

C ADDITIONAL INFORMATION ON MICROBENCHMARKS

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

C.1 UnderLetsPlus0

We provide more detail on the “nested binders” microbenchmark of [subsection 5.1.2](#) displayed in [Figure 4b](#).

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

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

The code used to define this microbenchmark is

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

We note some details of the rewriting framework that were glossed over in the main body of the paper, which are useful for using the code: Although the rewriting framework does not support dependently typed constants, we can automatically preprocess uses of eliminators like `nat_rect` and `list_rect` into non-dependent versions. The tactic that does this preprocessing is extensible via \mathcal{L}_{tac} ’s reassignment feature. Since pattern-matching compilation mixed with NbE requires knowing how many arguments a constant can be applied to, we must internally use a version of the recursion principle whose type arguments do not contain arrows; current preprocessing can handle recursion principles with either no arrows or one arrow in the motive. Even though we will eventually plug in 0 for v , we jump through some extra hoops to ensure that our rewriter cannot cheat by rewriting away the $+0$ before reducing the recursion on n .

We can reduce this expression in three ways.

C.1.1 Our Rewriter. One lemma is required for rewriting with our rewriter:

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

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

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

Recall from [subsection 1.1](#) that `eval_rect` is a definition provided by our framework for eagerly evaluating recursion associated with certain types. It functions by triggering typeclass resolution for the lemmas reducing the recursion principle associated to the given type. We provide instances for `nat`, `prod`, `list`, `option`, and `bool`. Users may add more instances if they desire.

C.1.2 setoid_rewrite and rewrite_strat. To give as many advantages as we can to the pre-existing work on rewriting, we pre-reduce the recursion on nats 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.2 Plus0Tree

This is a version of [subsection C.1](#) without any let binders, discussed in [subsection 5.1.1](#) but not displayed in [Figure 4](#).

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.3 LiftLetsMap

We now discuss in more detail the “binders and recursive functions” example from [subsection 5.1.4](#).

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

```
let v1,1 := v + v in
:
let v1,n := v + v in
let v2,1 := v1,1 + v1,1 in
:
let v2,n := v1,n + v1,n in
:
[vm,1, ..., vm,n]
```

Recall that we make this example with the code

Definition map_double (ls : list Z) :=

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

Definition make (n : nat) (m : nat) (v : Z) :=

```
nat_rect _ (List.repeat v n) (λ _ rec, map_double rec) m.
```

We can perform this rewriting in four ways; see [Figure 4c](#).

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 1.1](#). Recall again from [subsection 1.1](#) that we use `ident.eagerly` to ask the reducer to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree. Our current version only allows a limited, hard-coded set of eliminators with `ident.eagerly` (`nat_rect` on return types with either zero or one arrows, `list_rect` on return types with either zero or one arrows, and `List.nth_default`), but nothing in principle prevents automatic generation of the necessary code.

We construct our rewriter with

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

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 ζ . Throughout most of this paper, anywhere that `let ... in ...` appears, we have actually used **Let_In** in the code. It would alternatively be possible to extend the reification preprocessor to automatically convert `let ... in ...` to **Let_In**, but this may cause problems when converting the interpretation of the reified term with the pre-reified term, as Coq's conversion does not allow fine-tuning of when to inline or unfold **lets**.

To rewrite, we start with `cbv [example make map_db1]` 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

$$\begin{aligned} \text{map_dbl_cps}(\ell, k) &= \begin{cases} k([]) & \text{if } \ell = [] \\ \text{let } y := h +_{\text{ax}} h \text{ in } \text{map_dbl_cps}(t, & \text{if } \ell = h :: t \\ \quad (\lambda ys, k(y :: ys))) & \end{cases} \\ \text{make_cps}(n, m, v, k) &= \begin{cases} k(\underbrace{[v, \dots, v]}_n) & \text{if } m = 0 \\ \text{make_cps}(n, m - 1, v, & \text{if } m > 0 \\ \quad (\lambda \ell, \text{map_dbl_cps}(\ell, k)) & \end{cases} \\ \text{example_cps}_{n,m} &= \forall v, \text{make_cps}(n, m, v, \lambda x. x) = [] \end{aligned}$$

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 `pacity` to allow `vm_compute`, remains the best performer here, though the cost of rewriting everything is CPS may be prohibitive. Our method soundly beats `rewrite_strat`. We are additionally bottlenecked on `cbv`, which is used to unfold the goal post-rewriting and costs about a minute on the largest of terms; see Coq bug [#11151](#) for a discussion on what is wrong with Coq’s reduction here.

C.4 SieveOfEratosthenes

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
```

```

1471         PositiveSet.add
1472         composites
1473         (List.map
1474         (λ n, Pos.mul (Pos.of_nat (S n)) np)
1475         (List.seq 0 (Z.to_nat(max/Z.pos np))))),
1476         Pos.succ np)))
1477     fuel
1478     (nil, PositiveSet.empty, 2%positive))).
1479
1480 Definition sieve (n : Z)
1481   := Eval cbv [sieve'] in sieve' (Z.to_nat n) n.
1482
1483   We need four lemmas and an additional instance to create the rewriter:
1484
1485 Lemma eval_fold_right A B f x ls :
1486 @List.fold_right A B f x ls
1487 = ident.eagerly list_rect _ _
1488   x
1489   (λ l ls fold_right_ls, f l fold_right_ls)
1490   ls.
1491
1492 Lemma eval_app A xs ys :
1493 xs ++ ys
1494 = ident.eagerly list_rect A _
1495   ys
1496   (λ x xs app_xs_ys, x :: app_xs_ys)
1497   xs.
1498
1499 Lemma eval_map A B f ls :
1500 @List.map A B f ls
1501 = ident.eagerly list_rect _ _
1502   []
1503   (λ l ls map_ls, f l :: map_ls)
1504   ls.
1505
1506 Lemma eval_rev A xs :
1507 @List.rev A xs
1508 = (@list_rect _ (fun _ => _))
1509   []
1510   (λ x xs rev_xs, rev_xs ++ [x])%list
1511   xs.
1512
1513 Scheme Equality for PositiveSet.tree.
1514
1515 Definition PositiveSet_t_beq
1516   : PositiveSet.t -> PositiveSet.t -> bool
1517   := tree_beq.
1518
1519 Global Instance PositiveSet_reflect_eqb
1520   : reflect_rel (@eq PositiveSet.t) PositiveSet_t_beq
1521   := reflect_of_brel
1522     internal_tree_dec_b1 internal_tree_dec_lb.
1523
1524   We then create the rewriter with

```

```

1520 Make myrew := Rewriter For
1521   (eval_rect nat, eval_rect prod, eval_fold_right,
1522    eval_map, do_again eval_rev, eval_rect bool,
1523    @fst_pair, eval_rect list, eval_app)
1524   (with extra idents (Z.eqb, orb, Z.gtb,
1525    PositiveSet.elements, @fst, @snd,
1526    PositiveSet.mem, Pos.succ, PositiveSet.add,
1527    List.fold_right, List.map, List.seq, Pos.mul,
1528    S, Pos.of_nat, Z.to_nat, Z.div, Z.pos, 0,
1529    PositiveSet.empty))
1530   (with delta).

```

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

```

1532 Global Arguments Pos.to_nat !_ / .
1533

```

1534 D EXPERIENCE VS. LEAN AND SETOID_REWRITE

1536 Although all of our toy examples work with setoid_rewrite or rewrite_strat (until the terms
 1537 get too big), even the smallest of examples in Fiat Cryptography fell over using these tactics. When
 1538 attempting to use rewrite_strat for partial evaluation and rewriting on unsaturated Solinas
 1539 with 1 limb on small primes (such as 29), we were able to get rewrite_strat to finish after about
 1540 90 seconds. The bugs in rewrite_strat made finding the right magic invocation quite painful,
 1541 nonetheless; the invocation we settled on involved *sixteen* consecutive calls to rewrite_strat with
 1542 varying arguments and strategies. Trying to synthesize code for two limbs on slightly larger primes
 1543 (such as 113, which needs two limbs on a 64-bit machine) took about three hours. The widely used
 1544 primes tend to have around five to ten limbs; we leave extrapolating this slowdown to the reader.

1545 We have attached this experiment using rewrite_strat as fiat_crypto_via_rewrite_strat.v,
 1546 which is meant to be run in emacs/PG from inside the fiat-crypto directory, or in coqc by setting
 1547 COQPATH to the value emitted by make printenv in fiat-crypto and then invoking the command
 1548 coqc -q -R /path/to/fiat-crypto/src Crypto /path/to/fiat_crypto_via_rewrite_strat.v.
 1549 To test with the two-limb prime 113, change of_string "2^5-3" 8 in the definition of p to
 1550 of_string "2^7-15" 64.

1551 We also tried Lean, in the hopes that rewriting in Lean, specifically optimized for performance,
 1552 would be up to the challenge. Although Lean performed about 30% better than Coq on the 1-limb
 1553 example, taking a bit under a minute, it did not complete on the two-limb example even after four
 1554 hours (after which we stopped trying), and a five-limb example was still going after 40 hours.

1555 We have attached our experiments with running rewrite in Lean on the Fiat Cryptography
 1556 code as a supplement as well. We used Lean version 3.4.2, commit cbd2b6686ddb, Release. Run
 1557 make in fiat-crypto-lean to run the one-limb example; change open ex to open ex2 to try the
 1558 two-limb example, or to open ex5 to try the five-limb example.

1560 E READING THE CODE SUPPLEMENT

1562 We have attached both the code for implementing the rewriter, as well as a copy of Fiat Cryptography
 1563 adapted to use the rewriting framework. Both code supplements build with Coq 8.9 and Coq 8.10,
 1564 and they require that whichever OCaml was used to build Coq be installed on the system to
 1565 permit building plugins. (If Coq was installed via opam, then the correct version of OCaml will
 1566 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.

E.1 Code from section 1, Introduction

E.1.1 Code from subsection 1.1, A Motivating Example. The `prefixSums` example appears in the Coq source file `rewriter/src/Rewriter/Rewriter/Examples/PrefixSums.v`. Note that we use `dlet` rather than `let` in binding `acc` so that we can preserve the `let` binder even under ι reduction, which much of Coq's infrastructure performs eagerly. Because we attempt to isolate the dependency on the axiom of functional extensionality as much as possible, we also in practice require `Proper` instances for each higher-order identifier saying that each constant respects function extensionality. We hope to remove the dependency on function extensionality altogether in the future. Although we glossed over this detail in the body of this paper, we also prove

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

The `Make` command is exposed in `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 `RewriterRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`.

E.1.2 Code from subsection 1.2, Concerns of Trusted-Code-Base Size. There is no code mentioned in this section.

E.1.3 Code from subsection 1.3, Our Solution. We claimed that our solution meets five criteria. We briefly justify each criterion with a sentence or a pointer to code:

- We claimed that we **did not grow the trusted base** (excepting the axiom of functional extensionality). In any example file (of which a couple can be found in `rewriter/src/Rewriter/Rewriter/Examples/`), the `Make` command creates a rewriter package. Running

Print Assumptions on this new constant (often named `rewriter` or `myrew`) should demonstrate a lack of axioms other than functional extensionality. Print Assumptions may also be run on the proof that results from using the rewriter.

- We claimed **fast** partial evaluation with reasonable memory use; we assume that the performance graphs stand on their own to support this claim. Note that memory usage can be observed by making the benchmarks while passing `TIMED=1` to make.
- We claimed to allow reduction that **mixes rules of the definitional equality** with *equalities proven explicitly as theorems*; the “rules of the definitional equality” are, for example, β reduction, and we assert that it should be self-evident that our rewriter supports this.
- We claimed common-subterm **sharing preservation**. This is implemented by supporting the use of the `dlet` notation which is defined in `rewriter/src/Rewriter/Util/LetIn.v` via the `Let_In` constant. We will come back to the infrastructure that supports this.
- We claimed **extraction of standalone partial evaluators**. The extraction is performed in the 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 `ocamlpt` on Windows seem not to support outputting binaries without the `.exe` extension.

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`.

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

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

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

E.2.1 Code from subsection 2.1, Our Approach in Nine Steps. We match the nine steps with functions from the source code:

- (1) The given lemma statements are scraped for which named functions and types the rewriter package will support. This is performed by `rewriter_scrape_data` in the file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml` which invokes the \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`.

E.2.2 Our Approach in More Than Nine Steps. As the nine steps of [subsection 2.1](#) do not exactly match the code, we describe here a more accurate version of what is going on. For ease of readability, we do not clutter this description with references to the code supplement, instead allowing the reader to match up the steps here with the more coarse-grained ones in [subsection 2.1](#) or [subsubsection E.2.1](#).

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

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

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

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

- (1) We use `etransitivity` to allow rewriting separately on the left- and right-hand-sides of an equality. Note that we do not currently support rewriting in non-equality goals, but this is easily worked around using `let v := open_constr:(_) in replace <some term> with v` and then rewriting in the second goal.

- (2) We revert all hypotheses mentioned in the goal, and change the form of the goal from a universally quantified statement about equality into a statement that two functions are extensionally equal. Note that this step will fail if any hypotheses are functions not known to respect function extensionality via typeclass search.
- (3) We reify the side of the goal that is not an existential variable using the inductive codes in the specified package; the resulting goal equates the denotation of the newly reified term with the original `evvar`.
- (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 `evvar` with the resulting rewritten term.

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

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

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

E.3.1 Code from subsection 3.1, 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 is 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. [2008] 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`.

E.3.2 Code from subsection 3.2, Adding Higher-Order Features. The type `NbEt` 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`.

E.4 Code from section 4, Scaling Challenges

E.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 subsection E.2.1.

E.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.
```

```

1863
1864 Definition p : params
1865   := Eval compute in invert_Some (of_string "2^256-2^224+2^192+2^96-1" 64).
1866
1867 Goal True.
1868   (* Successful run: *)
1869   Time let v := (eval cbv
1870     -[Let_In
1871       runtime_nth_default
1872       runtime_add runtime_sub runtime_mul runtime_opp runtime_div runtime_modulo
1873       RT_Z.add_get_carry_full RT_Z.add_with_get_carry_full RT_Z.mul_split]
1874     in (GallinaDefOf p)) in
1875     idtac.
1876   (* Unsuccessful OOM run: *)
1877   Time let v := (eval cbv
1878     -[(*Let_In*)
1879       runtime_nth_default
1880       runtime_add runtime_sub runtime_mul runtime_opp runtime_div runtime_modulo
1881       RT_Z.add_get_carry_full RT_Z.add_with_get_carry_full RT_Z.mul_split]
1882     in (GallinaDefOf p)) in
1883     idtac.
1884
1885 Abort.
1886
1887   The UnderLets monad is defined in the file rewriter/src/Rewriter/Language/UnderLets.v.
1888   The definitions nbeT', nbeT, and nbeT_with_lets are in rewriter/src/Rewriter/Rewriter/
1889   Rewriter.v and are named value', value, and value_with_lets, respectively.

```

E.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`.

E.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`.

E.4.5 Code from subsection 4.5, Limitations and Preprocessing. The \mathcal{L}_{tac} hooks for extending the pre-processing 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. [subsection E.1.1](#)) 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. [subsection E.1.1](#)) 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 `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`.

E.5 Code from [section 5, Evaluation](#)

E.5.1 Code from [subsection 5.1, Microbenchmarks](#). This code is found in the files in `rewriter/src/Rewriter/Rewriter/Examples/`. We ran the microbenchmarks using the code in `rewriter/src/Rewriter/Rewriter/Examples/PerfTesting/Harness.v` together with some Makefile cleverness.

The code from [subsection 5.1.1, Rewriting Without Binders](#) can be found in `Plus0Tree.v`.

The code from [subsection 5.1.2, Rewriting Under Binders](#) can be found in `UnderLetsPlus0.v`.

The code used for the performance investigation mentioned in [subsection 5.1.3, Performance Bottlenecks of Proof-Producing Rewriting](#) and detailed in [Appendix A](#) is not part of the framework we are presenting, and thus not in the supplement.

The code from [subsection 5.1.4, Binders and Recursive Functions](#) can be found in `LiftLetsMap.v`.

The code from [subsection 5.1.5, Full Reduction](#) can be found in `SieveOfEratosthenes.v`.

E.5.2 Code from [subsection 5.2, Macrobenchmark: Fiat Cryptography](#). The rewrite rules are defined in `fiat-crypto/src/Rewriter/Rules.v` and proven in the file `fiat-crypto/src/Rewriter/RulesProofs.v`. They are turned into rewriters in the various files in `fiat-crypto/src/Rewriter/Passes/`. The shared inductives and definitions are defined in the Coq source 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.