Compositional Computational Reflection

Compositional Computational Reflection

Gregory Malecha[1], Adam Chlipala[2], and Thomas Braibant[3]

[1] Harvard University SEAS, Cambridge, MA, USA
gmalecha@cs.harvard.edu
[2] MIT CSAIL, Cambridge, MA, USA
adamc@csail.mit.edu
[3] INRIA, Rocquencourt, France
thomas.braibant@inria.fr

Abstract. Current work on computational reflection is single-minded; each reflective procedure is written with a specific application or scope in mind. Composition of these reflective procedures is done by a proof-generating tactic language such as Ltac. This composition, however, comes at the cost of both larger proof terms and redundant preprocessing. In this work, we propose a methodology for writing composable reflective procedures that solve many small tasks in a single invocation. The key technical insights are techniques for reasoning semantically about extensible syntax in intensional type theory. Our techniques make it possible to compose sound procedures and write generic procedures parametrized by lemmas mimicking Coq's support for hint databases.

Keywords: Computational reflection · automation · Coq · verification

1 Introduction

Imperative program verification requires orchestrating many different reasoning procedures. For it to scale to more sophisticated languages and larger programs, these procedures must be efficient. When using a proof assistant, a popular way to achieve good performance is with computational reflection [2], a technique for discharging proof obligations by running verified programs implemented in the proof assistant's logic.

While individually these procedures are fast, composing them relies on non-reflective, proof-generating tactic languages like Ltac [7]. While simple and flexible, this method is expensive. The brunt of the cost of computational reflection is in setting up the procedure and constructing the proof term; the actual computation is often relatively cheap. Composing many small reflective procedures requires paying this price for many moderately sized proof obligations rather than once for the entire goal.

To achieve this composition without returning to the non-reflective world, high-level reflective procedures must support extension in order to reason about domain-specific problems. Tactic-based languages support patterns such as higher-order tactics and hint databases that allow extending automation after the fact.

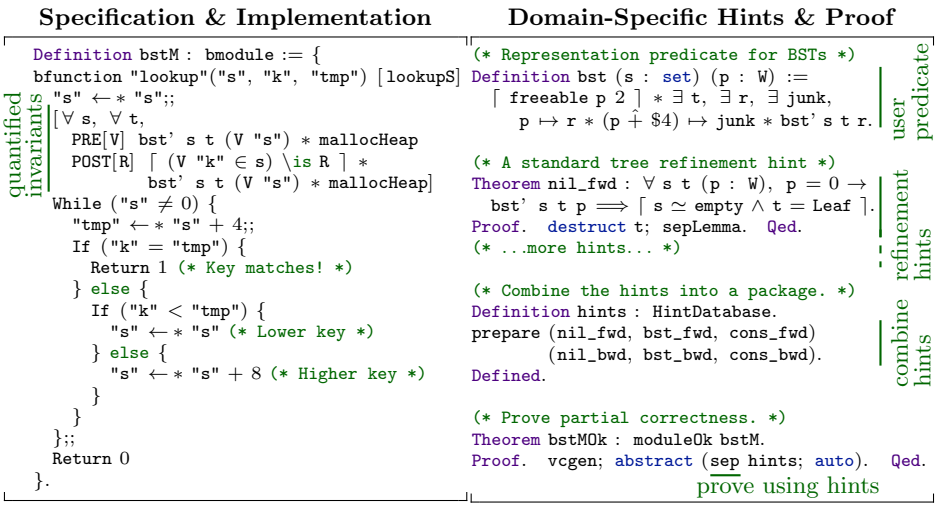

Fig. 1. Verified implementation of binary search trees implementing finite-set "lookup".

For example, Coq's `autorewrite` tactic is based on hint databases that package together a collection of rewrites and associated tactics to solve side conditions. These features, however, have not made their way to reflective procedures.

In this work, we focus on building extensible reflective procedures that perform many reasoning steps in a single invocation. Figure 1 demonstrates the degree of automation that we achieve applying our techniques to program verification in Bedrock [4], a Coq [6] library for low-level, imperative programming. Note that the implementation is completely separated (with the exception of loop invariants) from the automated verification. Effective automation for verifying such a program requires simultaneously reasoning about abstract predicates, low-level machine words, and high-level sets. To that end, our automation (`sep`) is written modularly and composed into large reflective procedures. Reasoning for problem-specific constructs is incorporated via `HintDatabase`s that are constructed completely automatically from both fully verified reflective procedures (similar to Ltac's `Hint Extern`) and guarded rewriting lemmas (similar to Ltac's `Hint Rewrite ... using ...`). The latter of these is constructed completely automatically from standard lemmas like `nil_fwd` above, which drastically lowers the overhead of applying our automation to reason about new abstract predicates.

In the rest of the paper we discuss the techniques that we have developed to support that kind of sophisticated reasoning reflectively. We begin with a short primer on computational reflection (Section 2) before discussing our technical contributions, which correspond to the features of our reflective procedures:

- Our proof procedures reason extensively about two forms of *variables* (Section 3.2): variables introduced by *existential quantifiers* in the goal and *unification variables* introduced by Ltac before our reflective procedures run.
- Our proof procedures *reason semantically about an open-ended set of symbols and types* (Section 3.3). Our approach allows us to build independent pro-

```
Inductive sexpr := (* syntactic separation logic formulas *)
| Star (l r : sexpr) | Opaque (p : nat) | Emp.

Fixpoint sexprD (ps : list hprop) (s : sexpr) : hprop :=
  match s with
  | Star l r ⇒ sexprD ps l ∗ sexprD ps r
  | Opaque f ⇒ nth_with_default (default := ⌜ False ⌝) ps f
  | Emp ⇒ ∅
  end.
Definition check_entailment (l r: sexpr): bool := (* reflective procedure *)

Theorem check_entailment_sound : ∀ ps lhs rhs, (* soundness proof *)
  check_entailment lhs rhs = true →
  sexprD ps lhs ⊢ sexprD ps rhs. (* separation-logic entailment *)
```

Listing 1. A reflective entailment checker for propositional separation logic.

cedures for reasoning about different domains, such as lists and bit-vectors, and compose them after the fact.

– Finally, our proof procedures are easy to *customize and extend without knowing about the details of reflection*. One drawback of reflective verification has been the need to write and verify programs in order to extend the automation. Combining the above techniques, we have built a more elementary interface that allows users to construct verified hint databases from Coq theorems completely automatically and pass them to generic reflective automation that applies the theorems (Section 3.4).

After presenting our technical contributions, we evaluate the performance and power of our automation (Section 4) and discuss related work (Section 5). Our techniques are implemented in the MirrorShard library that lays the foundation for the Bedrock automation. Both repositories are available online.

`https://github.com/gmalecha/mirror-shard/`
`https://github.com/gmalecha/bedrock-mirror-shard/`

2 Simple Entailment: A Computational Reflection Primer

Before diving into the novel bits, we sketch the high-level approach of computational reflection. We use entailment checking in a toy fragment of propositional separation logic [16] as our running example. Separation logic describes the program state compositionally by splitting it into disjoint pieces using the separating conjunction (notated $*$), which has the empty state (notated $\emptyset$) as its unit. For example, the formula $P * Q * R * \emptyset$ states that the entire program state can be divided into three disjoint parts described respectively by the opaque propositions P, Q, and R.

The first step in using computational reflection is to define a syntactic (called "reified") representation of formulas (`sexpr`); Listing 1 shows the code. The denotation function (`sexprD`) formalizes its meaning. `Star` and `Emp` represent $*$ and $\emptyset$ respectively; while `Opaque n`, for some index `n`, represents an uninterpreted proposition in the `ps` environment, e.g. P, Q, and R above. This indirection

provides a decidable equality on `sexpr`, which allows us to detect (conservatively) when two opaque propositions are equal. When `ps` does not contain a value for an index, our denotation function uses ⌈`False`⌉, a contradictory assertion.

Next we write a function (`check_entailment`) that determines whether the entailment is provable. Our simple algorithm erases all ∅ terms and crosses common terms off both sides of the entailment. If both sides wind up empty, then the entailment is provable. In order to use the procedure to prove an entailment, we prove a derived proof rule (the Coq theorem `check_entailment_sound`). The premise to this inference rule asserts that the function returns `true`, which can be checked efficiently by running the computation. If the result is `true`, the premise is justified by the reflexivity of equality. Notice that arbitrary entailments can be proved using this theorem by (1) reifying their syntax into the `sexpr` type and (2) applying the theorem with the quantifiers instantiated appropriately.

3 Composing Procedures

The entailment checker in the previous section is a good start, but it is not up to the challenges of program verification. Throughout this section we discuss how our technical contributions enable us to take it from a toy decision procedure to an extensible entailment checker capable of proving complex goals.

3.1 Syntax

Before we present our technical contributions, we set the stage with some more conventional elements of our syntax (shown in Listing 2).

The biggest inadequacy of the syntax presented in Section 2 is the representation of predicates. To illustrate the problem, consider the proposition $p \mapsto x$, expressing that the pointer p points to the value x. In the previous syntax this formula might be represented as `Opaque 1`, making it impossible to determine equivalence with $p + 0 \mapsto x$, which would be reified using a different index, e.g. as `Opaque 2`, since the terms are not *syntactically* equal.

To address this problem, we replace `Opaque n` with `Pred n xs` where `xs` is a list of arguments to the n^{th} predicate. Because these arguments are not separation-logic formulas, we introduce a second syntactic category (`expr`) to represent them. We could stop here if all arguments to predicates were e.g. machine words, which would be quite restrictive. To enable `expr` to represent expressions of an open set of types, we introduce a third syntactic category for types (`typ`) and an associated denotation function (`typD`). This denotation function shows up in the return type of `exprD`, which determines the meaning of a syntactic expression at a given (syntactic) type. When the expression does not have the given type, `exprD` returns `None`, signaling a type error.

Our new syntax also supports constants using the `Const` constructor of `expr`. While constants are special cases of 0-ary function symbols, distinguishing them allows our reflective procedures to compute with them. The price that we pay for this flexibility is an additional parameter (`ts`, introduced by the `Variable`

```
Inductive typ := tyProp | tyType (idx : nat). (* type syntax *)
Variable ts : list Type. (* remaining definitions are parametrized by ts *)
Definition typD (t : typ) : Type :=
  match t with
  | tyProp ⇒ Prop
  | tyType i ⇒ nth_with_default (default := Empty_set) ts i
  end.

Inductive expr := (* expression syntax *)
| Func (f : nat) (args : list expr) | Equal (t : typ) (l r : expr)
| Const (t : typ) (val : typD t) | Var (idx : nat) | UVar (idx : nat).

Definition env := list { t : typ & typD t }. (* variable environments *)

Record func := (* syntactic functions *)
{ Args: list typ; Range: typ; Impl: fold_right (→) (typD Range) (map typD Args) }

Definition exprD (fs: fenv) (us vs: env) (e: expr) (t: typ)
: option (typD t) := ...

Inductive sexpr := (* separation logic syntax *)
| Star (l r:sexpr) | Pred (p:nat) (args:list expr) | Emp | Inj (p:expr)
| Exists (t : typ) (s : sexpr).

Record pred := (* syntactic separation logic predicates *)
{ PArgs : list typ ; PImpl : fold_right (→) hprop (map typD PArgs) }.

Definition sexprD (fs : fenv) (ps : penv) (us vs : env) (e : sexpr) : hprop := ...
```

Listing 2. Our three-level, extensible syntax & its denotation.

line) to `expr` and `sexpr` to represent the type environment. Beyond constants, we also support injecting propositions into separation-logic formulas using `Inj` and polymorphic equality using `Equal` in `expr`. The latter is important since our extensible function environment does not support polymorphic definitions, an issue we discuss in more detail in Section 4.3.

The final syntactic forms are for binders and are discussed in the next section.

3.2 Binders & Unification Variables

Existential quantification is common in verification conditions for functional correctness, especially when reasoning about data abstraction. As a result, quantifier support is essential to fully reflective reasoning.

Our syntax supports existential quantifiers in separation-logic formulas using the `Exists` constructor. Syntactically, variables are represented using de Bruijn indices, and the environment (`vs : env`) is encoded as a list of dependent pairs of values and their syntactic types. The denotation of an existential quantifier prepends the quantified value to the variable environment, while the denotation of a variable looks up the value and checks it against the expected type.

The final syntactic form, `UVar`, represents Coq unification variables, which are placeholders for currently unknown terms. Our procedures determine appropriate values for these variables using a reflective unification procedure coded in Gallina. As we will see in Section 3.4, our ability to implement unification reflectively is a powerful feature of our approach.

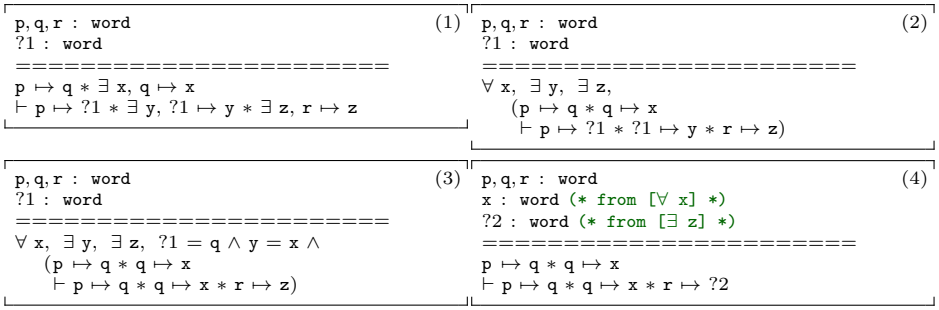

Fig. 2. Representation of quantifiers and unifications as they pass through our verification procedures: (a) initial goal; (b) result of lifting quantifiers; (c) direct output of the unification procedure; (d) after simplification with Ltac.

Figure 2 shows how our reflective procedures manipulate quantifiers and unification variables that occur in entailments. Note that while we show each step as an individual goal, all of the steps except the last are performed within a single reflective call.

Box (1) A simple entailment that might be passed to our reflective checker. As in Coq, unification variables are prefixed with question marks. For clarity, we include them explicitly above the line, implicitly representing their contexts as the identifiers that occur above them[4]. For example, the term used to instantiate ?1 can mention any of `p`, `q`, and `r`.

Box (2) The normal form that our procedures use lifts quantifiers to the top. Existentials to the left are introduced as `Var`s that are universally quantified, while those to the right are represented as `UVar`s and are existentially quantified. Here, the leading quantifiers are represented syntactically as lists of types, and the denotation function interprets them with the appropriate quantifiers.

Box (3) The result of unification is an instantiation of the unification variables. Semantically, this instantiation is a conjunction of equations, each between a unification variable and its instantiation. Here we see that ?1 was unified with `q`, and the value of the existentially quantified `y` has been chosen to be the newly introduced `x`.

Box (4) From here we cannot go any further reflectively, since unification variables only exist at the meta-level and thus cannot be manipulated in Coq's logic Gallina. Post-processing with Ltac cleans up the goal in Box (3) to look like the goal in Box (4). In particular, universally quantified variables are pulled into the context; unification variables are constructed for leading existentials using `eexists`; and instantiations are side-effected into the proof state by solving leading equations using `reflexivity`[5].

[4] The context of unification variables is not given to our procedures. They make the simplifying assumption that all terms are available in all contexts.

[5] If our reflective procedure instantiates a unification variable using terms outside of its context, `reflexivity` will fail, leaving the (likely unsolvable) goal to the user.

3.3 Compositional Semantic Reasoning

Only a small subset of operators are explicit in the syntax; the rest are represented by `Func` and `Pred`. For example, when we reason about the expression `Star x y`, the denotation, eliding the environments, is `sexprD x * sexprD y`. However, when we reason about the expression `Func 0 [x;y]`, the denotation becomes, again eliding the environments and the error-handling code, `(getFunc fs 0) (exprD x) (exprD y)`. To reason about the latter, we must express these assumptions as premises to the soundness proof.

To explain our technique for expressing these constraints, we introduce the following simple procedure for reasoning about the commutativity of addition.

```
Definition prove_plus_comm ts (e : expr ts) : bool :=
  match e with
  | Equal 1 (Func 0 [x ; y]) (Func 0 [y' ; x']) ⇒ expr_eq x x' && expr_eq y y'
  | _ ⇒ false
  end.
```

Already this procedure makes the assumption that `nat` is at type index 1 and `plus` is at function index 0. We could prove this procedure sound for the environments `[bool;nat]` and `[plus]`, but this proof would not be useful for extended environments. To develop reflective procedures independently and link them together after the fact, we need a compositional way to express these assumptions.

Our approach is to use a computational, rather than propositional, constraint formulation. To express constraints computationally, we quantify over an *arbitrary* environment and compute a *derived* environment that manifestly satisfies the constraints and is otherwise exactly the same as the original. The following function derives an environment from `e` that is guaranteed to satisfy `c`.

```
Fixpoint applyC (T: Type) (d : T) (c: constraints T) (e: list T) : list T :=
  match c with
  | nil ⇒ e
  | Any :: c' ⇒ hd_with_default (default := d) e :: applyC T d c' (tl e)
  | Exact v :: c' ⇒ v :: applyC T d c' (tl e)
  end.
```

To see `applyC` in action, we return to our example and declare the type environment constraints for the commutativity prover. Note that by using `Any` in position 0, we allow other procedures to choose a meaning for `tyType 0`.

```
Let TC : constraints Type := [Any; Exact nat].
```

Next we state the constraints for the function environment, which requires a syntactic representation of the `plus` function. Since the type of this syntactic representation depends on the type environment, we apply our technique, parametrizing by an arbitrary environment and retrofitting it with our constraints via `applyC`. In code:

```
Definition plus_fn (ts : list Type) : func (applyC TC ts) :=
{ Args := [tyType 1; tyType 1] ; Range := tyType 1 ; Impl := plus }.
```

This term type checks because `applyC` and `typD` reduce, making the following equations hold *definitionally*.

$$\mathsf{typD}(\mathsf{applyC}\,\mathsf{TC}\,ts)(\mathsf{tyType}\,1) \equiv \mathsf{typD}\,(\mathsf{hd}\,\mathsf{d}\,ts :: \mathsf{nat} :: \mathsf{tl}\,(\mathsf{tl}\,ts))\,(\mathsf{tyType}\,1) \equiv \mathsf{nat}$$

The essential enabling property of `applyC` is that when `c` is a cons cell, the result is *syntactically* a cons cell and is not blocked by a `match` on `e`.

If we had stated the property propositionally and *proved* the equality, then `Impl` would require an explicit cast, like so.

```
Definition plus_fn_bad ts (pf : TC ⊨ ts) : func ts := (* ⊨ is 'holds on' *)
{ Args := [tyType 1; tyType 1] ; Range := tyType 1
; Impl := match compatible_reduces pf in _ = t return t → t → t with
          | eq_refl ⇒ plus end }.
```

Reasoning about casts in intensional type theory is difficult because the discriminee of the `match` must reduce to a constructor before the match can be eliminated. This behavior blocks conversion, making "seemingly equal" terms unequal. Our technique, on the other hand, does not even manifest the cast.

Using these definitions, we can prove the soundness of our simple commutativity prover using the following theorem statement.

```
Let FC ts : constraints (function (applyC TC ts)) := [Exact (plus_fn ts)].
Theorem prove_plus_comm_sound
: ∀ (ts : list Type), let ts' := applyC TC ts in
  ∀ (fs : functions ts'), let fs' := applyC FC fs in
  ∀ e1 e2, WellTyped ts' fs' e1 (tyType 1) → WellTyped ts' fs' e2 (tyType 1) →
    prove_plus_comm e1 e2 = true →
    exprD ts' fs' e1 (tyType 1) =nat exprD ts' fs' e1 (tyType 1).
```

With this formulation, `getFunc fs' 0` reduces to `plus`, making the proof follow from the commutativity of `plus`; a simple proof for a simple property.

```
  ∀ fs, let fs' := applyC FC fs in (getFunc fs' 0) x y = (getFunc fs' 0) y x
≡ ∀ fs, x + y = y + x
```

The fact that `applyC c l = l` when $c \models l$ justifies the completeness of the technique. Any theorem that is provable with the propositional formulation is also provable with our computational formulation.

Composition. `applyC`'s computational properties make it well-suited for composition. When two constraints are compatible, i.e. they do not specify different values for any index, `applyC` commutes *definitionally*.

$$\mathsf{applyC}\,C_1\,(\mathsf{applyC}\,C_2\,e) \equiv \mathsf{applyC}\,C_2\,(\mathsf{applyC}\,C_1\,e)$$

We can leverage this property for easy composition of functions and proofs without needing to reason about casts. For example, suppose we have two functions (analogously proofs) phrased using `applyC`, say $\mathtt{p}_1$ and $\mathtt{p}_2$, with different, but compatible, constraints $\mathtt{TC}_1$ and $\mathtt{TC}_2$. Composing each function with the application of the other's constraints gives us the following:

```
(fun ts ⇒ p1 (applyC TC2 ts)) : ∀ ts, expr (applyC TC1 (applyC TC2 ts)) → bool
(fun ts ⇒ p2 (applyC TC1 ts)) : ∀ ts, expr (applyC TC2 (applyC TC1 ts)) → bool
```

Since these types are definitionally equal, we can interchange the terms, for example, adding them to the same list or composing them via a simple disjunction without the need for explicit proofs or explicit casts:

```
Definition either ts (p1 p2 : expr ts → bool) (e : expr ts) : bool := p1 e || p2 e.
```

Packaging. To simplify passing provers around, we package them with their constraints and their soundness proofs using Coq's dependent records[6].

```
Record HintDatabase :=
{ Types  : constraints Type
; Funcs  : ∀ ts, constraints (func (applyC Types ts))
; Prover : ∀ ts, ProverT (applyC Types ts)
; Prover_sound : ∀ ts fs,
  ProverOk (applyC Types ts) (applyC (Funcs ts) fs) (Prover ts) }.
```

The first two fields represent the constraints for the type and function environments, with the latter phrased using our technique. The `Prover` field would contain the functional prover code, e.g. `prove_sum_comm` wrapped in our prover interface. The final field, `Prover_sound`, would contain the proof that the procedure is sound, derived from `prove_sum_comm_sound`.

To invoke a reflective procedure with a particular hint database, we rely on Ltac to handle the constraints. For example, the top-level soundness lemma for cancellation has the following form:

```
Theorem Apply_cancel_sound ts (fs : fenv ts)
  (prover : ProverT ts) (prover_ok : ProverOk ts fs prover)
: ∀ (lhs rhs : sexpr ts) (us vs : env ts), ...
```

To apply such a theorem, our reification process projects out the constraints from the hint database and uses them as base environments when constructing the syntactic terms. We can then instantiate the prover and its soundness proof to work on the extended environments (distinguished using primes) by simple application.

```
Apply_cancel_sound ts' fs'
   (hints.(Prover) ts' : ProverT ts' (*≡ProverT (applyC hints.(Types) ts')*))
   (hints.(Prover_sound) ts' fs' : ProverOk ts' fs') ...
```

Taking a closer look at the types we notice that the definition of `applyC` justifies the type assertions for the final two arguments. For example, the third argument actually has type `ProverT (applyC hints.(Types) ts')`, but since we construct `ts'` to retain the entries of `hints.(Types)`, this type is definitionally equal to `ProverT ts'`.

[6] The `HintDatabase` record in MirrorShard contains an additional field for the constraints on the predicate environment, but our example does not require it.

3.4 Generic Extension with Reified Lemmas

Writing and verifying reflective procedures can be cumbersome. For example, when verifying linked data structures like lists, the automation often requires rewriting by a separation-logic entailment. If each new entailment lemma required writing and verifying a new reflective procedure, users would spend more time building automation than using it. In Ltac, the sort of generic procedure we want is provided by the parametrized tacticals `rewrite` or `autorewrite`. In this section we show how the techniques from the previous two sections allow us to implement a generic, reflective rewriting procedure for separation-logic formulas that is parametrized by a list of lemmas to rewrite with.

As with all reflective tasks, the first hurdle is the representation. The variable-related features from Section 3.2 provide a simple way to represent lemmas.

```
Record lemma (ts : list Type) :=        (* Example:                        *)
{ Foralls : list typ                    (* ∀ x ls,                         *)
; Hyps : list (expr ts)                 (*   x = 0 →                       *)
; Lhs : sexpr ts ; Rhs : sexpr ts }.     (*   llist x ls ⊢ [ ls = nil ]     *)

Definition lemmaD ts (fs : fenv ts) (ps : penv ts) (l : lemma) : Prop :=
  forallEach ts l.(Foralls) (fun vs ⇒
    implEach ts fs nil vs l.(Hyps)
      (sexprD ts fs ps nil vs l.(Lhs) ⊢ sexprD ts fs ps nil vs l.(Rhs))).
```

The function `lemmaD` translates a reified lemma statement into a Gallina proposition. Here, `forallEach` introduces universal quantifiers for the given types, packaging the quantified variables into an environment (`vs`) that it passes to the continuation. This environment is then used as the variable environment for the premises (denoted using `implEach`) and the conclusion.

We reduce the problem of determining when a lemma can be used to a unification problem. Our procedure replaces the universally quantified `Var`s in the lemma statement with `UVar`s, setting up a unification "pattern." This pattern is then passed to the unification procedure that we mentioned in Section 3.2. If unification succeeds, we get a substitution that we can use to instantiate the lemma. Using provers like `prove_plus_comm` from the previous section allows us to discharge the premises. If all of the premises are discharged, we can replace the candidate term completely reflectively. Our rewriting procedure is able to rewrite in the premise and in the conclusion reusing mostly the same code and proofs in both cases.

The most difficult part of the proof lies in justifying the existence of values for the quantified variables. The unification procedure returns the expressions to use but, in order to support finding them incrementally, it only guarantees that they are well-typed in the environment that contains the new unification variables. Justifying that these expressions do not mention the new unification variables requires reasoning about the acyclicity of the instantiation, which is guaranteed by the occurs check in the unification algorithm. While complex, this proof is done once and used whenever we need to type-theoretically strengthen the result of unification.

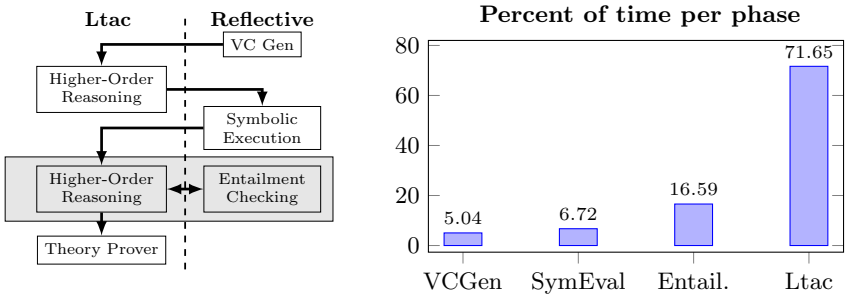

Fig. 3. Verification process and the breakdown of verification time.

To make rewriting hints easy to use, we have completely automated the construction of syntactic lemmas and their composition into (an extended version of) the hint databases described in the previous section. These extended hint databases carry two lists of lemmas, one for forward rewriting and the other for backward rewriting, as well as their corresponding proof terms.

4 Evaluation & Discussion

The techniques in Sections 3.2-3.4 form the core insights of the MirrorShard framework. In this section we discuss our results applying this framework to verify programs written in the Bedrock system. We begin with an overview of the end-to-end verification process before justifying our claims from the introduction about the benefits of building broader reflective procedures.

We restrict our evaluation to a collection of data structure libraries including a memory allocator, linked-list operations, sets implemented as unsorted lists and binary trees, and queues. Together, these constitute approximately 355 lines of code that generate 253 verification conditions. The source code to these examples is found in the `examples` directory of our `bedrock-mirror-shard` repository.

4.1 The Verification Procedure

The end-to-end automation that verifies an entire Bedrock module is broken down into three independent, reflective tasks (verification condition generation, symbolic execution, and entailment checking) punctuated by Ltac-based higher-order reasoning. Figure 3 shows the overall process. We focus on the latter two tasks that, combined, apply to a single verification condition. Each verification condition assumes a precondition and that a particular code path is followed. The obligation is to show either that the code runs without errors (progress) or finishes in a state satisfying some postcondition (preservation). We focus on the second case since it is more interesting.

To solve a preservation verification condition, symbolic execution runs to compute the (strongest) postcondition of the path under the precondition. Next, an Ltac tactic runs to determine the postcondition. We use Ltac because we

may require non-trivial higher-order reasoning (for instance, if the postcondition comes from the spec of a first-class function being called). This step reduces the goal to a separation-logic entailment that is discharged by our entailment checker. Because higher-order function specifications may involve nested assertions about specifications for other functions, entailment checking and the Ltac for higher-order reasoning run in a loop. Finally, user-defined Ltac runs to discharge any side conditions that could not be solved by our reflective procedures. In practice, these side conditions tend to be the pure parts of specifications, e.g. reasoning about Coq's `length` function when verifying its Bedrock implementation.

Figure 3 shows how the verification time is distributed between the different phases across our data-structure examples. Note that while our reflective procedures end up doing most, if not all, of the heavy lifting, almost three-quarters of our verification time is spent running Ltac, suggesting that while we could further optimize our reflective procedures, the biggest improvements would come from making more of the verification reflective.

4.2 Reflective Performance

Previous work [2,11] has demonstrated the performance and scaling benefits of reflective automation, and our work enjoys similar benefits. More central to our thesis is the benefit of reflective composition and user extension, which we evaluate in the context of symbolic execution.

Consider the following path through the length function for linked lists:

```
assume( *(Sp+4) ≠ 0 );    (* not at the end of the list *)
*(Sp+8) := *(Sp+8) + 1 ; (* increment the length counter *)
Rv := *(Sp+4) ;           (* get the next pointer *)
*(Sp+4) = *Rv             (* update "current" *)
```

The references from the stack pointer `Sp` are to local variables. `Sp`+8 is the location of the length counter, and `Sp`+4 is the location of the "current" pointer. The first line is the result of knowing that the conditional comparing `current` to `null` returned false, implying that evaluation is not at the end of the list, which justifies the memory dereference on the last line where the code reads the `next` pointer of the current linked-list cell (`**(Sp+4)`).

In order to exploit this information during symbolic execution, our symbolic executor uses the following hint, provided in a hint database:

```
Lemma llist_cons_fwd : ∀ ls (p : W), p ≠ 0
    → llist ls p ⊢ ∃ x, ∃ ls', [| ls = x :: ls' |] * ∃ p',  (p ↦ x, p') * llist ls' p'.
```

This lemma is fed to the reflective rewriting framework discussed in Section 3.4, which exposes the ↦ predicate that symbolic execution knows how to interpret[7].

[7] Not just ↦ but also some other "base" predicates are interpreted by independent, user-defined reflective procedures that plug into our symbolic execution framework.

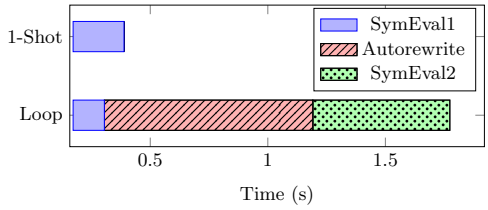

Fig. 4. Profiling of 1-shot symbolic execution versus Ltac composition with `autorewrite` on the small example goal.

Without this mechanism, we could achieve the same automation by running an Ltac loop bouncing between our reflective symbolic execution and the `autorewrite` tactic to perform this rewriting. In the above example, this loop would call symbolic execution, which would get stuck on the final instruction, falling back on `autorewrite` to expose the cons cell, enabling a second call to symbolic evaluation to complete the task.

Figure 4 shows how the loop approach compares to our fully reflective procedure (1-Shot). Using the latter, the entire symbolic execution takes 0.39 seconds, less than half the amount of time (0.89 seconds) taken by `autorewrite` to perform just the rewriting. Overall the reflective composition results in a 4.6x speedup over the Ltac-based composition on this goal, translating into 44 seconds when applied to the entire linked-list module.

While our reflective rewriter is not as powerful as `autorewrite`, it is customizable in the same way. Further, because it is written in Gallina rather than hardcoded inside Coq, we can extend it with smarter unification that, for example, can reason about provable rather than just definitional equality.

4.3 Limitations & Future Work

MirrorShard's success as the core automation for Bedrock is strong evidence for its expressivity. However, the expressivity of Coq's logic limits the power of reflective procedures.

MirrorShard's computational formulation of constraints relies crucially on constants in certain places, for example the indices of types. For example, while it is easy to write a procedure that is sound for any environment where `nat` is located at position 1, it is more difficult to write a procedure parametrized by `x` that is sound for any environment where `nat` is at position `x`. While it is possible to manipulate proofs explicitly and achieve the latter degree of parametrization, in this work we have opted for the simpler solution. As we expand the ideas and techniques beyond separation logic, developing more parametrized procedures will likely become more important.

MirrorShard's syntax does not support a general notion of binders, only existential quantifiers in separation-logic formulas. While this limitation has not been problematic for entailment checking and symbolic execution, it prevents us from reasoning about e.g. inline functions and `match` expressions. Supporting a general notion of binder may provide a way to automate reflectively some of the

tasks that we currently accomplish in Ltac, increasing the scope of reflection and further improving performance.

While binders should be within our grasp, restrictions of the logic put other features, like general support for polymorphic types, farther out of reach. Type functions can be encoded for special fixed arities, but a general solution allowing arbitrary arities requires universe polymorphism. Universe polymorphism as described by Harper and Pollack [12] is slated for Coq 8.5 and will solve some of these issues.

Finally, general value-dependent types pose an even greater problem. The MirrorShard representation stratifies the type and term languages, but truly dependent types would require these to be unified, making the type of the denotation function mention itself in the style of very dependent functions [13].

5 Related Work

MirrorShard is not the first verified implementation of separation-logic automation, but it is the first to support modular user extension. Marti and Affeldt [15] implemented a verified version of Smallfoot [1], and Stewart et al. [17] verified a more sophisticated heap theorem prover based on paramodulation. Both of these systems are limited to the standard points-to and singly-linked list predicates, and extending either to support user-defined abstract predicates with equations would likely require a considerable overhaul of both the procedure and its proof.

While program verification is our application, our technical contributions are our techniques for phrasing, composing, and extending reflective procedures and their proofs. The applicability of these techniques extends well beyond program verification. Several projects have built large, generic reflective procedures. In his PhD thesis, Lescuyer [14] describes a reflective implementation of an SMT solver. While he also uses an environment-based representation, he is unable to reason about it semantically. As a result, it is not clear how to support first-class hint databases or include additional theories that need to reason semantically about symbols represented using the environment. Our work also supports quantifiers.

Similar to our rewriting engine is the work by Braibant and Pous on reasoning modulo associativity and commutativity [3]. Like Lescuyer, they specialize their procedures for reasoning semantically about a fixed set of symbols (in their case an abstract commutative, associative operator), which removes the need to reason about multiple types or multiple operators. Our techniques support both.

Recent work by Claret et al. [5] on posterior simulation for reflective proofs aims to make it easier to write reflective procedures by supporting side effects and branching proof search efficiently. This goal is complementary to our own work and offers a method of automatic caching for results of the (potentially large) proof searches that our extensible procedures enable. This caching may become essential if reflective procedures begin to rely heavily on speculation.

In the wider sphere of proof automation, Mtac [18] proposes a monadic language for writing Gallina terms that are run during program elaboration. Unlike MirrorShard, Mtac supports dependent and polymorphic types; however, its

support for binder manipulation is less sophisticated. For example, it does not appear to be possible to apply lemmas without knowing their types a priori, making it difficult to parametrize by lemmas that are applied automatically.

The Ssreflect tactic library [10] has become a popular alternative to Ltac. Ssreflect provides a higher-level tactic language and support for "small-scale" reflection. The tactics aim to make it easier to refactor proofs and lemmas, but it is still focused on smaller reasoning steps. This approach avoids the need to compose reflective procedures but requires more effort by the user to determine and perform the appropriate reasoning explicitly.

One of the core problems that we overcome in our formulation is the expression problem. Our concrete syntax is similar to that of Garillot and Werner [9], though their work does not suggest any methods for achieving semantic reasoning, which is essential to reasoning about actual terms. Delaware et al. [8] recently proposed another solution to this problem using Church encodings. While useful for reasoning about the metatheoretic properties of programming languages, it is not clear that Church encodings completely solve the issues that arise in computational reflection. In particular, representing terms as functions can make them costly to compute with and type check.

6 Conclusions

In this work we presented three novel techniques for building extensible reflective procedures in Coq. First, we presented a reflected representation of unification variables and existential quantifiers, which we reason about using a verified unification algorithm. Our second technique is a simple encoding of extensible syntax suitable for computational reflection, plus a formulation of constraints that allows reasoning about this representation without any runtime overhead. Our third technique is a method for building first-class, reflected hint databases that can be used by reflective procedures.

These techniques form the core technical insights of MirrorShard, a reusable Coq library for reflective procedures about separation logic. The extensibility of these procedures allows them to reason about broader problems by reflectively orchestrating general and domain-specific reasoning. Our evaluation shows that this approach can provide a significant speedup over performing the extensible reasoning in a hybrid of reflective procedures and Ltac.

Acknowledgments. The authors thank Patrick Hulin and Edward Z. Yang for their contributions to the MirrorShard implementation. We received helpful feedback on this paper from: Andrew W. Appel, Jesper Bengtson, Josiah Dodds, Georges Gonthier, Daniel Huang, Andrew Johnson, Jacques-Henri Jourdan, Scott Moore, Greg Morrisett, and Kenneth Roe. This work has been supported in part by a Facebook Fellowship, an NSF Graduate Research Fellowship, NSF grant CCF-1253229, AFRL under agreement FA8650-10-C-7090, and DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions

contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2005.
2. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. TACS*, 1997.
3. Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In *CPP*, pages 167–182, 2011.
4. Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proc. ICFP*, pages 391–402. ACM, 2013.
5. Guillaume Claret, Lourdes Del Carmen Gonzalez Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *Interactive Theorem Proving*, Rennes, France, July 2013.
6. Coq Development Team. The Coq proof assistant reference manual, version 8.4. 2012.
7. David Delahaye. A tactic language for the system Coq. In *Proc. LPAR*, 2000.
8. Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory a la carte. *SIGPLAN Not.*, 48(1):207–218, January 2013.
9. Franois Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In *Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 368–382. Springer Berlin Heidelberg, 2007.
10. Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq System. Rapport de recherche RR-6455, INRIA, 2008.
11. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proc. ICFP*, 2002.
12. Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107 – 136, 1991.
13. Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996.
14. Stéphane Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq.* Thèse de doctorat, Université Paris-Sud, January 2011.
15. Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 25(3):135–147, 2008.
16. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, pages 55–74. IEEE Computer Society, 2002.
17. Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. In *Proc. ICFP*, 2012.
18. Beta Ziliani, Derek Dreyer, Neel Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *Proc. ICFP*, 2013.