt{in}\\
\texttt{let}\ p26 = 1 \cdot (19 \cdot (f5 \cdot g5))\ \texttt{in} & \\
[p23 + (p20 + (p18 + (p17 + (p2 + 0)))); & p24 + (p21 + (p19 + (p7 + (p3 + 0))));\\
\ p25 + (p22 + (p11 + (p8 + (p4 + 0)))); & p26 + (p14 + (p12 + (p9 + (p5 + 0))));\\
\ p16 + (p15 + (p13 + (p10 + (p6 + 0))))] &
\end{array}
$$

Fig. 2. Code specialized for multiplication mod $2^{255} - 19$ on 64-bit processors

$p = \texttt{place}\ t\ (n-1)$ and (c) mark this let-binder as well as the multiplications and additions on the second components of pairs as non-reducing. Although CPS-transforming all of our definitions is painful in general, it is not too bad in this particular case:

$$\begin{aligned}\texttt{from\_associational}\ n\ f\ K \quad &= \texttt{fold\_right} \\ &\quad (\lambda t\ K\ ls.\ \texttt{let}\ q = \texttt{place}\ t\ (n-1)\ \texttt{in} \\ &\qquad\quad \texttt{let}\ p = q_2\ \texttt{in}\ K\ (\texttt{add\_to\_nth}\ q_1\ p\ ls)) \\ &\quad K\ f\ (\texttt{zeros}\ n)\end{aligned}$$

If we additionally mark the relevant definitions and `let ⋯ in ⋯` for non-unfolding, then Coq's kernel reduction takes about 3–4 seconds. If we do the same thing using 10-limb base-25.5 numbers (the most common representation for this particular elliptic curve on 32-bit processors), partial reduction takes just over 20 seconds to produce about 100 lines of code. Producing 1000 lines of code would likely take at least 200 seconds. Further code transformations become quite slow, at this point.

For example, trying to rewrite away the +0 and 1· expressions requires setoid-based rewriting for rewriting under the binders of a `let ⋯ in ⋯`, and proof size blows up quite quickly. Proving anything about these terms is also quite slow unless the proof author is quite careful to track reduction behavior everywhere; as a simple example, proving that the reduced term is equal to the term we started with takes just as long as reducing it in the first place (and then that cost is paid again, in Coq, at `Qed` time). Additionally, this sort of partial reduction is nearly impossible to debug. With even one place where reduction is blocked during unfolding (for example, because one definition that should have been CPS-converted was not, or one use of an arithmetic operation was not marked correctly as not-to-be-unfolded), Coq can easily spend hours reducing down the wrong path. As an anecdote, in our experiments trying to prove outputs of similar reductions to the example here, we ran into many cases where it takes Coq over 800 hours to verify that two terms are equal, even when our partially reduced code is only a dozen or so lines long. In our new framework, it takes only about 0.3 seconds to do the partial reduction on the given example, and about 1 second for the 10-limb base-25.5 version.

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 [Martin-Dorel 2018], and support for native arrays is in the works.

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

The contribution of this paper is a new Coq library embodying **the first partial-evaluation approach to satisfy the following criteria**.

- It integrates with a general-purpose, foundational proof assistant.
- 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*. The former includes standard support for higher-order functions via λ-calculus reduction rules, while the latter exposes customized algebraic simplification, and the two need to be interleaved appropriately.
- It actually allows a **smaller trusted base** than in widely used systems like Coq, where partial evaluation *only requires that the kernel know how to normalize terms fully*, as in standard functional-language interpreters that the community knows very well how to optimize. (We still rely on more specialized reduction to convert to final form from residual syntax trees, with cost proportional to sizes of result terms, rather than proportional to intermediate-term sizes as in the bottlenecks we sketched earlier.)
- It also allows **extraction of standalone partial evaluators**, as even within Coq the technique works by generating syntax-tree-manipulating functional programs with proofs of soundness.

The secret sauce of our approach is a novel mix of four big ideas from the type-theory and functional-programming communities.

- *Proof by reflection* [Boutin 1997], which encodes logical goals explicitly with syntax trees that can be operated on by proved functional programs – allowing us to delegate to the kernel the execution of a partial-evaluation process embodied in such a proved program
- *Parametric higher-order abstract syntax (PHOAS)* [Chlipala 2008], a good encoding for such syntax trees of a higher-order functional language – allowing us to delegate all variable and environment management to the kernel
- *Normalization by evaluation (NbE)* [Berger and Schwichtenberg 1991], a method of bootstrapping syntactic object-language reduction on top of semantic metalanguage reduction – allowing us to delegate all simplification of higher-order function calls to the kernel
- *Pattern-matching compilation for functional languages*, as in OCaml [Maranget 2008] – our method for fast bottom-up rewriting with custom rules

In short, the strategy is to avoid duplication with what a minimal proof-assistant kernel must provide anyway, while wrapping with suitable hooks for tweaking algebraic simplification, without interfering with the kernel's ability to run a full partial evaluation as one full normalization of a term in the logic. We want to take as much advantage as possible of engineering effort that has gone into the kernel's normalization engine.

The next few sections introduce the ingredients in detail. In section 2, we describe the core of our system: rewriting on first-order terms. In section 3, we describe how to do reduction on higher-order terms via NbE. In section 4, we describe how we make use of PHOAS to implement the algorithms in a relatively straightforward manner. In section 5, we describe other features that we support in our rewriter: type-level parametric polymorphism, let-lifting, and a design pattern for applying custom reflective provers to establish side conditions of rewrite rules. Finally, in section 6, we describe particular challenges of implementing and proving this rewriter in Coq, mostly as relate to dependent types.

Our implementation is included as non-anonymous supplementary material.

2 PATTERN-MATCHING COMPILATION AND EVALUATION

We begin with the core of our reduction engine, providing term rewriting in the classic style applicable to first-order languages; we build up to higher-order terms in the following sections. The key parts of the rewriting core are pattern-matching compilation, decision-tree evaluation, and rewriting with particular rewrite rules.

Our expression language has a simple inductive syntax:

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

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 $+$, $\cdot$, and literal constants.

We use $[\![e]\!]_e$ to mean the denotation of an expression e, which is defined in a standard way.

Let us now consider some simple rewrite rules, where (following Coq's tactic language) we preface pattern variables with question marks at their binding sites:

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

There are three steps to turn these 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.

2.1 Pattern-Matching Compilation

This part of the rewriter does not need to be verified, because the rewriter-compiler is proven correct independent of the decision tree used. Note that we could avoid this stage altogether and simply try each rewrite rule in sequence, at significant performance cost.

We follow Maranget [2008], who describes compilation of pattern matches in OCaml to decision trees that eliminate needless repeated work (for example, decomposing an expression into $x + y + z$

only once even if two different rules match on that pattern). We have not yet implemented any of the optimizations described therein for finding *minimal* decision trees.

2.1.1 The type of decision trees. 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. The cases of a `decision_tree` are:

- `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 its 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`.

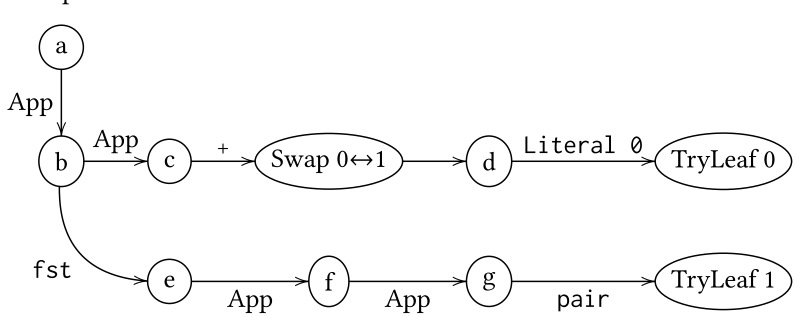
For the two rules mentioned above, their patterns are:

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

The pattern language has a slightly simpler grammar than for expressions:

$$p ::= \texttt{App}\ p_1\ p_2 \mid \texttt{Ident}\ i \mid \texttt{Wildcard}$$

The decision tree produced is



where every non-swap node implicitly has a "default" case arrow to `Failure`. See subsubsection 2.1.3 for the complete algorithm of compiling patterns.

2.1.2 Evaluating decision trees. Let us now consider evaluating the above decision tree on two examples, where our literals are integer constants, addition (+) (used with infix syntax where appropriate), pair first projection $\texttt{fst}_{\mathbb{Z},\mathbb{Z}}$, and pair construction $\texttt{pair}_{\mathbb{Z},\mathbb{Z}}$ (where we abbreviate $\texttt{pair}_{\mathbb{Z},\mathbb{Z}}\ a\ b$ as (a, b)).

(1) $n + 0$
(2) $\texttt{fst}_{\mathbb{Z},\mathbb{Z}}(n + 0, n)$

The first example:

- To begin with, we are matching the singleton list $[n + 0]$ against the entire decision tree. The first node is a `Switch`, and the head of the first element of our list is an application, so we are in the application case; our new list is the head of the application, followed by its argument, followed by the tail of our original list (which is empty).
- Now we are matching the list $[(n+); 0]$ against the tree starting at node (b). The first node is a `Switch`, and the head of the first element of our list is again an application, so we are in the application case.
- Now we are matching the list $[(+); n; 0]$ against the tree starting at node (c). The first node is a `Switch`, and the head of the first element of our list is an identifier node, in particular +. So we select the + case from the identifier list of the `Switch` node and drop the head of our list.
- Now we are matching the list $[n; 0]$ against the tree starting at the node labeled Swap $0 \leftrightarrow 1$. Hence we swap the head (0^{th}) element with the element in position 1.
- Now we are matching the list $[0; n]$ against the tree starting at node (d). The first node is a `Switch` node, and the head of our list is the identifier `Literal 0`.
- Now we are matching the list $[n]$ against the tree starting with `TryLeaf 0`, which says to try rewriting with the first rewrite rule and implicitly, if it fails, then to continue with the tree `Failure`. Hence we have selected to rewrite with $n + 0 \to n$, which succeeds on our expression, giving the expression n.

The second example:

- To begin with, we are matching the singleton list $[\texttt{fst}_{\mathbb{Z},\mathbb{Z}}(n + 0, n)]$ against the entire decision tree. The first node is a `Switch`, and the head of the first element of our list is an application, so we are in the application case; our new list is the head of the application, followed by its argument, followed by the tail of our list.
- Now we are matching the list $[\texttt{fst}_{\mathbb{Z},\mathbb{Z}}; (n + 0, n)]$ against the tree starting with node (b). The first node is a `Switch`, and the head of the first element of our list is the identifier `fst`. We choose the corresponding tree of the identifier case and drop the head of the list.
- Now we are matching the list $[(n + 0, n)]$ against the tree starting with node (e). The first node is a `Switch`, and the head of the first element of our list is an application, so we are in the application case; our new list is the head of the application, followed by its argument, followed by the tail of our list.
- Now we are matching the list $[(\texttt{pair}\ (n + 0)); n]$ against the tree starting with node (f). The first node is a `Switch`, and the head of the first element of our list is again an application.
- Now we are matching the list $[\texttt{pair}; (n + 0); n]$ against the tree starting with node (g). The first node is a `Switch`, and the head of the first element of our list is the identifier `pair`.
- Finally, we are matching the list $[(n + 0); n]$ against the tree starting with `TryLeaf 1`. This says to try rewriting with the second rewrite rule, and, implicitly, if it fails, then to continue with the tree `Failure`. Hence we have selected to rewrite with $\texttt{fst}_{\mathbb{Z},\mathbb{Z}}(?x, ?y) \to x$, which succeeds on our expression, giving the expression $n + 0$.

Let us look at the general form of the decision-tree-evaluation procedure, where, as in the next few sections, we omit some Coq-encoding complications that we return to in section 6. This procedure takes in a list of expressions currently being matched against (called a vector in Maranget [2008]), a decision tree, and a procedure that actually performs the rewriting on the overall expression and returns either the success value or an indication of failure. Here is the general form, where continuations k and defined functions $\mathcal{E}_{\text{dt}}$ and $\mathcal{E}_{\text{ident}}$ are partial.

$$\mathcal{E}_{\text{dt}}(\ell, \texttt{TryLeaf}(n, d_f), k) = \begin{cases} r, & \text{if } k(n) = r \\ \mathcal{E}_{\text{dt}}(\ell, d_f, k), & \text{otherwise} \end{cases}$$

$$\mathcal{E}_{\text{dt}}(\texttt{App}(f, x) :: \ell, \texttt{Switch}(\_, d_{\text{app}}, d_*), k) = \begin{cases} r, & \text{if } \mathcal{E}_{\text{dt}}(f :: x :: \ell, d_{\text{app}}, k) = r \\ \mathcal{E}_{\text{dt}}(\texttt{App}(f, x) :: \ell, d_*, k), & \text{otherwise} \end{cases}$$

$$\mathcal{E}_{\text{dt}}(\texttt{Ident}(i) :: \ell, \texttt{Switch}(\ell', \_, d_*), k) = \mathcal{E}_{\text{ident}}(i, \ell, \ell', d_*, k)$$

$$\mathcal{E}_{\text{dt}}(\ell, \texttt{Swap}(i, d), k) = \mathcal{E}_{\text{dt}}(\texttt{swap}(0, i, \ell), d, k)$$

$$\mathcal{E}_{\text{ident}}(i, \ell, (i', d_{i'}) :: \ell', d_*, k) = \begin{cases} r, & \text{if } i = i' \text{ and } \mathcal{E}_{\text{dt}}(\ell, d_{i'}, k) = r \\ \mathcal{E}_{\text{ident}}(i, \ell, \ell', d_*, k), & \text{otherwise} \end{cases}$$

$$\mathcal{E}_{\text{ident}}(i, \ell, [], d_*, k) = \mathcal{E}_{\text{dt}}(\texttt{Ident}(i) :: \ell, d_*, k)$$

Note that we use Coq's normal partial evaluation to turn our 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 reducing with the matcher we get out dominates this time of meta-compilation.

Though we have not yet presented the code that acts on a choice of rewrite rule, assume for now that it is properly composed with decision-tree evaluation, so that we can demonstrate the effect of evaluation on our running example:

$$?n + 0 \to n$$

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

For this pair of rewrite rules, we get out the 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.
```

Note that the correctness lemma for decision-tree evaluation is simple: for any decision tree, the output of evaluation must be either $\lightning$ (the mark for failure) or the result of invoking the continuation on some rewrite rule. In math:

$$\forall \vec{e}, d, k.\, \texttt{eval\_decision\_tree}\ \vec{e}\ d\ k = \lightning \lor \exists n.\, \texttt{eval\_decision\_tree}\ \vec{e}\ d\ k = k\ n$$

2.1.3 Compiling patterns. We wrote in subsubsection 2.1.1 about the output decision trees of pattern compilation. Here we describe the complete algorithm for compiling them, again in Haskell-style pseudocode. Note that this algorithm is quite close to the one of [Maranget 2008]. We use terminating but non-structural recursion; in Coq, we use a large amount of fuel rather than tracking actual size, for convenience.

We use a function mapP that is like map but allows the function argument to fail on some inputs, in which case those inputs do not contribute to the output list; and we use higher-order function splitAt to break a list into its longest prefix of elements *not* satisfying a predicate, followed by all the rest.

$$
\begin{aligned}
C([]) &= \texttt{Failure} \\
C((n,\vec{p}) :: \ell) &= \begin{cases}
\texttt{TryLeaf}(n, C(\ell)), & \text{if first } \neg\texttt{Wildcard}\ \vec{p} = \lightning \\
\left(\begin{array}{l}
\text{let } (\vec{p}_{NW}, \vec{p}_d) = \text{splitAt } \texttt{Wildcard}\ ((n,\vec{p}) :: \ell) \text{ in} \\
\text{let } A = C(\text{mapP } \mathcal{F}_{\text{app}}\ \vec{p}_{NW}) \text{ in} \\
\text{let } I = \text{map } (\lambda i.\ (i, C(\text{mapP } (\mathcal{F}_{\text{ident}}\ i)\ \vec{p}_{NW}))) \\
\quad (\text{uniqueHeads } \vec{p}_{NW}) \text{ in} \\
\texttt{Switch}(I, A, C(\vec{p}_d))
\end{array}\right), & \text{if first } \neg\texttt{Wildcard}\ \vec{p} = 0 \\
\texttt{Swap}(i, C(\text{map } (\lambda(n,\vec{p}').\ (n, \text{swap } 0\ i\ \vec{p}'))\ ((n,\vec{p}) :: \ell))), & \text{if first } \neg\texttt{Wildcard}\ \vec{p} = i
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_{\text{app}}(n,\vec{p}) &= \left\{(n, f :: x :: \vec{p}'), \quad \text{if } \vec{p} = \text{App}(f,x) :: \vec{p}'\right. \\
\mathcal{F}_{\text{ident}}(i)(n,\vec{p}) &= \left\{(n, \vec{p}'), \quad \text{if } \vec{p} = \texttt{Ident}(i) :: \vec{p}'\right.
\end{aligned}
$$

Note that `compile_rewrites` needs no correctness theorem, as the correctness theorem for `eval_decision_tree` is universally quantified over all decision trees. One might think of decision trees as merely a hint; proving soundness does not require knowing anything about the hint structure. Proving completeness, on the other hand, would require a much more interesting correctness lemma for `compile_rewrites`.

2.1.4 *Actually rewriting with rewrite rules.*

We now write down the final missing step of simple rewrite-rule replacement: actually evaluating the rewrite rule on a given expression. There are two parts to this step: binding values to pattern variables and evaluating the replacement expression under those bindings. This partial function extracts a nested tuple of wildcard values, whose shape mirrors the pattern.

$$
\begin{aligned}
\mathcal{U}_p(e, \texttt{Wildcard}) &= e \\
\mathcal{U}_p(\text{App}(f_e, x_e), \text{App}(f_p, x_p)) &= (\mathcal{U}_p(f_e, f_p), \mathcal{U}_p(x_e, x_p)) \\
\mathcal{U}_p(\texttt{Ident}(i), \texttt{Ident}(i)) &= ()
\end{aligned}
$$

We can also write down a simple semantics for patterns:

$$
\begin{aligned}
[\![\texttt{Wildcard}]\!]_p(e) &= [\![e]\!]_e \\
[\![\text{App}(f_p, x_p)]\!]_p(f_d, x_d) &= \left([\![f_p]\!]_p(f_d)\right)([\![x_p]\!]_p(x_d)) \\
[\![\texttt{Ident}(i_p)]\!]_p() &= [\![i_p]\!]_i
\end{aligned}
$$

We can write down a simple correctness lemma for any rewrite rule: that the interpretation of the rewrite rule on binding data must equal the interpretation of the pattern on the same binding data. For a rewrite rule that combines a pattern p with a body expression b, expressed as a function over a nested tuple of wildcard values produced by matching p, here is the condition.

$$
\forall d.\ [\![p]\!]_p(d) = [\![b(d)]\!]_e
$$

3 ADDING HIGHER-ORDER FEATURES

Fast rewriting at the top level of a term is the key ingredient for supporting customized algebraic simplification. However, not only do want to rewrite throughout the structure of a term, but we also want to integrate with simplification of higher-order terms, in a way where we can prove to Coq that our syntax-simplification function always terminates. Normalization by evaluation

(NbE) [Berger and Schwichtenberg 1991] is an elegant technique for adding the latter aspect, in a way where we avoid needing to implement our own λ-term reducer or prove it terminating.

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

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

using the rewrite rule

$$?n + 0 \to n$$

3.1 Normalization by Evaluation

We begin by reviewing standard NbE, a technique for performing full β-reduction in a simply typed term in a guaranteed-terminating way. The simply typed λ-calculus syntax we use is:

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

where v is for variables, c is for constants, and b is for base types.

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

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

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

Now the core of NbE 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.

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

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

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

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

These definitions apply some of the usual corner-cutting that we see in on-paper descriptions of λ-term transformations. We write v for object-language variables and x for metalanguage (Coq) variables, and we overload λ notation using the metavariable kind to signal whether we are building a host λ or a λ syntax tree for the embedded language. The crucial first clause for reduce replaces object-language variable v with fresh metalanguage variable x, and then we are somehow tracking that all free variables in an argument to reduce must have been replaced with metalanguage variables by the time we reach them. We reveal in section 4 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.

3.2 Fusing NbE with Rewriting

To fuse NbE with rewriting, we only modify the constant case of `reduce`. First, we wrap our earlier rewriting engine in a convenient top-level form, writing d for the decision tree we compiled from a list of rewrite rules, whose ith pattern is p_i.

$$\text{rewrite-head}(e) = \mathcal{E}_{\text{dt}}([e], d, \lambda n.\, \mathcal{U}_p(e, p_n))$$

In the constant case, we still reflect the constant, but underneath the binders introduced by full η-expansion, we perform one instance of rewriting. The code for the rewriting (with some simplifications that we fix in section 6) is then

$$\begin{aligned}
\text{reflect}^{\mathcal{R}}_t &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reflect}^{\mathcal{R}}_{t_1 \to t_2}(e) &= \lambda v.\, \text{reflect}^{\mathcal{R}}_{t_2}(e\ \ (\text{reify}_{t_1}(v))) \\
\text{reflect}^{\mathcal{R}}_b(e) &= \text{rewrite-head}(e)
\end{aligned}$$

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

$$\begin{aligned}
\text{rewrite-bottomup} &: \text{expr}(t) \to \text{expr}(t) \\
\text{rewrite-bottomup}(e) &= \text{reify}(\text{reduce}^{\mathcal{R}}(e))
\end{aligned}$$

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.

Let us now evaluate the example from the start of this subsection, using the new procedure. We will be careful to write x, y, z for metalanguage variables and u, v, w for object-language variables (whose occurrences are implicit uses of a variable constructor for syntax trees). We also follow the convention from before that metalanguage free variable z is allowed to be used as an object-language variable (wrapped in a variable constructor, where needed).

$$
\begin{aligned}
&\text{rewrite-bottomup}((\lambda u\ v\ w.\, u\ v\ w)\ (+)\ z\ 0)\\
&= \text{reify}(\text{reduce}^{\mathcal{R}}((\lambda u\ v\ w.\, u\ v\ w)\ (+)\ z\ 0))\\
&= \text{reify}(\text{reduce}^{\mathcal{R}}(\lambda u\ v\ w.\, u\ v\ w)(\text{reduce}^{\mathcal{R}}(+))(\text{reduce}^{\mathcal{R}}(z))(\text{reduce}^{\mathcal{R}}(0)))\\
&= \text{reify}(\text{reduce}^{\mathcal{R}}(\lambda u\ v\ w.\, u\ v\ w)(\text{reflect}^{\mathcal{R}}(+))(\text{reflect}^{\mathcal{R}}(z))(\text{reflect}^{\mathcal{R}}(0)))\\
&= \text{reify}(\text{reduce}^{\mathcal{R}}(\lambda u\ v\ w.\, u\ v\ w)(\lambda x_0\ x_1.\, \text{reflect}^{\mathcal{R}}(\text{reify}(x_0) + \text{reify}(x_1)))(\text{rewrite-head}(z))(\text{rewrite-head}(0)))\\
&= \text{reify}(\text{reduce}^{\mathcal{R}}(\lambda u\ v\ w.\, u\ v\ w)(\lambda x_0\ x_1.\, \text{reflect}^{\mathcal{R}}(\text{reify}(x_0) + \text{reify}(x_1)))(z)(0))\\
&= \text{reify}((\lambda x\ y_0\ y_1.\, \text{reduce}^{\mathcal{R}}(x\ y_0\ y_1))(\lambda x_0\ x_1.\, \text{reflect}^{\mathcal{R}}(\text{reify}(x_0) + \text{reify}(x_1)))(z)(0))\\
&= \text{reify}((\lambda x\ y_0\ y_1.\, x\ y_0\ y_1)(\lambda x_0\ x_1.\, \text{reflect}^{\mathcal{R}}(\text{reify}(x_0) + \text{reify}(x_1)))(z)(0))\\
&= \text{reify}((\lambda x_0\ x_1.\, \text{reflect}^{\mathcal{R}}(\text{reify}(x_0) + \text{reify}(x_1)))(z)(0))\\
&= \text{reify}(\text{reflect}^{\mathcal{R}}(\text{reify}(z) + \text{reify}(0)))\\
&= \text{reify}(\text{reflect}^{\mathcal{R}}(z + 0))\\
&= \text{reify}(\text{rewrite-head}(z + 0))\\
&= \text{reify}(z)\\
&= z
\end{aligned}
$$

Notice how mundane λ-reduction did most of the work for bringing the parts of the starting term together, and then rewrite-head materialized at just the right point at the end, thanks to its use in the base-type case of reflect$^{\mathcal{R}}$. However, we have been accumulating enough notational debt that we should turn to Coq encoding details.

4 ENCODING TERM SYNTAX

Parametric higher-order abstract syntax (PHOAS), first described by Chlipala [2008], is a dependently typed encoding of syntax where binders are managed by the enclosing type system. It allows for relatively easy implementation and proof for NbE, so we adopted it for our framework.

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

```
Inductive type := arrow (s d : type) | base (b : base_type).
Infix "->" := arrow.
Inductive expr {var : type -> Type} : type -> Type :=
| Var {t} (v : var t) : expr t
| Abs {s d} (f : var s -> expr d) : expr (s -> d)
| App {s d} (f : expr (s -> d)) (x : expr s) : expr d
| Const {t} (c : const t) : expr 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 by the time we reach them.

```
Fixpoint denoteE {t} (e : expr (var := 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 arbitrarily.

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

Fixpoint reify {var t} : nbeT (var := var) t -> expr (var := 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 := var) t -> nbeT (var := 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 (var := nbeT (var := var)) t) : nbeT (var := 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 := 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.

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

To understand how we now apply this 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 $\texttt{Rewrite}(E)$ by *full reduction via compilation to native code*, since we carefully designed $\texttt{Rewrite}(E)$ 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.

The payoffs from fully satisfying Coq's type checker are:

(1) We know that this procedure always terminates, and Coq's kernel is therefore willing to run the procedure for us implicitly during proof checking. In a sense, we have bootstrapped this reduction strategy into the conversion rule of the type theory.
(2) In that setting, all bookkeeping about variable binding and environments is handled by the kernel, whose implementation in OCaml allows certain efficient implementation strategies not available to us in the logic.

5 ADDITIONAL USER-VISIBLE FEATURES

Our framework and best practices for using it incorporate a few other wrinkles that users see directly, which we describe here.

5.1 The `UnderLets` Monad

One additional feature of the rewriter is support for `let`-lifting: we lift `let ⋯ in ⋯`s to top level, so that applications of functions to `let ⋯ in ⋯`s are available for rewriting. For example, we can perform the rewriting

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

using the rules

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

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 := 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, placing the application of their `Base` expressions underneath.

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 := var) (expr (var := var) t)
          else expr (var := 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 intermediate list, since such a list might be reached multiple times in different ways while evaluating the `let` body.

5.2 Melding with Abstract-Interpretation Outputs

It is natural to phrase rewrite rules in terms of side conditions. For instance, our motivating setting of 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 $?n + 0 \to n$. When we get to reducing terms that have been specialized to fixed integer widths, we must be a bit more legalistic:

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

An intuitive operationalization of such rules would be to build evaluation of side conditions into execution of rewrite rules from section 2. However, even if side conditions look manifestly executable, we would get stuck when they contain free variables, as `n` would often be in uses of this rule. The pragmatic choice we made proceeds in two parts, the second of which is able to invoke our reduction engine with rules whose side conditions always resolve to closed expressions.

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 $l \leq n < u$. Now, before we begin partial evaluation, we can run a verified abstract interpreter to find conservative bounds for each program variable. When bounds l and u are found for variable x, it is sound to replace x with $\texttt{clip}_{l,u}(x)$. Therefore, at the end of this phase, we assume all variable occurrences have been rewritten in this manner to record their proved bounds.

Second, we proceed with our example rule refactored:

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

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. Our actual pattern language records which variables are required to be literals, and only these are allowed to be mentioned in side conditions, which must themselves be object-language expressions of Boolean type.

5.3 Type Variables and Polymorphism

We want to handle rewrite rules not only like $\texttt{fst}_{\mathbb{Z},\mathbb{Z}}(?x, ?y) \to x$ but also like $\texttt{fst}_{?A,?B}(?x, ?y) \to x$. To that end, we incorporate a notion of *type variables* into patterns. Type variables are stripped before compiling patterns into decision trees, and they are only unified when a particular rule is chosen for rewriting. (We found that the overhead of juggling dependent-typing details was not justified for this corner of our engine.)

We represent type variables with positive numbers, and type unification returns a map from positives to reified types. In the unification of terms with patterns, function $\mathcal{U}_p$ of subsubsection 2.1.4 needs to be typed dependently over those maps.

6 COQ ENCODING CHALLENGES IN DETAIL

6.1 Continuation-Passing Style

There are two challenges that we have mostly elided so far: being able to pre-evaluate the rewriter, and the interleaving of rewrite-rule replacement with normalization by evaluation.

6.1.1 Interleaving Evaluation and Rewriting. Consider the example $\texttt{map}\ (\lambda x.\ y + x)\ [0; 1; 2] \rightsquigarrow [y; y + 1; y + 2]$. In order to evaluate the rule $?y + 0 \to y$ after applying the λ to elements of the list, we store the do-the-rewriting thunk in the NbE-value term for $(\lambda x.\ y + x)$. In order to apply , we must store such value-thunks in the expressions themselves. Hence the expr type is insufficient. In subsection 3.2, therefore, the following line was oversimplified.

$$\text{reflect}^{\mathcal{R}}_{t_1 \to t_2}(e) = \lambda x.\ \text{reflect}^{\mathcal{R}}_{t_2}(e\ (\text{reify}_{t_1}(x)))$$

The use of reify here is wrong, as it would prevent any further rewriting opportunities from being realized. (For instance, imagine that x is the function argument passed to map, with redices inside enabled when particular arguments are substituted.) Instead, we use a delayed sort of reification that stores the existing value-thunk in a constructor of an inductive type, and only performs reification if we require an expr at this location. Additionally, *only if* t_1 is itself an arrow type, we record the fact that there can be no expression structure here to inspect for further pattern matching. If t_1 is a base type, then reification is a no-op, and we permit further pattern matching on the expression structure. This distinction is needed to support, for example, both the map rewrite rules used in the above simultaneously with the rule $?y + 0 \to y$ which requires inspecting the form of the second argument to +.

6.1.2 Revealing "Enough" Structure. The inductive type introduced above pulls triple duty. It allows storing thunked values in expressions. It is an inductive syntax tree that is not dependently typed over a type of type codes, for which reason we name it `rawexpr`. Finally, we use it to track how much structure has been "revealed" via pattern matching. This last duty is important because we pre-evaluate the decision-tree evaluation on the particular decision tree for a given set of rewrite rules, for performance. In order to do so, we work in continuation-passing style so that the evaluation happens underneath pattern matching, and we track which expressions have already been matched on, so that we need not match on them again, duplicating work. Furthermore, we track a couple of other things, such as which identifiers are fully known (and can therefore be matched on before we know what expression we are rewriting from), as well as the unmatched versions of expressions, so that when we reassemble terms in the rewrite-rule replacements, they need not be built from the deepest match we have found.

Finally, in order to pre-evaluate decision-tree evaluation, we $\eta\iota$-expand the identifier type as the first step in the definition of `rewrite-head`. That is, in subsection 3.2, the line

$$\text{reduce}^{\mathcal{R}}(c) = \text{reflect}^{\mathcal{R}}(c)$$

should really be

$$\text{reduce}^{\mathcal{R}}(c) = \eta\iota\text{-expand-cps}(c, \text{reflect}^{\mathcal{R}})$$

The `rawexpr` inductive type is:

```
Inductive rawexpr : Type :=
| rIdent (known : bool) {t} (idc : ident t) {t'} (alt : expr t')
| rApp (f x : rawexpr) {t} (alt : expr t)
| rExpr {t} (e : expr t)
| rNbeT {t} (e : nbeT t).
```

Note how nodes are tagged for easy conversion back to fully syntactic expressions (`expr`s) as needed. Each `alt` argument is an `expr` version of the current node, so nontrivial conversion is only needed for embedded values (`nbeT`s), which can be made syntactic with reify.

A key motivator of our strategy is to support partial evaluation of our partial evaluator, specializing it to a set of constants and rewrite rules, to avoid runtime costs of generality. One good example is precompilation of the constant case of $\text{reduce}^{\mathcal{R}}$, as defined above. We do $\eta\iota$-expansion of identifiers, the first step where identifier types matter. Note that we use $f@x$ to denote `rApp f x (App (expr_of_rawexpr f) (expr_of_rawexpr x))`, where

$$\begin{aligned}
\texttt{expr_of_rawexpr(rIdent _ } e) &= e\\
\texttt{expr_of_rawexpr(rApp _ _ } e) &= e\\
\texttt{expr_of_rawexpr(rExpr } e) &= e\\
\texttt{expr_of_rawexpr(rNbeT } v) &= \text{reify}(v)
\end{aligned}$$

We also use $\#_t(c)$ to denote `rIdent true c (Ident c)`. Then we can trace the evaluation, showing four representative constants:

$$\text{reduce}^{\mathcal{R}}(c) = \eta\iota\text{-expand-cps}(c, \text{reflect}^{\mathcal{R}})$$

$$= \begin{cases} \text{reflect}^{\mathcal{R}}(\#_t((+))) & \text{if } c = (+) \\ \text{reflect}^{\mathcal{R}}(\#_t(\texttt{map})) & \text{if } c = \texttt{map} \\ \text{reflect}^{\mathcal{R}}(\#_t(\texttt{Literal}\ n)) & \text{if } c = \texttt{Literal}\ n \\ \text{reflect}^{\mathcal{R}}(\#_t([])) & \text{if } c = [] \end{cases}$$

$$= \begin{cases} \lambda x\ y.\ \text{rewrite-head}((\#_t((+)))@\texttt{rExpr}\ x@\texttt{rExpr}\ x) & \text{if } c = (+) \\ \lambda f\ \ell.\ \text{rewrite-head}(\#_t(\texttt{map})@\texttt{rNbeT}\ f@\texttt{rExpr}\ \ell) & \text{if } c = \texttt{map} \\ \text{rewrite-head}(\#_t(\texttt{Literal}\ n)) & \text{if } c = \texttt{Literal}\ n \\ \text{rewrite-head}(\#_t([])) & \text{if } c = [] \end{cases}$$

Note the asymmetry between the cases for addition and `map`. For the former, the operands are numbers, which will not reduce differently depending on how they are used later. For the latter, the second operand (the list to map over) gets the same treatment, but the first argument (the function to apply to list elements) is encoded with `rNbeT` instead, because it holds a thunk ready to perform further rewriting and normalization, as it is informed of particular list elements.

The full implementation is complicated by our design choice to make `rawexpr` a type rather than a type family – so we do not have dependent types enforcing that every representable expression is well-typed in the object language. Computations that operate solely on `rawexpr` are pleasingly simplified, but we have to jump through some hoops at interface points between the two representations.

6.2 Dependent Types

6.2.1 Typing rewrite rules. As usual with developments in type-theoretic proof assistants, we faced quite a few design decisions in how heavily to use dependent types. On the one hand, it is a core requirement of our architecture that Coq accept the termination of all functions we write, and types are often essential for making that argument. On the other hand, detailed dependent typing requires explicit casts and other proof-complicating artifacts.

One of the places where we accepted the most complexity was in representation of rewrite rules. Rewrite-rule replacements are dependently indexed over patterns in a nested fashion: over the environment of bound type variables and the types of wildcards (which are typed in terms of the former). A replacement then takes as input a collection of wildcard values from the pattern, which is a dependently typed structure over both the pattern and the collection of type variables from the pattern. Hence rewrite rule-replacements are triply dependently typed, and proving anything about them is rather involved. It may be especially worth tinkering here with the strategy of what to type dependently and what not.

6.2.2 Rewriting again in the output of a rewrite rule. By building on NbE, we inherit a satisfying strategy for full normalization according to standard λ-calculus rules. Integration of custom rewrite rules poses a trickier termination problem. We do not attempt to prove that our partial evaluator avoids any lingering opportunities for rewriting. Instead, each rewrite rule is tagged with expectations on whether or not additional rounds of rewriting should happen after it is applied. The entire framework is parameterized over an amount of "fuel", which is the maximum number of consecutive times we'll need to rewrite again in the output of a rewrite rule. By setting the fuel to the length of the full list of rewrite rules, we can guarantee support for any properly annotated list of rewrite rules where the lingering opportunities form a DAG, though we do not prove this fact.

A good example is the rule for the `flat_map` function applied to a list of known shape, which can be rewritten into repeated list-append of the function results on the list elements. When those appended lists themselves have known top-level structure (after more λ-calculus-style reduction), we want to see the defining equations of list-append applied thereafter.

Our key lever for continued execution is the `var` type family of PHOAS syntax trees. Recall that, to generate reduced syntax trees with variable type `var`, our final reducer used a helper function producing values of type `nbeT (var := var) t`, given terms of type `expr (var := nbeT (var := var)) t` as input. To support an additional round of rewriting afterward, we require different types for the functions producing rewrite replacement expressions depending on whether the framework is asked to rewrite again in output of a rewrite rule. If no additional rewriting is needed, a replacement rule takes in one `nbeT (var := var) t` for each wildcard and produces an `expr (var := var) t` as output. If additional rewriting is needed, however, a replacement rules still takes in one `nbeT (var := var) t` for each wildcard, but now it must produce a `expr (var := nbeT (var := var)) t` as output. This allows the framework to reduce underneath any new λs introduced in the output of the rewrite rule before passing the expression back to the pattern matcher for another round of rewriting. Reification of rewrite rules automatically handles selecting the correct `var` type based on the annotation of the rewrite rule; any uses of terms bound to wildcards are wrapped in `Var` nodes.

6.2.3 Applying a primitive-recursive definition fully. It is possible to give all clauses of a primitive-recursive function definition as rewrite rules, using the previously described mechanism with enough fuel to allow repeated rewriting on the recursive occurrences in the outputs of rules. However, we include special support for rewrites that are designated as following this pattern, where we should reduce fully with a "fold" operation when the recursive argument has known top-level structure (e.g., for a list, it is a sequence of "cons" operations ending in a "nil"). Rewrite rules may require that certain pattern variables match terms with known top-level structure, in which case their replacement expressions may use special eager "fold" operations. The rewriter then has special cases for these "eager" identifiers, which apply corresponding "fold" operations in the metalanguage.

6.3 Well-Formedness Predicates

One of the subtleties of PHOAS is that, while the encoding saves us from variable bookkeeping in writing code transformations, modeling of explicit environments tends to reappear in proofs. Specifically, as an encoded term is a polymorphic function waiting to be told how to represent variables, we often find ourselves dealing in one proof with multiple different instantiations of the same polymorphic term. We must know that the term was *well-formed* to begin with, meaning that structurally isomorphic syntax trees emerge, regardless of how we represent variables. An *equivalence* predicate is key to this method, where we say a term is well-formed if any two of its instantiations are equivalent.

An equivalence judgment takes the form $\Gamma \vdash e_1 \sim e_2$, asserting that syntax trees e_1 and e_2 (usually with different variable types) are equivalent, up to assumptions Γ on which variables (usually of the two different types) are equivalent. The core rules are as follows.

$$\frac{(x_1, x_2) \in \Gamma}{\Gamma \vdash x_1 \sim x_2} \qquad \frac{\Gamma \vdash f_1 \sim f_2 \qquad \Gamma \vdash a_1 \sim a_2}{\Gamma \vdash f_1\ a_1 \sim f_2\ a_2} \qquad \frac{\forall x_1, x_2.\ \Gamma, (x_1, x_2) \vdash e_1(x_1) \sim e_2(x_2)}{\Gamma \vdash \lambda e_1 \sim \lambda e_2}$$

Correctness arguments for NbE typically use logical relations (relations defined recursively on type structure, with a distinctive shape of the function-arrow case), and ours is no exception.

However, this relation needs to interact with PHOAS well-formedness, since we represent base-type values with PHOAS syntax. We must assert Kripke-style that such values are well-formed in *any variable context that extends the current one*. Here is the relation, parameterized both over under_lets, saying whether this term has let-lifting enabled; and G, the environment of PHOAS free variables accumulated up to this point.

```
Fixpoint wf_nbeT' {with_lets : bool} G {t : type}
  : nbeT' (var := var1) with_lets t -> nbeT' (var := var2) with_lets t -> Prop
  := match t, with_lets with
     | type.base t, true => UnderLets.wf expr.wf G
     | type.base t, false => expr.wf G
     | type.arrow s d, _
       => fun f1 f2
          => (forall seg G' v1 v2,
                 G' = (seg ++ G)%list
                 -> @wf_nbeT' false seg s v1 v2
                 -> @wf_nbeT' true G' d (f1 v1) (f2 v2))
     end.
```

Note the classic sort of function-arrow case, saying that related arguments are mapped to related function outputs. Where we encountered some subtlety was in stating exactly how the variable environment may be extended. For instance, we found that using list inclusion rather than list concatenation led to a relation too weak for some uses and too strong for others. The key use of this flexibility is justifying lifting of additional let binders above an expression.

The “too strong in some places and too weak in others” is a common theme that we encountered when trying to construct definitions. We encountered it again in trying to specify well-formed-relatedness of rawexprs. We eventually settled on a four-place relation involving two rawexprs and two exprs. Roughly, it encodes the fact that each rawexpr is the result of revealing the same amount of structure of the corresponding expr, and also that the exprs at the leaves are wf-related.

```
Inductive wf_rawexpr : list { t : type & var1 t * var2 t }%type
  -> forall {t}, rawexpr (var := var1) -> expr (var := var1) t
  -> rawexpr (var := var2) -> expr (var := var2) t -> Prop :=
| Wf_rIdent {t} G known (idc : ident t)
  : wf_rawexpr G (rIdent known idc (expr.Ident idc)) (expr.Ident idc)
                 (rIdent known idc (expr.Ident idc)) (expr.Ident idc)
| Wf_rApp {s d} G
        f1 (f1e : expr (var := var1) (s -> d)) x1 (x1e : expr (var := var1) s)
        f2 (f2e : expr (var := var2) (s -> d)) x2 (x2e : expr (var := var2) s)
  : wf_rawexpr G f1 f1e f2 f2e
    -> wf_rawexpr G x1 x1e x2 x2e
    -> wf_rawexpr G
                  (rApp f1 x1 (expr.App f1e x1e)) (expr.App f1e x1e)
                  (rApp f2 x2 (expr.App f2e x2e)) (expr.App f2e x2e)
| Wf_rExpr {t} G (e1 e2 : expr t)
  : expr.wf G e1 e2 -> wf_rawexpr G (rExpr e1) e1 (rExpr e2) e2
| Wf_rNbeT {t} G (v1 v2 : nbeT t)
  : wf_nbeT G v1 v2
    -> wf_rawexpr G (rNbeT v1) (reify v1) (rNbeT v2) (reify v2).
```

We also ran into some complications due to the fact that rawexprs are untyped, but their types can be extracted. For example, we can pull out that the inferred types of two rawexprs are equal from a proof that the rawexprs are jointly well-formed. We needed to transport various terms across this equality proof and then be able to eliminate the transports later on.

6.4 Interpretation-Relatedness Predicates and Complications Thereof

Consider the rule `flat_map` $?f\ (?x :: ?xs) \to_{\mathcal{R}} f\ x$ ++ `flat_map` $f\ xs$, which contains an opportunity for rewriting again. If $f\ x$ is a concrete list of cons cells, then we can reduce the list concatenation into cons cells. As discussed in subsubsection 6.2.2, in order for the types to work out, this rule needs to return a PHOAS `expr` with a `var` type of `nbeT (var := var)`, providing a kind of type-level "fuel" to run another round of simplification. To express the correctness of such a rewrite rule, we need to relate a such an `expr` to an interpreted denotation. We cannot simply interpret the `expr` to express its relatedness, because the `Var` nodes contain values that may be thunked rewriting functions, and we must express correctness for *any* valid *expression* inputs to those rewriting thunks. Hence we need a recursive notion of interpretation-relatedness which, when it hits `Var` nodes, uses a notion of `nbeT`-interpretation-relatedness (which itself bottoms out with `expr`-interpretation-relatedness).

7 IMPLEMENTATION AND EVALUATION

Our implementation is a Coq library parameterized over the types, constants, and rewrite rules that should determine a partial evaluator. The core evaluator is proved correct once and for all, subject to natural correctness conditions on the rewrite rules, assuming only the common axiom of functional extensionality. With the parameters known, we can partially evaluate the partial evaluator, using more standard reduction methods, where performance is not as crucial. For instance, in our application to Fiat Cryptography, we build the partial evaluator once and apply it to generate hundreds of different algorithm variants. That generation can happen either via reducing the specialized partial evaluator in Coq or by running a version of it extracted to OCaml or Haskell. The second mode is particularly appealing for integration with real-world software projects; e.g., now Fiat Cryptography partial evaluation can be integrated into Google's build processes for Chrome, where before Coq was run offline to generate C files to check into a repository. (Here we realize dual benefits of dramatically improved partial-evaluation performance and a switch away from asking engineers to run an arcane formal-methods tool directly.)

What performance improvements did we find? Figure 3 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 reduction within Coq. 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 10X 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 primes as in the experiments run by Erbsen et al. [2019], which were chosen based on searching the archives of an elliptic-curves mailing list for all prime numbers.) 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

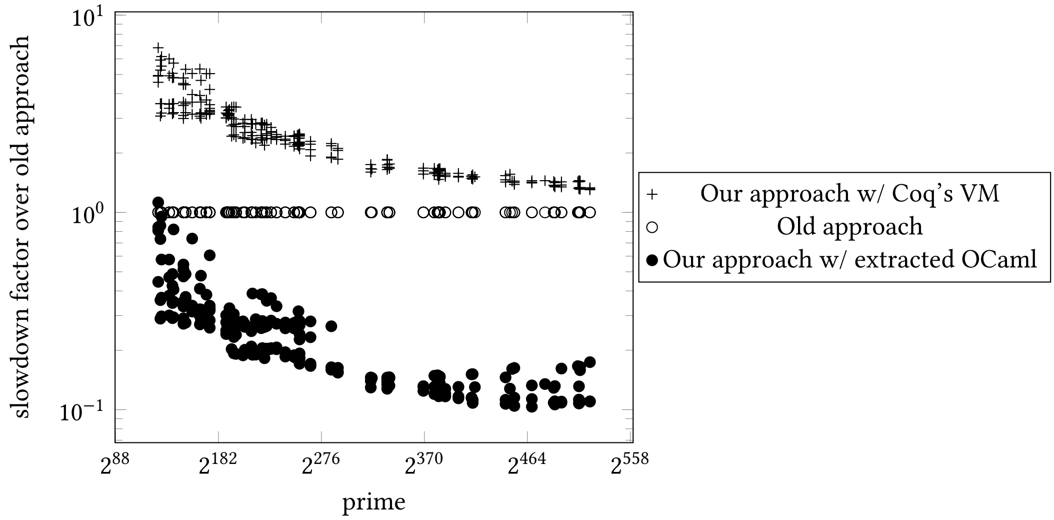


Fig. 3. Scaling of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows

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 shows the clear (and probably not surprising) performance advantage of running reduction in extracted code. By the time we reach middle-of-the-pack prime size around 300 bits, the extracted version is running about 10X as quickly as the baseline.

8 RELATED WORK

Aehlig et al. [2008] showed how to integrate NbE and algebraic rewriting in a proof assistant, with a verified translation from object-language syntax to a deep embedding of a small ML subset. In contrast to our use of a generator for code implementing rewrite rules, their engine is left parameterized over an ML function to implement rewrite rules. They avoid proving termination but do prove that any terminating run of the rewriter yields a term in normal form with no more rewrite opportunities, while we do the opposite, proving termination but not normal forms. Lack of termination proofs poses a challenge for integration in a proof assistant without expanding the trusted base. Indeed, the Isabelle/HOL tactic that they built required trusting a freestanding ML implementation and the code to extract to it from deeply embedded syntax. Their core translation uses de Bruijn indices and represents closures explicitly, missing out on an opportunity to delegate these details to the proof-assistant kernel. Performance experiments were reported, but only on examples where all inputs are known statically, leaving open the empirical question of suitability for partial evaluation.

Our implementation builds on fast full reduction in Coq's kernel, via a virtual machine [Grégoire and Leroy 2002] or compilation to native code [Boespflug et al. 2011]. Especially the latter is similar in adopting an NbE style for full reduction, simplifying even under λs, on top of a more traditional implementation of OCaml that never executes preemptively under λs. Neither approach unifies support for rewriting with proved rules, and partial evaluation only applies in very limited cases, where functions that should not be evaluated at compile time must have properly opaque definitions that the evaluator will not consult. Neither implementation involved a machine-checked proof suitable to bootstrap on top of reduction support in a kernel providing simpler reduction.

A more limited form of code generation for cryptography was already widespread through libraries like OpenSSL, which specifically uses Perl scripts to generate assembly code. The Vale tool suite [Bond et al. 2017] formalizes these practices with a more principled language and associated verification tools. However, code generation done in this style is significantly simpler than what we treat here, amounting mostly to loop unrolling, macro substitution, and computation of compile-time constants. Also, Vale involves a significantly larger trusted code base than with our approach, with no reduction to some kernel proof checker, instead placing trust in language-specific tooling and an SMT solver.

A variety of forms of pragmatic partial evaluation have been demonstrated, with Lightweight Modular Staging [Rompf and Odersky 2010] in Scala as one of the best-known current examples. A kind of type-based overloading for staging annotations is used to smooth the rough edges in writing code that manipulates syntax trees. The LMS-Verify system [Amin and Rompf 2017] can be used for formal verification of generated code after-the-fact. Typically LMS-Verify has been used with relatively shallow properties (though potentially applied to larger and more sophisticated code bases than we tackle), not scaling to the kinds of functional-correctness properties that concern us here, justifying investment in verified partial evaluators.

REFERENCES

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

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

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

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

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

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

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/

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/

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

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

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.

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

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

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

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