# A Performance Bottlenecks of Proof-Producing Rewriting

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

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

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

We posit that this overhead comes from `setoid_rewrite` looking for head-symbol matches and then creating evars (existential variables) to instantiate the arguments of the lemmas for each head-symbol-match location; hence even if there are no matches of the rule as a whole, there may still be head-symbol matches. Since Coq uses a locally nameless representation [3] for its terms, evar contexts are necessarily represented as *named* contexts. Representing a substitution between named contexts takes linear space, even when the substitution is trivial, and hence each evar incurs overhead linear in the number of binders above it. Furthermore, fresh-name generation in Coq is quadratic in the size of the context, and since evar-context creation uses fresh-name generation, the additional multiplicative factor likely comes from fresh-name generation. (Note, though, that this pattern suggests that the true performance is quartic rather than merely cubic. However, doing a linear regression on a log-log of the data suggests that the performance is genuinely cubic rather than quartic.)

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

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

## A.2 Proof-term size

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

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

## A.3 Poor subterm sharing

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

## A.4 Overhead from the `let` typing rule

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

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

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

## A.5 Inherent advantages of reflection

Finally, even if this quadratic bottleneck were fixed, Aehlig et al. [1] reported a 10×–100× speed-up over the *simp* tactic in Isabelle, which performs all of the intermediate rewriting steps via the kernel API. Their results suggest that even if all of the superlinear bottlenecks were fixed—no small undertaking—rewriting and partial evaluation via reflection might still be orders of magnitude faster than any proof-term-generating tactic.

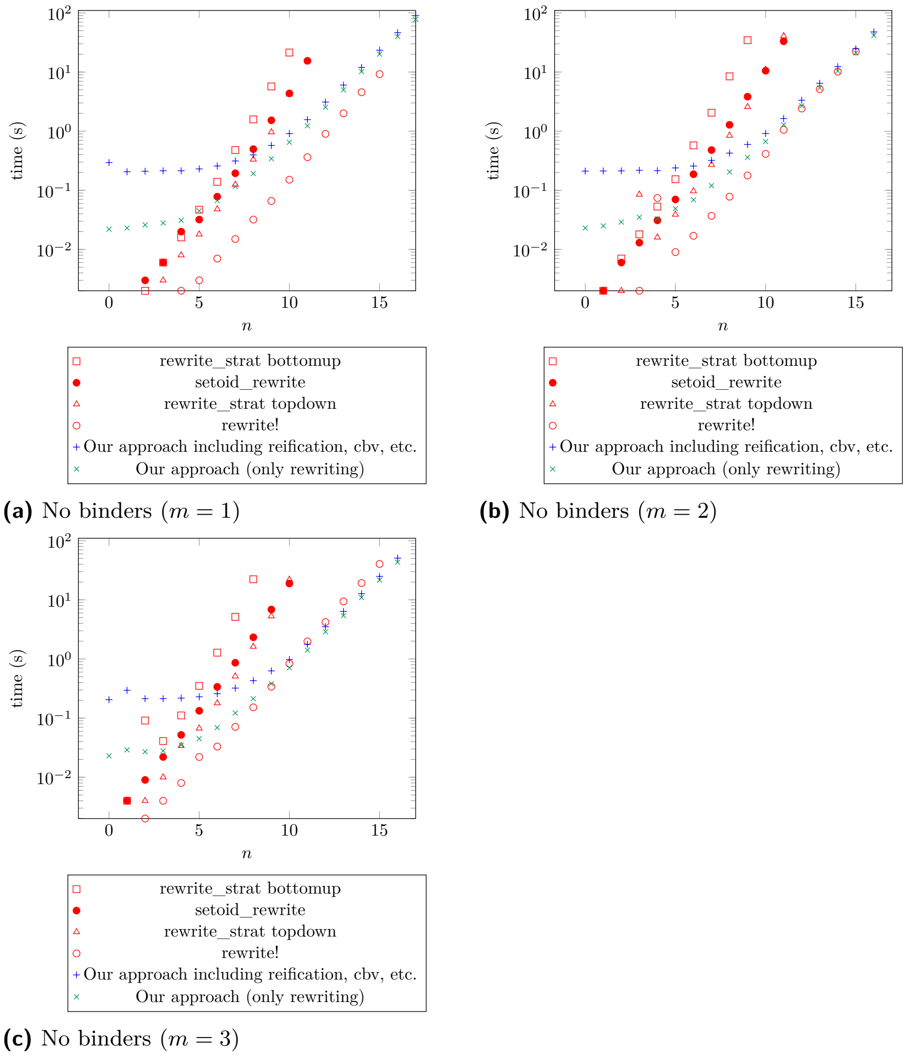


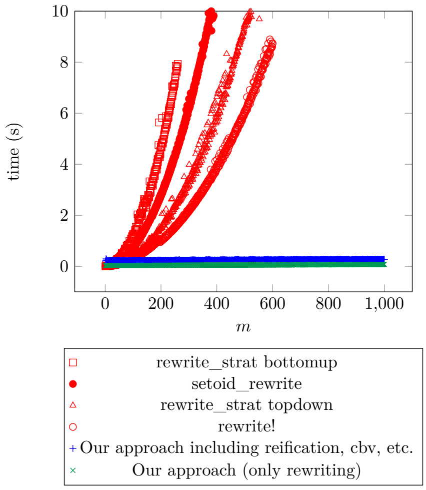

**Figure 4** Timing of different partial-evaluation implementations for code with no binders for fixed $m$. Note that we have a logarithmic time scale, because term size is proportional to $2^n$.

# B Additional Benchmarking Plots

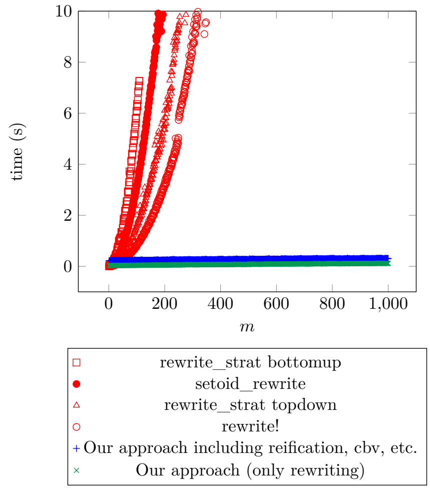
## B.1 Rewriting Without Binders

The code in Figure 7a in Appendix C.1 is parameterized on both $n$, the height of the tree, and $m$, the number of rewriting occurrences per node. The plot in Figure 3a displays only the case of $n = 3$. The plots in Figure 4 display how performance scales as a factor of $n$ for fixed $m$, and the plots in Figure 5 display how performance scales as a factor of $m$ for fixed $n$. Note the logarithmic scaling on the time axis in the plots in Figure 4, as term size is proportional to $m \cdot 2^n$.

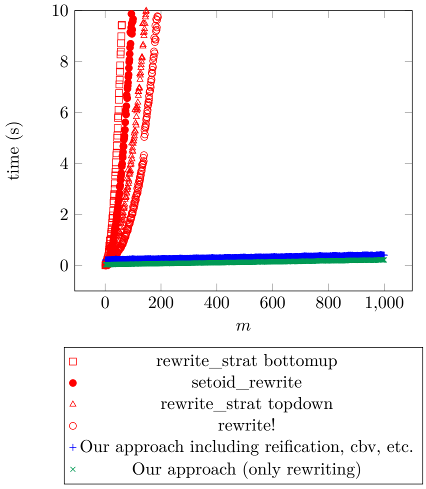
We can see from these graphs and the ones in Figure 5 that (a) we incur constant overhead over most of the other methods, which dominates on small examples; (b) when the term is quite large and there are few opportunities for rewriting relative to the term size (i.e., $m \leq 2$), we are worse than `rewrite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



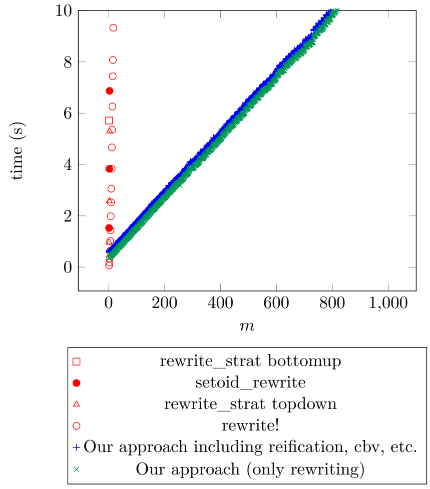

**(a)** No binders ($n = 1$)




**(b)** No binders ($n = 2$)




**(c)** No binders ($n = 3$)



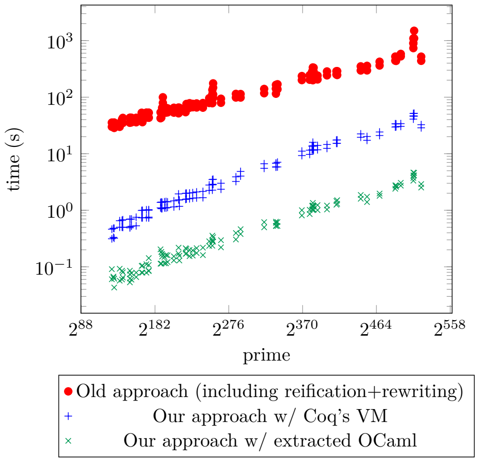

**(d)** No binders ($n = 9$)

**Figure 5** Timing of different partial-evaluation implementations for code with no binders for fixed $n$ (1, 2, 3, and then we jump to 9) and (c) when there are many opportunities for rewriting relative to the term size ($m > 2$), we thoroughly dominate the other methods.

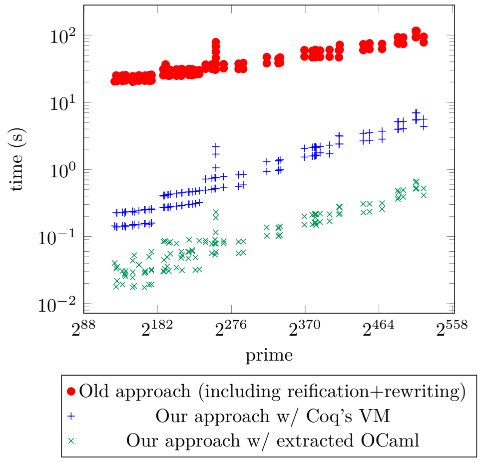
## B.2 Additional Information on the Fiat Cryptography Benchmark

The data for this benchmark can be found on GitHub at `mit-plv/fiat-crypto@perf-testing-data-ITP-2022-rewriting`.

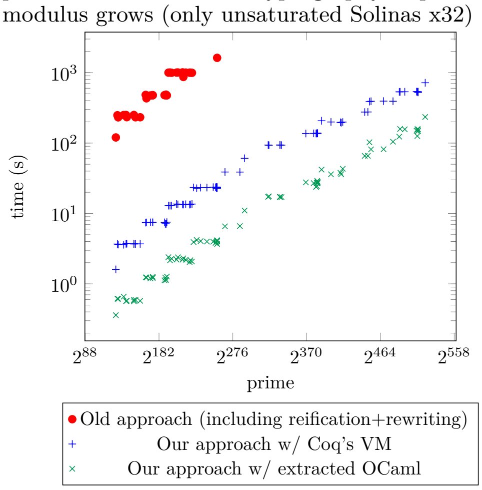
It may also be useful to see performance results with absolute times, rather than normalized execution ratios vs. the original Fiat Cryptography implementation. Furthermore, the benchmarks fit into four quite different groupings: elements of the cross product of two algorithms (unsaturated Solinas and word-by-word Montgomery) and bitwidths of target architectures (32-bit or 64-bit). Here we provide absolute-time graphs by grouping in Figure 6.



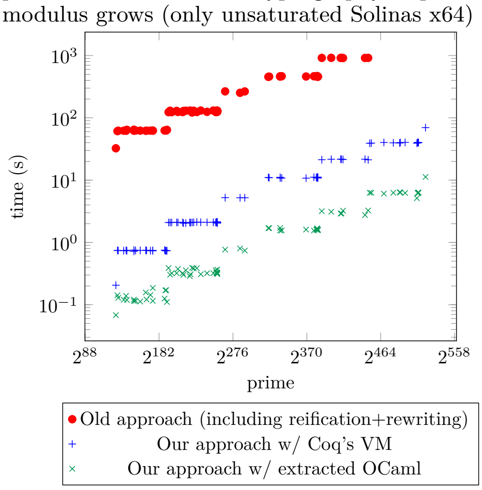

**(a)** Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only unsaturated Solinas x32)




**(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 6** Timing of different partial-evaluation implementations for Fiat Cryptography vs. prime modulus

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

**(a)** Expressions computing initial code for Rewriting Without Binders

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

**(b)** Initial code for Rewriting Under Binders

**Figure 7** Code for rewriting without and under binders

# C Additional Information on Microbenchmarks

We performed all benchmarks on a 3.5 GHz Intel Haswell running Linux and Coq 8.11.1. We name the subsections here with the names that show up in the code supplement.

## C.1 Rewriting Without Binders: Plus0Tree

Consider the code defined by the expression $\mathrm{tree}_{n,m}(v)$ in Figure 7a. We want to remove all of the $+\,0$s. There are $\Theta(m \cdot 2^n)$ such rewriting locations. We can start from this expression directly, in which case reification alone takes as much time as Coq's `rewrite`. As the reification method was not especially optimized, and there exist fast reification methods [10], we instead start from a call to a recursive function that generates such an expression.

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).
```

## C.2 Rewriting Under Binders: UnderLetsPlus0

Consider now the code in Figure 7b, which is a version of the code above where redundant expressions are shared via `let` bindings.

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 nondependent versions. The tactic that does this preprocessing is extensible via $\mathcal{L}_{tac}$'s reassignment feature. Since pattern-matching compilation mixed with NbE requires knowing how many arguments a constant can be applied to, 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.

### C.2.1 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 Section 2 that `eval_rect` is a definition provided by our framework for eagerly evaluating recursion associated with certain types. It functions by triggering typeclass resolution for the lemmas reducing the recursion principle associated to the given type. We provide instances for `nat`, `prod`, `list`, `option`, and `bool`. Users may add more instances if they desire.

### C.2.2 `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`.

## C.3 Binders and Recursive Functions: LiftLetsMap

The next experiment uses the code in Figure 8. Note that the `let` $\cdots$ `in` $\cdots$ binding blocks further reduction of map_dbl when we iterate it $m$ times in make, and so we need to take care to preserve sharing when reducing here.

Figure 9 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

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

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

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

**Figure 8** Initial code for binders and recursive functions

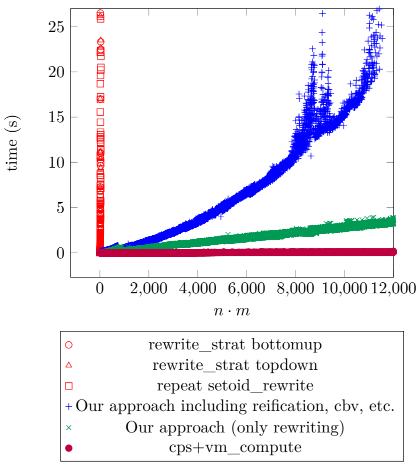



**Figure 9** Benchmark with recursive functions

applied to the original, more readable definitions loses ground steadily to VM-based reduction on CPS'd code. On the largest terms ($n \cdot m > 20,000$), the gap is 6s vs. 0.1s of compilation time, which should often be acceptable in return for simplified coding and proofs, plus the ability to mix proved rewrite rules with built-in reductions. Note that about 99% of the difference between the full time of our method and just the rewriting is spent in the final **cbv** at the end, used to denote our output term from reified syntax. We blame this performance on the unfortunate fact that reduction in Coq is quadratic in the number of nested binders present; see Coq bug #11151. This bug has since been fixed, as of Coq 8.14; see Coq PR #13537.

We can perform this rewriting in four ways.

### C.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 4.3. Recall again from Section 2 that we use `ident.eagerly` to ask the reducer to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree. Our current version only allows a limited, hard-coded set of eliminators with `ident.eagerly` (`nat_rect` on return types with either zero or one arrows, `list_rect` on return types with either zero or one arrows, and `List.nth_default`), but nothing in principle prevents automatic generation of the necessary code.

We 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.

### C.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 $\zeta$. 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 prereified 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.

### C.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

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

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

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

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.

### C.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 opacity 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.

## C.4 SieveOfEratosthenes

The final experiment involves full reduction in computing the Sieve of Eratosthenes, taking inspiration on benchmark choice from Aehlig et al. [1]. We find in Figure 10 that we are

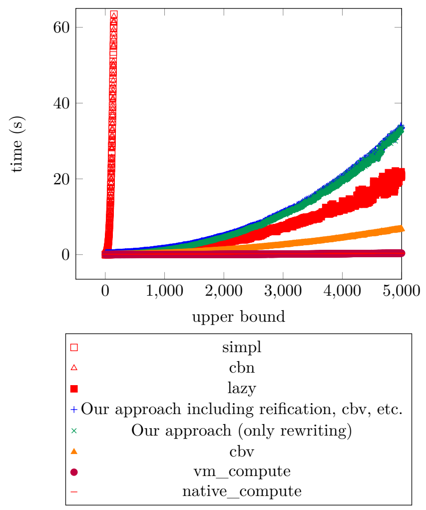



**Figure 10** Full evaluation, Sieve of Eratosthenes

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 !_ / .
```

# D Fusing Compiler Passes

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

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

Another essential pass to fuse with rewriting and partial evaluation is let-lifting. Unless all of the code is CPS-converted ahead of time, attempting to do let-lifting via rewriting, as must be done when using **`setoid_rewrite`**, **`rewrite_strat`**, or $\mathcal{R}_{tac}$, results in slower asymptotics. This pattern is already apparent in the `LiftLetsMap` / “Binders and Recursive Functions” example in Appendix C.3. We achieve linear performance in $n \cdot m$ when ignoring the final **`cbv`**, while **`setoid_rewrite`** and **`rewrite_strat`** are both cubic. The rewriter in $\mathcal{R}_{tac}$ cannot possibly achieve better than $\mathcal{O}\left(n \cdot m^2\right)$ unless it can be sublinear in the number of rewrites, because our rewriter gets away with a constant number of rewrites (four), plus evaluating recursion principles for a total amount of work $\mathcal{O}(n \cdot m)$. But without primitive support for let-lifting, it is instead necessary to lift the lets by rewrite rules, which requires $\mathcal{O}\left(n \cdot m^2\right)$ rewrites just to lift the lets. The analysis is thus: running **`make`** simply gives us $m$ nested applications of `map_dbl` to a length-$n$ list. To reduce a given call to `map_dbl`, all existing let-binders must first be lifted (there are $n \cdot k$ of them on the $k$-innermost-call) across `map_dbl`, one-at-a-time. Then the `map_dbl` adds another $n$ let binders, so we end up doing $\sum_{k=0}^{m} n \cdot k$ lifts, i.e., $n \cdot m(m + 1)/2$ rewrites just to lift the lets.

# E 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 `setoid_rewrite` for partial evaluation and rewriting on unsaturated Solinas with 1 limb on small primes (such as $2^{61} - 1$), we were able to get `setoid_rewrite` to finish after about 100 seconds. Trying to synthesize code for two limbs on slightly larger primes (such as $2^{107} - 1$, which needs two limbs on a 64-bit machine) took about 10 minutes; three limbs took just under 3.5 hours, and four limbs failed to synthesize with an out-of-memory error after using over 60 GB of RAM. The widely used primes tend to have around five to ten limbs. See #13576 for more details and for updates.

The `rewrite_strat` tactic, which does not require duplicating the entire goal at each rewriting step, fared a bit better. Small primes with 1 limb took about 90 seconds, but further performance tuning of the typeclass instances dropped this time down to 11 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. Two limbs took about 90 seconds, three limbs took a bit under 10 minutes, and four limbs took about 70 minutes and about 17 GB of RAM. Extrapolating out the exponential asymptotics of the fastest-growing subcall to `rewrite_strat` indicates that 5 limbs would take 11–12 hours, 6 limbs would take 10–11 days, 7 limbs would take 31–32 weeks, 8 limbs would take 13–14 years, 9 limbs would take 2–3 centuries, 10 limbs would take 6–7 millennia, and 15 limbs would take 2–3 times the age of the universe, and 17 limbs, the largest example we might find at present in the real world, would take over 1000× the age of the universe! See #13708 for more details and updates.

This experiment using `rewrite_strat` can be found online in the Coq source file at `src/fiat_crypto_via_setoid_rewrite_standalone.v` on GitHub at `coq-community/coq-performance-tests`. To test with the two-limb prime $2^{107} - 1$, change `Goal goal` to `Goal goal_of_size 2%nat` near the bottom of the file.

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's `setoid_rewrite` 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.

Our experiments with running `rewrite` in Lean on the Fiat Cryptography code can be found in the file `fiat-crypto-lean/src/fiat_crypto.lean` on GitHub at `mit-plv/fiat-crypto@lean`. 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.

## F Limitations and Preprocessing

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

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

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

We define a constant `Let_In` which we use for writing `let ⋯ in ⋯` expressions that do not reduce under $\zeta$ (Coq's reduction rule for **let**-inlining). Throughout most of this paper, anywhere that `let ⋯ in ⋯` appears, we have actually used `Let_In` in the code. It would alternatively be possible to extend the reification preprocessor to automatically convert `let ⋯ in ⋯` to `Let_In`, but this strategy may cause problems when converting the interpretation of the reified term with the prereified term, as Coq's conversion does not allow fine-tuning of when to inline or unfold **let**s.

## G 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 versions 8.9–8.13, 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.

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.

## G.1 Code from Section 1, Introduction

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

### G.1.1 Code from Subsection 1.1, Related Work

There is no code mentioned in this section.

### G.1.2 Code from Subsection 1.2, 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 code base**. 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. `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, $\beta$ reduction, and we assert that it should be self-evident that our rewriter supports this.
- We claimed to allow **rapid iteration** on rewrite rules with *minimal verification overhead.* We invite the reader to alter the list of constants in any of the `Make ... := Rewriter For ...` invocations in `rewriter/src/Rewriter/Rewriter/Examples/` or to alter the list of rewrite rules in `fiat-crypto/src/Rewriter/Rules.v` to experience iteration on rewrite rules.
- 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 files `perf_unsaturated_solinas.v` and `perf_word_by_word_montgomery.v`, and the 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.

We mention encoding pattern matching explicitly by adopting the performance-tuned approach of Maranget [17]; the code for this is in `rewriter/src/Rewriter/Rewriter/Rewriter.v` starting from the comment above `Inductive decision_tree` and including the Gallina definitions `eval_decision_tree` and `compile_rewrites`.

We mention integration with abstract interpretation; the abstract-interpretation pass is implemented in `fiat-crypto/src/AbstractInterpretation/`; integration is achieved in rewrite rules in `fiat-crypto/src/Rewriter/Rules.v` making use of the various `Local Notation`s defined in that file for `ident.cast`.

We mention parametric higher-order abstract syntax (PHOAS); the definition of our datatype is `Inductive expr` in module `Compilers.expr` in `rewriter/src/Rewriter/Language/Language.v`. We mention a let-lifting transformation threaded throughout reduction; this is `Inductive UnderLets`, a monad defined in module `Compilers.UnderLets` in the file `rewriter/src/Rewriter/Language/UnderLets.v`.

### G.2 Code from Section 2, 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 $\iota$ reduction, which much of Coq's infrastructure performs eagerly. Because we do not depend on the axiom of functional extensionality, we also in practice require `Proper` instances for each higher-order identifier saying that each constant respects function extensionality. 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 `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v` and defined in `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`.

### G.3 Code from Section 3, The Structure of a Rewriter

#### G.3.1 Code from Subsection 3.1, Our Approach in Ten 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 $\mathcal{L}_{tac}$ tactic named `make_scrape_data` in a submodule in the source file `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 the constant `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 the tactic

`make_rewriter_of_scraped_and_ind` in the source file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml` which invokes the tactic `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 $\mathcal{L}_{tac}$ tactics `build_package` in `rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v` and also in `rewriter/src/Rewriter/Language/IdentifiersGenerate.v` (there is a different tactic named `build_package` in each of these files), and `prove_package_proofs_via` which can be found in `rewriter/src/Rewriter/Language/IdentifiersGenerateProofs.v`.

4. The statements of rewrite rules are reified and soundness and syntactic-well-formedness lemmas are proven about each of them. This is done 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 the syntactic-well-formedness proofs 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 closed 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`.

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

## Our Approach in More Than Nine Steps

As the nine steps of Subsection 3.1 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 3.1 or Appendix G.3.1.

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 code 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.

### G.3.2 Code from Subsection 3.2, Pattern-Matching Compilation and Evaluation

The pattern-matching 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 here 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 the theorem `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 goes by the name `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 of the lemma `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 $\mathcal{L}_{tac}$ tactic `make_rewrite_head` in `rewriter/src/Rewriter/Rewriter/Reify.v`.

### G.3.3 Code from Subsection 3.3, Adding Higher-Order Features

The type $\text{NbE}_t$ 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`.

## G.4 Code from Section 4, Scaling Challenges

### G.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 proofs are `InterpRewrite` in the Coq source file `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 Appendix G.3.1.

### G.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.

### G.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`.

### G.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`, as well as `strip_literal_casts_rewrite_rulesT`, as well as `fancy_with_casts_rewrite_rulesT`, and finally as well as `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`.

## G.5 Code from Section 5, Evaluation

### G.5.1 Code from Subsection 5.1, Iteration on the Fiat Cryptography Compiler

The old continuation-passing-style versions of verified arithmetic functions can be found in the folder `fiat-crypto/src/ArithmeticCPS/`, while the new versions can be found in the folder `fiat-crypto/src/Arithmetic/`.

The rewrite rules for reassociating arithmetic can be found in `arith_rewrite_rulesT` starting at the comment "We reassociate some multiplication of small constants" in `fiat-crypto/src/Rewriter/Rules.v`.

The following frontend constructs are in `all_ident_named_interped` defined in `fiat-crypto/src/Language/IdentifierParameters.v`.

- The multiplication primitives are `with_name ident_Z_mul_split Z.mul_split` as well as `with_name ident_Z_mul_high Z.mul_high`, as well as the various Coq expressions `with_name ident_fancy_mulXX ident.fancy.mulXX` for each `X` being either `l` or `h`.
- The "comment" function is both `with_name ident_comment (@ident.comment)` as well as `with_name ident_comment_no_keep (@ident.comment_no_keep)`.
- The bitwise exclusive-or is `with_name ident_Z_lxor Z.lxor`.
- The special identity function which prints in the backend as a call to some inline assembly is `with_name ident_value_barrier (@Z.value_barrier)`.

The rules about bitmasking operations can be found in `arith_with_casts_rewrite_rulesT` in `fiat-crypto/src/Rewriter/Rules.v` and involve `Z.land` and `Z.lor`.

The compiler configuration about conditional-move instructions is the flag `-cmovznz-by-mul` defined in `fiat-crypto/src/CLI.v`. The `if`-statement using the thus-defined `use_mul_for_cmovznz` is in `src/PushButtonSynthesis/Primitives.v`.

The rewrite rules for the new backends are defined by `fancy_with_casts_rewrite_rulesT` and `mul_split_rewrite_rulesT` as well as `multiret_split_rewrite_rulesT` as well as `noselect_rewrite_rulesT` in `fiat-crypto/src/Rewriter/Rules.v`. The special function `Z.combine_at_bitwidth` is defined in `fiat-crypto/src/Util/ZUtil/Definitions.v`. The designation of `Z.combine_at_bitwidth` as an identifier that should be inlined occurs by listing it in the definition `var_like_idents` in the source file `fiat-crypto/src/Language/IdentifierParameters.v`.

The rules involving carries mentioned in Appendix D, Fusing Compiler Passes are in `arith_with_casts_rewrite_rulesT` in `fiat-crypto/src/Rewriter/Rules.v`.

### G.5.2 Code from Subsection 5.2, 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 code for Figure 3a from Appendix C.1, Rewriting Without Binders: Plus0Tree can be found in `Plus0Tree.v`.

The code for Figure 3b from Appendix C.2, Rewriting Under Binders: UnderLetsPlus0 can be found in `UnderLetsPlus0.v`.

The code for Figure 9 from Appendix C.3, Binders and Recursive Functions: LiftLetsMap can be found in `LiftLetsMap.v`.

The code for Figure 10 from Appendix C.4, SieveOfEratosthenes can be found in `SieveOfEratosthenes.v`.

### G.5.3 Code from Subsection 5.3, 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 file `fiat-crypto/src/Language/IdentifiersBasicGENERATED.v`, the Coq source file `fiat-crypto/src/Language/IdentifiersGENERATED.v`, and finally also the Coq source file `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.

## G.6 Code from Appendix F, Limitations and Preprocessing

The $\mathcal{L}_{tac}$ hooks for extending the preprocessing of eliminators are `reify_preprocess_extra` and `reify_ident_preprocess_extra` in a submodule of `rewriter/src/Rewriter/Language/PreCommon.v`. These hooks are called by `reify_preprocess` and `reify_ident_preprocess` in a submodule of `rewriter/src/Rewriter/Language/Language.v`. Some recursion lemmas for use with these tactics are defined in the `Thunked` module in `fiat-crypto/src/Language/PreExtra.v`. These tactics are overridden in the file `fiat-crypto/src/Language/IdentifierParameters.v`.

The typeclass associated to `eval_rect` (c.f. Appendix G.2) is `rules_proofs_for_eager_type` defined in `rewriter/src/Rewriter/Language/Pre.v`. The instances we provide by default are defined in a submodule of `src/Rewriter/Language/PreLemmas.v`.

The hard-coding of the eliminators for use with `ident.eagerly` (c.f. Appendix G.2) is done in the tactics `reify_ident_preprocess` and `rewrite_interp_eager` in `rewriter/src/Rewriter/Language/Language.v`, in the inductive type `restricted_ident` and the typeclass `BuildEagerIdentT` in `rewriter/src/Rewriter/Language/Language.v`, and in the $\mathcal{L}_{tac}$ tactic with the name of `handle_reified_rewrite_rules_interp` defined in the file `rewriter/src/Rewriter/Rewriter/ProofsCommonTactics.v`.

The `Let_In` constant is defined in `rewriter/src/Rewriter/Util/LetIn.v`.