Extensible and Efficient Automation through Reflective Tactics

Gregory Malecha and Jesper Bengtson

[1] University of California, San Diego
[2] IT University of Copenhagen

Abstract. Foundational proof assistants simultaneously offer both expressive logics and strong guarantees. The price they pay for this flexibility is often the need to build and check explicit proof objects which can be expensive. In this work we develop a collection of techniques for building reflective automation, where proofs are witnessed by verified decision procedures rather than verbose proof objects. Our techniques center around a verified domain specific language for proving, $\mathcal{R}_{tac}$, written in Gallina, Coq's logic. The design of tactics makes it easy to combine them into higher-level automation that can be proved sound in a mostly automated way. Furthermore, unlike traditional uses of reflection, $\mathcal{R}_{tac}$ tactics are independent of the underlying problem domain, which allows them to be re-tasked to automate new problems with very little effort. We demonstrate the usability of $\mathcal{R}_{tac}$ through several case studies demonstrating orders of magnitude speedups for relatively little engineering work.

1 Introduction

Foundational proof assistants provide strong guarantees about properties expressed in rich logics. They have been applied to reason about operating systems [**?**], compilers [**?**], databases [**?**], and cyber-physical systems [**?**], just to name a few. In all of these contexts, users leverage the expressivity of the underlying logic to state their problem and use the automation provided by the proof assistant, often in the form of a "tactic language," to prove the properties they care about, e.g. the correctness of a compiler.

The problem with this rosy picture is that foundational proofs are large because they need to spell out the justification in complete detail. This is especially true in rich dependent type theories such as Coq [**?**] and Agda [**?**] where proofs are first-class objects with interesting structural and computational properties. Several recent projects have made the overhead of proof objects painfully clear: the Verified Software Toolchain [**?**], CertiKOS [**?**], and Bedrock [**?**]. All of these projects run into resource limitations, often in memory, but also in build times. Proof generating automation tends to work well when applied to small problems, but scales poorly as problems grow. To solve this problem we need a way to extend the proof checker in a trustworthy way so that it can check certain properties more efficiently. Doing this allows us to dramatically improve both the time and memory usage of foundational verification.

The expressivity of foundational constructive proof assistants provides a technique for "extending the proof checker": computational reflection [?,?,?,?,?]. Using computational reflection, a developer implements a custom program to check properties outright and proves that the program is sound, i.e. if it claims that a property is provable, then a proof of the property exists. With computational reflection, previously large proof objects can be replaced by calls to these custom decision procedures. Executing these procedures can be much more efficient than type checking explicit proof terms. However, the current approach to building these reflective procedures makes them difficult to construct, cumbersome to adapt to new instances, and almost impossible to compose to build higher-level automation.

In this work we show how to easily build reflective automation that is both extensible and efficient. Our approach is to separate the core reflective language from the domain-specific symbols that the automation reasons about. This enables us to build a Domain Specific Language (DSL), $\mathcal{R}_{tac}$, for reflective automation that is reusable across any base theory. We show that naïvely translating automation from $\mathcal{L}_{tac}$, Coq's built-in tactic language, to $\mathcal{R}_{tac}$ typically leads to at least an order of magnitude speedup on reasonable problem sizes.

We begin with a brief primer applying computational reflection to automate monoidal equivalence checking (Section 2). In Section 3 we highlight our techniques by reworking this example in $\mathcal{R}_{tac}$ highlighting the key pieces of our work from the client's point of view. We then proceed to our contributions:

- We develop MirrorCore, a reusable Coq library for general and extensible computational reflection built around a core λ-calculus (Section 4).
- We build $\mathcal{R}_{tac}$—the first foundational, feature-rich reflective tactic language (Section 5). By separating domain-specific meaning from generic manipulation, $\mathcal{R}_{tac}$ is able to provide high-level building blocks that are independent of the underlying domain. Additionally the user-exposed interface makes it easy to combine these general tactics into larger, domain-specific automation and derive the corresponding soundness proofs essentially for free.
- We demonstrate the extensible nature of $\mathcal{R}_{tac}$ by developing a reflective setoid rewriter as an $\mathcal{R}_{tac}$ tactic (Section **??**). The custom implementation follows the MirrorCore recipe of separating the core syntax from the domain-specific symbols allowing the rewriter to be re-tasked to a range of problem domains. In addition, the procedure is higher-order and can invoke $\mathcal{R}_{tac}$ tactics to discharge side conditions during rewriting.
- We evaluate MirrorCore and $\mathcal{R}_{tac}$ by developing reflective procedures for different domains (Section **??**). We show that our automation has substantially better scaling properties than traditional $\mathcal{L}_{tac}$ automation has and is quite simple to write in most cases.

Before concluding, we survey a range of related work related to automation alternatives in proof assistants (Section **??**).

All of our results have been mechanized in the Coq proof assistant. The MirrorCore library and examples are available online:

`https://github.com/gmalecha/mirror-core`

2 A Computational Reflection Primer

In this section we give an overview of the core ideas in computational reflection which we build on in the remainder of the paper. We base our overview on the problem of proving equality in a commutative monoid. For example, consider proving the following where $\oplus$ is the plus operator in the monoid.

$$x \oplus 2 \oplus 3 \oplus 4 = 4 \oplus 3 \oplus 2 \oplus x$$

A naïve proof would use the transitivity of equality to witness the way to permute the elements on the left until they match those on the right. While not particularly difficult, the proof is often at least quadratic in the size of the property, which means that checking large problems quickly becomes expensive.

To use computational reflection to prove this theorem, we write a procedure that accepts the two expressions and checks the property directly. The first step is to build a syntactic (also called "reified") representation of the problem. In this example, we use the following language.

$$\text{(expressions) } e ::= \mathsf{N}\, n \mid \mathsf{R}\, i \mid e_1 \mathbin{\underline{\oplus}} e_2$$

The language represents constants directly (using N) but it hides quantified values such as x behind an index (using R). The syntactic representation is necessary because computations can not inspect the structure of terms, only their values. For example, pattern matching on $x \oplus y$ does not reduce since x and y are not closed values. A syntactic representation exposes the *syntax* of the goal, e.g. $\mathsf{R}\,1 \mathbin{\underline{\oplus}} \mathsf{R}\,2$, which functions can inspect.

We formalize the meaning of the syntax through a "denotation function" ($[\![-]\!]_\rho$) parameterized by the environment of opaque symbols (ρ). For our syntax, $[\![-]\!]_\rho$ has three cases:

$$[\![\mathsf{N}\, n]\!]_\rho = n \qquad [\![\mathsf{R}\, x]\!]_\rho = \rho\, x \qquad [\![e_1 \mathbin{\underline{\oplus}} e_2]\!]_\rho = [\![e_1]\!]_\rho \oplus [\![e_2]\!]_\rho$$

Using this syntax, the problem instance above could be represented as:

$$[\![\mathsf{R}\,0 \mathbin{\underline{\oplus}} \mathsf{N}\,2 \mathbin{\underline{\oplus}} \mathsf{N}\,3 \mathbin{\underline{\oplus}} \mathsf{N}\,4]\!]_{\{0 \mapsto x\}} = x \oplus 2 \oplus 3 \oplus 4$$
$$[\![\mathsf{N}\,4 \mathbin{\underline{\oplus}} \mathsf{N}\,3 \mathbin{\underline{\oplus}} \mathsf{N}\,2 \mathbin{\underline{\oplus}} \mathsf{R}\,0]\!]_{\{0 \mapsto x\}} = 4 \oplus 3 \oplus 2 \oplus x$$

With the syntax in hand, we can now write a procedure (Mcheck) that determines whether the terms are equal by flattening each expression into a list and checking whether one list is a permutation of the other. The soundness theorem of `Mcheck` states that if `Mcheck` returns true then the meaning of the two arguments are provably equal. Formally,

$$\mathsf{Mcheck\_sound} : \forall e_1\, e_2, \texttt{Mcheck}\, e_1\, e_2 = \mathsf{true} \to \forall \rho, [\![e_1]\!]_\rho = [\![e_2]\!]_\rho$$

Using $\mathsf{Mcheck\_sound}$ we can prove the example problem with following proof which is linear in the size of problem.

$\mathsf{Mcheck\_sound}$	
$(\mathsf{R}\,0 \mathbin{\underline{\oplus}} \mathsf{N}\,2 \mathbin{\underline{\oplus}} \mathsf{N}\,3 \mathbin{\underline{\oplus}} \mathsf{N}\,4)\ (\mathsf{N}\,4 \mathbin{\underline{\oplus}} \mathsf{N}\,3 \mathbin{\underline{\oplus}} \mathsf{N}\,2 \mathbin{\underline{\oplus}} \mathsf{R}\,0)$	(syntactic problem)
$\mathsf{eq\_refl}$	(proof `Mcheck` returned true)
$\{0 \mapsto x\}$	(environment)

3 $\mathcal{R}_{tac}$ from the Client's Perspective

Before delving into the technical machinery that underlies our framework we highlight the end result. In Figure 1, we excerpt an $\mathcal{R}_{tac}$ implementation of the monoidal equivalence checker described in Section 2[3]. The automation builds directly on the Coq definitions of commutative monoids shown in step (0).

The first step is to build a data type that can represent the properties that we care about. In Section 2 we built a custom data type and spelled out all of the cases explicitly. Here, we build the syntactic representation by instantiating MirrorCore's generic language (`expr`) with domain specific types (`mon_typ`) and symbols (`mon_sym`). As we will see in Section 4, the `expr` language provides a variety of features that are helpful when building automation. Once we have defined the syntax, we use MirrorCore's `Reify` plugin to automatically construct syntactic representations of the lemmas that we will use in our automation (step (2)).

In step (3), we use these lemmas to write our reflective automation using the $\mathcal{R}_{tac}$ DSL. The entry point to the automation is the `Mcheck` tactic but the core procedures are `iter_left` and `iter_right` which permute the left- (`iter_left`) and right-hand sides (`iter_right`) of the equality until matching terms can be cancelled. Each tactic consists of a bounded recursion (`REC`) where the body tries one of several tactics (`FIRST`). For example, `iter_right` tries to apply `lem_plus_c` to remove a unit element from the right-hand-side. The double semicolon sequences two tactics together. It applies the second tactic to all (`ON_ALL`) goals produced by the first or applyies a list of tactics one to each generated subgoal (`ON_EACH`).

In step (4) we prove the soundness of the automation. Soundness proofs are typically a major part of the work when developing reflective tactics; however, the compositional nature of $\mathcal{R}_{tac}$ tactics makes proving soundness almost completely automatic. The `rtac_derive_soundness_default` ($\mathcal{L}_{tac}$) tactic proves the soundness of a tactic by composing the soundness of its individual pieces.

Finally, we use `Mcheck` to verify equivalences in the monoid (step (5)). On small problems, the difference between $\mathcal{L}_{tac}$ and our technique is negligible. However, for large problem sizes, our automation performs several orders of magnitude faster. We defer a more detailed evaluation to section **??**.

Building and Evolving Automation While the goal of automation is a "push-button" solution, it rarely starts out that way. The automation shown in Figure 1, like most $\mathcal{R}_{tac}$ automation, was constructed incrementally in much the same way that users build automation using $\mathcal{L}_{tac}$ [**?**]. The developer inspects the goal and finds the next thing to do to make progress. This same process works when developing automation in $\mathcal{R}_{tac}$. When the developer runs a tactic that does not solve the goal, a new goal is returned showing what is left to prove. By default, the process of constructing the syntactic representation is hid-

[3] The full code can be found in the MirrorCore distribution.

```
(* (0) Develop the theory of monoids. *)
Parameter star : ℕ → ℕ → ℕ. (* ``a ⊕ b'' *)
Axiom plus_assoc_c1 : ∀ a b c d, d = a ⊕ (b ⊕ c) → d = (a ⊕ b) ⊕ c.

(* (1) Define a syntax for the problem & setup reification *)
Inductive mon_typ := tyArr (_ _ : mon_typ) | tyProp | tyNat.
Inductive mon_sym := Plus | N (_ : ℕ).
(* ...denotation functions and a few proofs about these types... *)

Let mon_term := expr mon_typ mon_sym.

Reify Declare Syntax reify_mon_term :=
  CFirst (CPatterns patterns_monoid :: ...).

(* (2) Automatically build syntactic lemmas *)
Reify BuildLemma < reify_mon_typ reify_mon_term reify_mon_term >
   lem_plus_assoc_c1 : plus_assoc_c1.
Definition lem_plus_assoc_c1_sound : lemmaD lem_plus_assoc_c1 :=
    plus_assoc_c1.
(* ...more lemmas... *)

(* (3) Build automation using tactics *)
Definition iter_right (n : ℕ) : rtac :=
  REC n (fun rec ⇒
          FIRST [ EAPPLY lem_plus_unit_c
                | EAPPLY lem_plus_assoc_c1 ;; ON_ALL rec
                | EAPPLY lem_plus_assoc_c2 ;; ON_ALL rec
                | EAPPLY lem_plus_cancel ;;
                  ON_EACH [ SOLVE solver | IDTAC ] ])  IDTAC.

Definition iter_left (k : rtac) (n : ℕ) : rtac :=
  REC n (fun rec ⇒
          FIRST [ EAPPLY lem_plus_unit_p
                | EAPPLY lem_plus_assoc_p1 ;; ON_ALL rec
                | EAPPLY lem_plus_assoc_p2 ;; ON_ALL rec
                | k ])  IDTAC.

Definition Mcheck : rtac := ...

(* (4) Prove the automation sound *)
Lemma iter_right_sound : ∀ Q, rtac_sound (iter_right Q).
Proof. unfold iter_right. intros. rtac_derive_soundness_default. Qed.

Lemma iter_left_sound : ∀ Q k, rtac_sound k → rtac_sound (iter_left k Q).
Proof. unfold iter_left. intros. rtac_derive_soundness_default. Qed.

(* (5) Use the reflective automation *)
Goal x ⊕ 2 ⊕ 3 ⊕ 4 = 4 ⊕ 3 ⊕ 2 ⊕ x.
Proof. run_tactic reify_mon_term Mcheck Mcheck_sound. Qed.
```

Fig. 1. Implementing a monoidal cancellation algorithm using $\mathcal{R}_{tac}$.

den from the user and new goals will be returned after they have been converted back into their semantic counter-parts.

It is important to note, that while we can develop tactics incrementally, $\mathcal{R}_{tac}$ is not built to do manual proofs in the style of $\mathcal{L}_{tac}$. When run alone, the core $\mathcal{R}_{tac}$ tactics (e.g. `APPLY`) are often slower than their $\mathcal{L}_{tac}$ counter-parts. $\mathcal{R}_{tac}$'s speed comes from the ability to replace large proof terms with smaller ones, and larger proofs only arise when combining multiple reasoning steps.

4 The MirrorCore Language

A key component of reflective automation is the syntactic representation of the problem domain. We need a representation that is both expressive and easy to extend. In this section we present MirrorCore's generic `expr` language which we used in Section 3.

Mathematically, the `expr` language is the simply-typed λ-calculus augmented with unification variables (see Figure 2). The `expr` language mirrors Coq's core logic, providing a rich structure that can represent higher-order functions and λ-abstractions. To provide extensibility, the language is parametric in both a type of types and a type of symbols. This parameterization allows the client to instantiate the language with domain-specific types, e.g. the monoid carrier, and symbols, e.g. monoid plus. Further, this compartmentalization makes it possible to implement and verify a variety of generic procedures for term manipulation. For example, MirrorCore includes lifting and lowering operations, beta-reduction, and a generic unification algorithm for the `expr` language.

Following the standard presentation of the λ-calculus, the language is divided into two levels: types and terms. The type language is completely user-defined but has two requirements. First, it must have a representation of function types so that λ-abstractions and applications can be typed. Second, it requires decidable equality to ensure that type checking `expr` terms is decidable. In order to use $\mathcal{R}_{tac}$ (which we discuss in Section 5) the type language also requires a representation of Coq's type of propositions (`Prop`).

The term language follows a mostly standard presentation of the λ-calculus using De Bruijn indicesto represent bound variables. To support binders, the denotation function of terms $\left(\begin{smallmatrix} t_u \\ t_v \end{smallmatrix} [\![-]\!]_t\right)$ is parameterized by two type environments (for unification variables, t_u, and regular variables, t_v) and the result type (t). These three pieces of information give us the type of the denotation. Concretely, the meaning of a term is a Coq function from the two environments (typed by $[\![t_u]\!]^{\overrightarrow{\tau}}$ and $[\![t_v]\!]^{\overrightarrow{\tau}}$) to the result type ($[\![t]\!]^{\tau}$). If the term is ill-typed then it has no denotation, which we encode Coq's `option` type. The denotation function returns `Some` with the denotation of the term if it is well-typed, or `None` if the term is ill-typed[4]. This choice means that some of the theorems in MirrorCore rely on the axiom of functional extensionality.

[4] We permit ill-typed terms in our representation to avoid indexing terms by their type. Indexed terms are larger which increases the time it takes to check them thus slowing down the automation.

Types (user specified, with restrictions)

$$\begin{array}{rrcl} & \tau & ::= & \mathsf{tyProp} \mid \tau_1 \to \tau_2 \mid \ldots \\ & [\![-]\!]^{\tau} & : & \tau \to \mathsf{Type} \\ \text{(Environments)} & [\![-]\!]^{\overrightarrow{\tau}} & : & \mathsf{list}\,\tau \to \mathsf{Type} \end{array}$$

$$[\![\mathsf{tyProp}]\!]^{\tau} = \mathsf{Prop} \qquad [\![t_1 \to t_2]\!]^{\tau} = [\![t_1]\!]^{\tau} \to [\![t_2]\!]^{\tau}$$

Domain-specific constants (user specified)

$$\langle - \rangle^{\tau} : b \to \tau \qquad [\![-]\!]_t : b \to \mathsf{option}\,[\![t]\!]^{\tau}$$

Terms

$$\begin{array}{rcl} \mathcal{E} & ::= & e_1\ e_2 \mid \lambda\tau.e \mid x \mid ?u \mid \lceil b \rceil \\ {}^{t_u}_{t_v}[\![-]\!]_t & : & \mathcal{E} \to \mathsf{option}\left([\![t_u]\!]^{\overrightarrow{\tau}} \to [\![t_v]\!]^{\overrightarrow{\tau}} \to [\![t]\!]^{\tau}\right) \end{array}$$

$$\begin{array}{rcll} {}^{t_u}_{t_v}[\![e_1\ e_2]\!]_t & = & \lambda d_u\ d_v.\ \left({}^{t_u}_{t_v}[\![e_1]\!]_{t' \to t}\ d_u\ d_v\right)\ \left({}^{t_u}_{t_v}[\![e_2]\!]_{t'}\ d_u\ d_v\right) & \\ {}^{t_u}_{t_v}[\![\lambda t.e]\!]_{t \to t'} & = & \lambda d_u\ d_v\ a.\ {}^{t_u}_{t_v \cdot t}[\![e]\!]_{t'}\ d_u\ (d_v\ \cdot a) & \\ {}^{t_u}_{t_v}[\![x]\!]_t & = & \lambda \_\ d_v.\ d_v\ x & \text{if } d_v\ x\ :\ [\![t]\!]^{\tau} \\ {}^{t_u}_{t_v}[\![?u]\!]_t & = & \lambda d_u\ \_.\ d_u\ u & \text{if } d_u\ u\ :\ [\![t]\!]^{\tau} \\ {}^{t_u}_{t_v}[\![\lceil b \rceil]\!]_t & = & \lambda \_\ \_.\ [\![b]\!]_t & \end{array}$$

Fig. 2. MirrorCore's calculus for syntactic terms, typing- and denotation functions, using mathematical notation.

The precise placement of the `option` in the type of the denotation function demarks the phase separation between type checking and computing the denotation. This phase separation is necessary to define the meaning of abstractions since the abstraction can only be introduced if the syntactic body has a denotation in the extended environment.

$${}^{t_u}_{t_v}[\![\lambda t'.e]\!]_{t' \to t} = \mathsf{Some}\,\lambda d_u\,d_v.\big(\lambda x.D\,d_u\,(x \cdot d_v)\big) \quad \text{if} \quad {}^{t_u}_{t' \cdot t_v}[\![e]\!]_t = \mathsf{Some}\,D$$

Without knowing that D is well-typed under the extended context, there is no way to construct a value of type $[\![t]\!]^{\tau}$ since not all types in Coq are inhabited.

The `expr` language also includes a representation of unification variables ($?u$), which are place-holders for arbitrary terms. The difficulty in representing unification variables comes when they are mixed with local variables. For example, suppose we represent the following proposition in `expr`

$$?u = ?u \wedge \forall x : \mathbb{N}, ?u = x$$

Focusing on the right-hand conjunct, it seems that we should instantiate $?u$ with x; however, x is not in scope in the left-hand conjunct. We solve this problem in the `expr` language by preventing the instantiation of unification variables from mentioning any locally introduced variables. This choice leads to a more concise representation but requires that we are careful when dealing with unification variables that are scoped with respect to different contexts. We will return to this point in Section 5.

5 $\mathcal{R}_{tac}$: Verification Building Blocks

In this section we implement a fully-reflective proving language, called $\mathcal{R}_{tac}$, modeled on $\mathcal{L}_{tac}$, Coq's built-in tactic language. $\mathcal{R}_{tac}$ allows clients to build completely reflective automation easily without ever needing to write Coq functions that inspect terms. In fact, after the reification step, $\mathcal{R}_{tac}$ almost completely encapsulates the fact that we are using computational reflection at all.

$\mathcal{R}_{tac}$ packages unification variables, premises, and a conclusion, into a "goal" that programs (also called tactics) operate on. Combining these pieces into a simple interface allows $\mathcal{R}_{tac}$ to cleanly export larger granularity operations that rely on multiple pieces. For example, a common operation is to apply a lemma to the conclusion and convert its premises into new goals [**?**]. Doing this requires inspecting the lemma, constructing new unification variables, performing unifications, constructing new goals, and justifying it all using the lemma's proof.

Implementing $\mathcal{R}_{tac}$ requires solving two intertwined problems. In Section 5.1, we describe how we represent and reason about proofs embedded in arbitrary contexts which contain regular variables, unification variables, and propositions. Our reasoning principles allow us to make inferences under these contexts, and evolve the contexts by instantiating unification variables in the context without needing to re-check proofs. In Section 5.2, we present our compositional phrasing of tactic soundness which allows us to easily compose sound tactics to produce sound automation. We close the section (Section 5.3) by discussing $\mathcal{R}_{tac}$'s client-facing interface including a subset of the tactics that we have implemented and an example using them to build a small piece of automation.

5.1 Contexts & Contextualized Proofs

End-to-end, $\mathcal{R}_{tac}$ is about building proof of implications between two propositions. That is, if an $\mathcal{R}_{tac}$ tactic runs on a goal P and returns the goal Q then the soundness of the tactic proves $Q \to P$. However, in order to incrementally construct these proofs, we need to strengthen the specification. What is missing from this global specification is the ability to construct a proof in a context that contains variables, unification variables (possibly with their instantiations), and propositional facts. In $\mathcal{R}_{tac}$, this information is represented by the context data-type defined in Figure 3. The meaning of a context is a function from propositions in the context to propositions outside of the context. That is,

$$\begin{smallmatrix} t_u \\ t_v \end{smallmatrix} [\![c]\!]^{\mathcal{C}} \vdash - : \Big([\![t_u \cdot c_u]\!]^{\overrightarrow{\tau}} \to [\![t_v \cdot c_v]\!]^{\overrightarrow{\tau}} \to \mathsf{Prop} \Big) \to \Big([\![t_u]\!]^{\overrightarrow{\tau}} \to [\![t_v]\!]^{\overrightarrow{\tau}} \to \mathsf{Prop} \Big)$$

where c_u and c_v represent the unification variables and real variables introduced by the context. For exposition purposes, we will consider t_u and t_v to be empty simplifying this definition to the following:

$$[\![c]\!]^{\mathcal{C}} \vdash - : \Big([\![c_u]\!]^{\overrightarrow{\tau}} \to [\![c_v]\!]^{\overrightarrow{\tau}} \to \mathsf{Prop} \Big) \to \mathsf{Prop}$$

$$\text{(Contexts)} \qquad \mathcal{C} ::= \epsilon \mid \mathcal{C}, \forall_\tau \mid \mathcal{C}, \exists_\tau \mid \mathcal{C}, \exists_\tau{=}\mathcal{E} \mid \mathcal{C}, \to \mathcal{E}$$

$$\begin{array}{l}
{}^{t_u}_{t_v}[\![\mathcal{C}]\!] \vdash - \;:\; \left([\![t_u \cdot c_u]\!]^{\overrightarrow{\tau}} \to [\![t_v \cdot c_v]\!]^{\overrightarrow{\tau}} \to \mathsf{Prop}\right) \to [\![t_u]\!]^{\overrightarrow{\tau}} \to [\![t_v]\!]^{\overrightarrow{\tau}} \to \mathsf{Prop} \\
{}^{t_u}_{t_v}[\![\epsilon]\!]^{\mathcal{C}} \vdash P \triangleq P \\
{}^{t_u}_{t_v}[\![c, \to e]\!]^{\mathcal{C}} \vdash P \triangleq {}^{t_u}_{t_v}[\![c]\!]^{\mathcal{C}} \vdash (\lambda d_u\ d_v.\ {}^{t_u}_{t_v}[\![e]\!]_{\mathsf{Prop}}\ d_u\ d_v \to P\ d_u\ d_v) \\
{}^{t_u}_{t_v}[\![c, \forall_t]\!]^{\mathcal{C}} \vdash P \triangleq {}^{t_u}_{t_v \cdot t}[\![c]\!]^{\mathcal{C}} \vdash (\lambda d_u\ d_v.\ \forall x : [\![t]\!]^\tau.\ P\ d_u\ (d_v \cdot x)) \\
{}^{t_u}_{t_v}[\![c, \exists_t]\!]^{\mathcal{C}} \vdash P \triangleq {}^{t_u \cdot t}_{t_v}[\![c]\!]^{\mathcal{C}} \vdash (\lambda d_u\ d_v.\ \forall x : [\![t]\!]^\tau.\ P\ (d_u \cdot x)\ d_v) \\
{}^{t_u}_{t_v}[\![c, \exists_t{=}e]\!]^{\mathcal{C}} \vdash P \triangleq {}^{t_u \cdot t}_{t_v}[\![c]\!]^{\mathcal{C}} \vdash (\lambda d_u\ d_v.\ \forall x : [\![t]\!]^\tau.\ x = {}^{t_u}_{t_v}[\![e]\!]_t\ d_u\ d_v \to P\ (d_u \cdot x)\ d_v)
\end{array}$$

Fig. 3. The definition and denotation of contexts. The denotation of the existential quantifier as a universal quantifier captures the parametricity necessary for local proofs.

Since it is quite common to work within these contexts, we will subscripts to mean pointwise lifting. For example,

$$P \to_c Q \triangleq \lambda d_u\ d_v.\ P\ d_u\ d_v \to Q\ d_u\ d_v$$

The intuitive interpretation of contexts, denoting unification variables as existential quantifiers, captures that property that they prove, but this interpretation is not sufficient for compositional proofs. To see why, consider a proof of $[\![c]\!]^{\mathcal{C}} \vdash P$ and $[\![c]\!]^{\mathcal{C}} \vdash Q$. We would like to combine these two proofs into a proof of $[\![c]\!]^{\mathcal{C}} \vdash (P \wedge_c Q)$ but we can not because the two proofs may make contradictory choices for existentially quantified values. For example, if c is $\exists x : \mathbb{N}$, P is $x = 0$, and Q is $x = 1$ both proofs exist independently by picking the appropriate value of x but the two do not compose. To solve this problem, we use the parametric interpretation of contexts defined in Figure 3 where unification variables are interpreted as universal quantifiers with an equation if they are instantiated. This interpretation captures the parametricity necessary to the proofs by ensuring that proofs that do not constrain the values of unification variables hold for *any* well-typed instantiation.

The parametric interpretation provides us with several, powerful, reasoning principles for constructing and composing proofs in contexts. The first two are related to the applicative nature of the parametric interpretation of contexts.

$$\begin{array}{rl}
\mathsf{ap} : & \forall c\, P\, Q, [\![c]\!]^{\mathcal{C}} \vdash (P \to_c Q) \to [\![c]\!]^{\mathcal{C}} \vdash P \to [\![c]\!]^{\mathcal{C}} \vdash Q \\
\mathsf{pure} : & \forall c\, P, P \to [\![c]\!]^{\mathcal{C}} \vdash P
\end{array}$$

Leveraging these two definitions, we can perform logical reasoning under a context. For example, we can use ap to prove modus ponens in an arbitrary context.

$$\to\text{-E} : \forall c\, P\, Q, [\![c]\!]^{\mathcal{C}} \vdash (P \to_c Q) \to [\![c]\!]^{\mathcal{C}} \vdash Q \to [\![c]\!]^{\mathcal{C}} \vdash Q$$

Similar rules hold for proving conjunctions and disjunctions under arbitrary contexts.

The final reasoning principle for contexts comes when using facts that occur in the premises. The following proof rule allows us to show that facts in the

context ($p \in c$) are provable in the logic.

$$\mathsf{assumption} : \forall c\, p, p \in c \to [\![c]\!]^{\mathcal{C}} \vdash [\![p]\!]$$

Context Morphisms In addition to reasoning parametrically in a context, it is also necessary to evolve contexts by instantiating unification variables. Context morphisms capture the property that any reasoning done under the weaker context is also valid under the stronger context. The following definition captures this transport property which holds for any two contexts c and c' where c' is an evolution of c, written $c \rightsquigarrow c'$.

$$c \rightsquigarrow c' \quad \triangleq \quad \forall P,\ [\![c]\!]^{\mathcal{C}} \vdash P \to [\![c']\!]^{\mathcal{C}} \vdash P$$

The core rule for context evolution is the one for instantiating a unification variable.

$$c, \exists_\tau \rightsquigarrow c, \exists_\tau{=}e \qquad \text{if } {}^{c_u}_{c_v}[\![e]\!]_\tau \text{ is defined}$$

The proof of this rule is trivial since the context on the right simply introduces an additional equation that is not necessary when appealing to the assumption. In addition to instantiating unification variables, context evolution is both reflexive and transitive which allows us to talk about zero or multiple context updates in a uniform way. In addition, context evolution satisfies the natural structural rules allowing updates of any piece of the context. For example,

$$c \rightsquigarrow c' \quad \to \quad c, \forall_\tau \rightsquigarrow c', \forall_\tau \qquad \text{and} \qquad c \rightsquigarrow c' \quad \to \quad c, \exists_\tau \rightsquigarrow c', \exists_\tau$$

Similar rules hold for all of the context constructors and all follow straightforwardly from the definition of $\rightsquigarrow$.

5.2 Implementing $\mathcal{R}_{tac}$

The contextual reasoning principles from the previous section form the heart of the proof theory of $\mathcal{R}_{tac}$. In this section, we describe the generic language constructs and their soundness criteria.

From a language point of view, tactics operate on a single goal; however, tactics can produce multiple goals, for example when proving $P \wedge Q$, it is common to break the goal into two subgoals, one for P and one for Q. Further, while these goals may start with the same context, further reasoning may lead them to extend their contexts in different ways. In order to represent all of the goals in a meaningful way, $\mathcal{R}_{tac}$ uses goal trees defined in Figure 4. All of the syntax maps naturally to the corresponding semantic counter-parts.

On top of goal trees and the contexts from the previous section, we define the two language constructs in $\mathcal{R}_{tac}$: tactics and tactic continuations.

$$\begin{aligned} \text{(Tactics)} \qquad & \mathsf{rtac} \triangleq \mathcal{C} \to \mathcal{E} \to \mathsf{option}\,(\mathcal{C} \times \mathcal{G}) \\ \text{(Tactic continuations)} \qquad & \mathsf{rtacK} \triangleq \mathcal{C} \to \mathcal{G} \to \mathsf{option}\,(\mathcal{C} \times \mathcal{G}) \end{aligned}$$

At the high level, the two constructs accept a context and a representation of the goal—a single expression in the case of tactics and a full goal tree in the case

$$\text{(Goal Trees)} \quad \mathcal{G} ::= \top \mid \lceil \mathcal{E} \rceil \mid \forall \tau . \mathcal{G} \mid \exists \tau . \mathcal{G} \mid \exists \tau = \mathcal{E} . \mathcal{G} \mid \mathcal{E} \to \mathcal{G} \mid \mathcal{G} \wedge \mathcal{G}$$

$$\begin{array}{rl}
{}^{t_u}_{t_v}[\![\top]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, \top \\
{}^{t_u}_{t_v}[\![\lceil e \rceil]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, {}^{t_u}_{t_v}[\![e]\!]_{\mathsf{Prop}}\, d_u\, d_v \\
{}^{t_u}_{t_v}[\![g_1 \wedge g_2]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, {}^{t_u}_{t_v}[\![g_1]\!]^{\mathcal{G}}\, d_u\, d_v \wedge {}^{t_u}_{t_v}[\![g_2]\!]^{\mathcal{G}}\, d_u\, d_v \\
{}^{t_u}_{t_v}[\![e \to g]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, {}^{t_u}_{t_v}[\![e]\!]_{\mathsf{Prop}}\, d_u\, d_v \to {}^{t_u}_{t_v}[\![g_2]\!]^{\mathcal{G}}\, d_u\, d_v \\
{}^{t_u}_{t_v}[\![\forall_\tau . g]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, \forall x : \tau, {}^{t_u}_{t_v \cdot \tau}[\![g]\!]^{\mathcal{G}}\, d_u\, (d_v \cdot x) \\
{}^{t_u}_{t_v}[\![\exists_\tau . g]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, \exists x : \tau, {}^{t_u \cdot \tau}_{t_v}[\![g]\!]^{\mathcal{G}}\, (d_u \cdot x)\, d_v \\
{}^{t_u}_{t_v}[\![\exists_\tau = e . g]\!]^{\mathcal{G}} & \triangleq \lambda d_u\, d_v .\, \exists x : \tau, x = {}^{t_u}_{t_v}[\![e]\!]_\tau \wedge {}^{t_u \cdot \tau}_{t_v}[\![g]\!]^{\mathcal{G}}\, (d_u \cdot x)\, d_v
\end{array}$$

Fig. 4. Goal trees represent the global proof structure.

of tactic continuations—and produce a new context and a goal tree. The option in the return type allows tactics to fail, which is convenient when a particular reasoning step does not apply to the current goal. We discuss this in more detail in Section 5.3.

An $\mathcal{R}_{tac}$ tactic (resp. tactic continuation) is sound if, when it succeeds, the resulting goal tree is sufficient to prove the original goal (resp. goal tree) in the *resulting* context and the resulting context is a consistent extension of the original context. Mathematically[5],

$$\mathsf{rtac\_sound}\, tac \triangleq \begin{cases} \forall c\, e\, g'\, c', tac\ c\ e = \mathsf{Some}\,(c', g') \to \\ \quad c \Rightarrow c' \wedge \\ \quad [\![c']\!]^{\mathcal{C}} \vdash \left({}^{c_u}_{c_v}[\![g']\!]^{\mathcal{G}} \to_c {}^{c_u}_{c_v}[\![e]\!]_{\mathsf{Prop}}\right) \end{cases}$$

where $c \Rightarrow c'$ states that c' is a consistent extension of the context c at all levels. For example,

$$c, \forall_\tau \Rightarrow c', \forall_\tau \quad \leftrightarrow \quad c \Rightarrow c' \wedge (c, \forall_\tau) \rightsquigarrow (c', \forall_\tau)$$

This stronger consistency definition is necessary when we need to escape from under a potentially inconsistent context. For example, when we apply a tactic under a universal quantifier, we shift the quantifier into the context and invoke the tactic with the enlarged context. Suppose that the soundness theorem of the tactic only guaranteed $c, \forall_\tau \Rightarrow c', \forall_\tau$, in order our out tactic to be correct, we must guarantee $c \Rightarrow c'$, but this does not follow. Informally, the consistent evolution of the smaller context should follow, but we can not argue this within Coq because we can not construct a value of type τ. The soundness of tactics follows directly from the soundness of tactic continuations by using the denotation of goal trees rather than the denotation of terms in the conclusion. Formally,

$$\mathsf{rtacK\_sound}\, tac \triangleq \begin{cases} \forall c\, e\, g'\, c', tac\ c\ g = \mathsf{Some}\,(c', g') \to \\ \quad c \Rightarrow c' \wedge \\ \quad [\![c']\!]^{\mathcal{C}} \vdash \left({}^{c_u}_{c_v}[\![g']\!]^{\mathcal{G}} \to_c {}^{c_u}_{c_v}[\![g]\!]^{\mathcal{G}}\right) \end{cases}$$

[5] In this definition we avoid the complexities of ill-typed terms. In the code, the soundness proof gets to assume that the context and goal are both well-typed.

Local Unification Variables The alternation of universal and existential quantifiers in contexts leads to some complexities when manipulating unification variables. As we mentioned previously, unification variables in MirrorCore implicitly have a single, global scope. This choice is at odds with the potential alternation of universally quantified variables and unification variables.

In $\mathcal{R}_{tac}$ we solve the scoping problem using the context. Unification variables introduced in the context are only allowed to mention variables and unification variables that are introduced below it. For example, in '$c, \exists_\tau{=}e$', e is only allowed to mention variables variables and unification variables introduced by c.

This design choice comes with a deficiency when introducing multiple unification variables. For example, if we wish to introduce multiple unification variables, we need to pick an order of those unification variables and the choice is important because we can not instantiate an earlier unification variable using a later one. While there can never be cycles, the order that we pick is significant. Our solution is to introduce mutually recursive blocks of unification variables simultaneously. The reasoning principles for these blocks are quite similar to the reasoning principles that we presented in this section and the last, but there is a bit more bookkeeping involved.

Goal Minimization This simplification does have a benefit for $\mathcal{R}_{tac}$. In particular, it allows us to precisely control the life-time of unification variables which allows us to substitute instantiated unification variables and entirely remove them from the goal tree. For example, the following rules state how we shift unification variables from the context into the goal.

$$[\![c, \exists_t]\!]^{\mathcal{C}} \vdash g \leftrightarrow [\![c]\!]^{\mathcal{C}} \vdash \exists_t, g \quad \text{and} \quad [\![c, \exists_t{=}\, e]\!]^{\mathcal{C}} \vdash g \leftrightarrow [\![c]\!]^{\mathcal{C}} \vdash g[e]$$

where $g[e]$ substitutes all occurrences to the top unification variable with the expression e and renumbers the remaining unification variables appropriately.

In addition to substitution of unification variables, we can also drop hypotheses and universal quantifiers on solved goals. For example,

$$[\![c, e]\!]^{\mathcal{C}} \vdash \top \;\leftrightarrow\; [\![c]\!]^{\mathcal{C}} \vdash \top \qquad \text{and} \qquad [\![c, \forall_t]\!]^{\mathcal{C}} \vdash \top \;\leftrightarrow\; [\![c]\!]^{\mathcal{C}} \vdash \top$$

and contract conjunctions of solved goals, e.g.

$$[\![c]\!]^{\mathcal{C}} \vdash \top \wedge g \;\leftrightarrow\; [\![c]\!]^{\mathcal{C}} \vdash g \qquad \text{and} \qquad [\![c]\!]^{\mathcal{C}} \vdash g \wedge \top \;\leftrightarrow\; [\![c]\!]^{\mathcal{C}} \vdash g$$

5.3 The Core Tactics

With the specification and verification strategy for tactics fleshed out, we return to the client-level. Figure 5 presents a subset of the core tactics that we implemented for $\mathcal{R}_{tac}$. While relatively small in number, the uniform interface of these tactics makes it easy to combine these tactics into higher-level automation. Further, the soundness proofs of tactics built from verified tactics is essentially free. In this section we present the soundness theorems for several representative tactics and show how they compose.

Search Tactics

IDTAC	: rtac	do nothing
FAIL	: rtac	fail immediately
REC	: $\mathbb{N} \to (\mathsf{rtac} \to \mathsf{rtac}) \to \mathsf{rtac} \to \mathsf{rtac}$	bounded recursion
SOLVE	: rtac → rtac	solve fully or fail
FIRST	: list rtac → rtac	first to succeed
AT_GOAL	: $(\mathcal{E} \to \mathsf{rtac}) \to \mathsf{rtac}$	
THEN	: rtac → rtacK → rtac	sequencing
THENK	: rtacK → rtacK → rtacK	sequencing
ON_ALL	: rtac → rtacK	
ON_EACH	: list rtac → rtacK	
MINIFY	: rtacK	reduce the goal size

Reasoning Tactics

EAPPLY	: lemma → rtac	apply a lemma
INTRO	: rtac	introduce a quantifier
EEXISTS	: rtac	witness an existential
SIMPL	: $(\mathcal{E} \to \mathcal{E}) \to \mathsf{rtac}$	compute in the goal
EASSUMPTION	: rtac	use an assumption

Fig. 5. Select $\mathcal{R}_{tac}$ tactics.

Select Soundness Theorems The soundness theorems for individual tactics are almost completely type-directed. For example, the soundness theorem for the REC tactic is the following:

$$\begin{array}{l}\mathtt{REC\_sound} : \forall n\ f\ t,\\ \quad (\forall x, \mathsf{rtac\_sound}\ x \to \mathsf{rtac\_sound}(f\ x)) \to \mathsf{rtac\_sound}\ t \to \mathsf{rtac\_sound}(\mathtt{REC}\ n\ f\ t)\end{array}$$

Similarly, the FIRST tactic allows clients to try a variety of alternatives selecting the first tactic that succeeds. Its soundness simply requires that all of the given tactics are sound.

The APPLY tactic is the work-horse of $\mathcal{R}_{tac}$ and exemplifies MirrorCore's separation of manipulation and meaning. The tactic is parameterized by a syntactic representation of a lemma and attempts to apply it to the goal.

$$\mathtt{APPLY\_sound} : \forall lem, [\![lem]\!]_{\mathsf{lemma}} \to \mathsf{rtac\_sound}\ (\mathtt{APPLY}\ lem)$$

where $[\![-]\!]_{\mathsf{lemma}}$ is the denotation function for lemmas which are defined as triples containing the types of universally quantified variables, a list of premises and the conclusion. The implementation of APPLY is quite complex; however, all of the complexity can be hidden behind the soundness of the tactic making it trivial for clients to use APPLY.

AT_GOAL allows automation to inspect the goal before choosing what to do. For example, recall the use of REC to iterate through the terms on each side of the equality in Figure 1. Rather than picking an arbitrary recursion depth, we can use AT_GOAL to inspect the goal and compute an adequate depth for the fixpoint. The soundness rule for AT_GOAL simply requires that the function

```
Def tac_even : rtac := REPEAT 10
  (FIRST [ ASSUMPTION
         | APPLY even_0_syn
         | APPLY even_odd_syn
         | APPLY odd_even_syn ]).

Thm tac_even_sound : rtac_sound tac_even.
  apply REPEAT_sound.
  apply FIRST_sound; breakForall.
  - apply ASSUMPTION_sound.
  - apply APPLY_sound; exact even_0.
  - apply APPLY_sound; exact even_odd.
  - apply APPLY_sound; exact odd_even.
Qed.
```

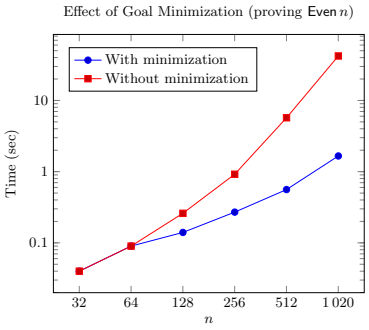


Fig. 6. Simple tactic for proving Evenness and the effect of minimization on run-time.

produces a sound tactic for *any* goal, i.e.

$$\texttt{AT\_GOAL\_sound} : \forall tac, (\forall g, \mathsf{rtac\_sound}\,(tac\,g)) \to \mathsf{rtac\_sound}\,(\texttt{AT\_GOAL}\,tac)$$

The `MINIFY` tactic continuation reduces the size of goals by substituting instantiated unification variables and removing solved branches of the proof. While we could have integrated it into all of the tactics, separating it modularizes the proof. Further, it allows the client to batch several operations before performing minimization thus amortizing the cost over more reasoning steps. A particularly extreme instance of this arises when dealing with very large terms in the substitution. Figure **??** shows a simple tactic for proving `Even`ness reflectively. The manual soundness proof is for presentation purposes only to demonstrate that our `rtac_derive_soundness_default` tactic truely is purely syntax directed.

6 Extending $\mathcal{R}_{tac}$: Exploiting Structure in Rewriting

$\mathcal{R}_{tac}$ allows users to develop goal-oriented automation with a tight coupling between the code we run and the justification of its soundness. In many cases, this is just what the doctor ordered; however, because $\mathcal{R}_{tac}$ is defined within Coq's logic, we can implement custom automation that inter-operates directly with $\mathcal{R}_{tac}$. In this section we present a custom $\mathcal{R}_{tac}$ tactic for setoid rewriting—we will explain what this means in a moment. In addition to being an $\mathcal{R}_{tac}$ tactic, our setoid rewriter is parameterized by lemmas to rewrite with and tactics to discharge the premises of the lemmas.

Writing custom tactics is an advanced topic that is still quite difficult, though we are working on easing the process. The benefit of writing a custom tactic is that we can customize the implementation to exploit additional structure that is present in the problem domain.

Rewriting Formally, rewriting answers the question: "what term is equal to e using a set of equational lemmas?" *Setoid* rewriting generalizes the use of equality in this question to arbitrary relations. For example, the Fiat system for deductive synthesis [**?**] is built around rewriting using refinement relations.

Our rewriter is inspired by Sozeau's `rewrite_strat` tactic [**?**] and handles similar features, including:

- General relations, including both non-reflexive and non-transitive ones,
- Rewriting in function arguments and under binders,
- Hint databases which store the lemmas to use in rewriting, and
- Discharging side-conditions using $\mathcal{R}_{tac}$ tactics.

The reflective implementation of our rewriter allows us to avoid explicitly constructing proof terms which, as we show in Section **??**, results in substantial performance improvements. In addition, exporting the rewriter as a tactic makes it easy to integrate into larger reflective automation.

We start the section off with an explanation of the mechanism underlying setoid rewriting using a simple example (Section **??**). In Section **??** we show how we exploit this structure to make the rewriter more effective. We conclude by presenting the client-facing interface to the rewriter (Section **??**).

6.1 Setoid Rewriting by Example

To demonstrate the rewriting process, consider the following example where we rewrite by an inverse entailment relation ($\dashv$), which is essentially "if":

$$\big(P \wedge \exists x : \mathbb{N}, Q(x+1)\big) \quad \dashv \quad ?_0$$

We are looking for a term to fill in $?_0$ that will make this goal provable. When we are done we will find that $?_0$ can be $\exists x : \mathbb{N}, \big(P \wedge Q(1+x)\big)$ and the entailment will be provable. We will focus on bottom-up rewriting where we rewrite in the sub-terms of an expression before attempting to rewrite the expression itself.

Proper Morphisms Rewriting bottom-up first requires getting to the leaves. To do this we need to determine the relations to use when rewriting sub-terms to ensure that the results fit into the proof. In our example problem, we are looking for relations R_1 and R_2 such that we can combine the proofs of

$$P \; R_1 \; ?_1 \qquad \text{and} \qquad (\exists x : \mathbb{N}, Q\,(x+1)) \; R_2 \; ?_2$$

into a proof of

$$P \wedge \exists x : \mathbb{N}, Q(x+1) \quad \dashv \quad ?_1 \wedge ?_2$$

This information is carried by *properness proofs* such as the following:

$$\mathsf{Proper\_and\_if} : \mathsf{Proper}\,(\dashv \Longrightarrow \dashv \Longrightarrow \dashv) \wedge$$

which means

$$\forall a\, b\, c\, d, (a \dashv b) \to (c \dashv d) \to ((a \wedge c) \dashv (b \wedge d))$$

This lemma tells us that if we use $\dashv$ for both R_1 and R_2, then we will be able to combine the sub-proofs to construct the overall proof. With concrete relations for R_1 and R_2, we can apply rewriting recursively to solve the goals and find appropriate values for $?_1$ and $?_2$.

When the rewriter recurses to solve the first obligation ($P \dashv ?_1$) it finds that there is no explicit proof about P and $\dashv$. However, the reflexivity of the $\dashv$ relation allows the rewriter to use P to instantiate $?_1$, solving the goal. While this may seem obvious, this check is necessary to support rewriting by non-reflexive relations since, for arbitrary relations, there may be no term related to P.

Rewriting on the right-hand side of the conjunction is a bit more interesting. In this case, the head symbol is an existential quantifier, which is represented using a symbol $\exists_{\mathbb{N}}$ applied to an abstraction representing the body, i.e.

$$\exists_{\mathbb{N}}(\lambda x : \mathbb{N}, Q(x+1))$$

At the highest level of syntax things are the same as above, we look up a properness proof for $\exists_{\mathbb{N}}$ and entailment and find the following:

$$\mathsf{Proper\_exists\_if} : \mathsf{Proper}\left((\mathsf{pointwise}\,\mathbb{N}\ \dashv)\implies \dashv\right)\exists_{\mathbb{N}}$$

which means

$$\forall a\,b, (\forall x, a\,x \dashv b\,x) \to (\exists x, a\,x) \dashv (\exists x, b\,x)$$

As above, the conclusion of the lemma exactly matches our goal, so we instantiate $?_2$ with $\exists x, ?_3\,x$ and produce a new rewriting problem to solve $?_3$.

The problem gets a bit more interesting when we rewrite the body at the pointwise relation. The definition of '$\mathsf{pointwise\_relation}\,\mathbb{N}\ \dashv$' makes it clear that we can rewrite in the function body as long as the two bodies are related by $\dashv$ when applied to the same x, so we will shift a universally quantified natural number into our context and begin rewriting in the body.

Rewriting in the rest of the term is similar. The only complexity comes from determining the appropriate morphism for Q. First, if we do not find a morphism for Q, we can still rewrite $Q\,(x+1)$ into $Q\,(x+1)$ by the reflexivity of $\dashv$ but this prevents us from rewriting the addition. The solution is to derive $\mathsf{Proper}\,(=\implies \dashv)\,Q$ by combining the fact that all Coq functions respect equality, i.e. $\mathsf{Proper}\,(=\implies =)\,Q$, and the reflexivity of $\dashv$.

Rewriting on the Way Up Eventually, our rewriter will hit the bottom of the term and begin coming back up. It is during this process that we make use of the actual rewriting lemmas. For example, take applying the commutativity of addition on $x+1$. Our rewriter just solved the recursive relations stating that $x = x$ and $1 = 1$ so we have a proof of $x+1 = x+1$. However, because equality is transitive, we can perform more rewriting here. In particular, the commutativity of addition justifies rewriting $x+1$ into $1+x$.

The new result logically fits into our system but the justification is a bit strained. The recursive rewriting above already picked a value for the unification variable that this sub-problem was solving. Noting this issue, we realize that we

should have appealed to transitivity *before* performing the recursive rewriting. Doing this requires a bit of foresight since blindly applying transitivity could yield in an unprovable goal if the relation is not also reflexive.

With the rewriting in the body completed, we continue to return upward finding no additional rewrites until we get to the top of the goal. At this point, we have proved the following:

$$P \wedge \exists x : \mathbb{N}, Q(x+1) \dashv\ P \wedge \exists x : \mathbb{N}, Q(1+x)$$

But we would like to continue rewriting on the left-hand side of the entailment. This rewriting is justified by the fact that $\dashv$ is a transitive relation. Again sweeping the need for foresight under the rug, we can assume that we wish to solve this rewriting goal:

$$P \wedge \exists x : \mathbb{N}, Q(1+x) \dashv ?'_0$$

Here we can apply the following lemma, which justifies lifting the existential quantifier over the conjunction:

$$\forall a\, b, a \wedge (\exists x : \mathbb{N}, b\, x) \dashv\ (\exists x : \mathbb{N}, a \wedge b\, x)$$

Note that a can not mention x.

After lifting the existential quantifier to the top, our rewriting is complete. The key property to note from the example is that the only symbols that the rewriter needed to interpret where the morphisms, e.g. respectful and pointwise. All other reasoning was justified entirely by a combination of rewriting lemmas, properness lemmas, and the reflexivity and transitivity of relations. Thus, like $\mathcal{R}_{tac}$, our rewriter is parametric in the domain.

6.2 Implementing the Rewriter

There are two opportunities to make custom rewriting more efficient than one implemented using $\mathcal{R}_{tac}$ primitives. First, rewriting tends to produce a lot of unification variables as properness rules have two unification variables for every function argument, only one of which will be solved when applying the theorem. Our small example above would introduce at least 8 unification variables where in reality none are strictly necessary. Second, rewriting needs to be clever about when it appeals to transitivity.

Expressing the rewriter as a function with a richer type allows us to solve both of these problems elegantly. Rather than representing the goal as a proposition relating a unification variable to a known term, we can explicitly carry around the known term and the relation and return the rewritten term. This insight leads us to choose the following type for the rewriter

$$\mathsf{rewriter} \triangleq \mathcal{C} \to \mathcal{E} \to \mathcal{R} \to \mathsf{option}(\mathcal{C} \times \mathcal{E})$$

where $\mathcal{R}$ is the type of relations. The attentive reader will notice the similarity to the type of tactics, which is even more apparent in the soundness criterion:

$$\mathsf{rewrite\_sound}\, rw \triangleq \begin{cases} \forall c\, e\, e'\, c', rw\ c\ e\ r = \mathsf{Some}\,(c', e') \to \\ \quad c \Rightarrow c' \wedge \\ \quad {\scriptstyle t_u \atop t_v}[\![c']\!]^{\mathcal{C}} \vdash (\lambda d_u\, d_v.\, {\scriptstyle t_u \cdot c_u \atop t_v \cdot c_v}[\![e]\!]_t\, d_u\, d_v\ \ [\![r]\!]^{\mathcal{R}}\ \ {\scriptstyle t_u \cdot c_u \atop t_v \cdot c_v}[\![e']\!]_t\, d_u\, d_v) \end{cases}$$

where $[\![-]\!]^{\mathcal{R}}$ is the denotation of relations. As one would expect, the same reasoning principles for contexts apply when verifying the rewriter. Using this representation, we are clearly reducing the number of unification variables since invoking the rewriter no longer requires a unification variable at all.

This encoding also allows us to perform additional processing after our recursive rewrites return. If we use unification variables to return results, we need to ensure that we do not instantiate that unification variable until we are certain that we have our final result. Therefore, when we make recursive calls, we would need to generate fresh unification variables and track them. Communicating the result directly solves this problem elegantly because the rewriter can inspect the results of recursive calls before it constructs its result. The key to justifying this manipulation is that, unlike $\mathcal{R}_{tac}$, the soundness proof of the rewriter gets a global view of the computation *before* it needs to provide a proof term. This gives it the flexibility to apply, or not apply, transitivity based on the entire execution of the function. That is, if multiple rewrites succeed, and the relation is transitive, then the soundness proof uses transitivity to glue the results together. If only one succeeds, there is no need to use transitivity and the proof from the recursive call is used. And if none succeed, and the relation is not reflexive then rewriting fails. It is important to note that this global view is purely a verification-time artifact. It incurs no runtime overhead.

Another benefit of accepting the term directly is that the rewriter can perform recursion on it directly. The core of the bottom-up rewriter handles the cases for the five syntactic constructs of the `expr` language, of which only application and abstraction are interesting. In the application case the rewriter accumulate the list of arguments delaying all of its decisions until it reaches the head symbol. In order to ensure that the rewriter can perform recursive calls on these sub-terms, the rewriter pairs the terms with closures representing the recursive calls on these sub-terms. This technique is reminiscent of hereditary substitutions and makes it quite easy to satisfy Coq's termination checker. Abstractions are the only construct that is treated specially within the core rewriter. For abstractions, we determine whether the relation is a pointwise relation and if so, we shift the variable into the context and make a recursive call. Otherwise, we treat the abstraction as an opaque symbol.

6.3 Instantiating the Rewriter

Figure **??** presents the types and helper functions exported by the rewriter. The interface to the rewriter defines four types of functions: `refl_dec`, `trans_dec`, `properness`, and `rewriter`. `refl_dec` (resp. `trans_dec`) returns true if the given relation is reflexive (resp. transitive). `properness` encodes properness facts and is keyed on an expression, e.g. $\lceil\wedge\rceil$, and a relation, e.g. $\dashv$, and returns suitable relations for each of the arguments, e.g. the list $[\dashv, \dashv]$. $\mathcal{L}_{tac}$'s setoid rewriter implements these three features using Coq's typeclass mechanism and these functions are essentially reified typeclass resolution functions. The `rewriter`, which we discussed in detail in Section **??** is what actually performs rewriting. Each one

Types

$$\begin{aligned}
\texttt{refl_dec} &\triangleq R \to \texttt{bool} && \text{is relation reflexive?}\\
\texttt{trans_dec} &\triangleq R \to \texttt{bool} && \text{is relation transitive?}\\
\texttt{properness} &\triangleq \mathcal{E} \to \mathcal{R} \to \texttt{list}\,\mathcal{R} && \text{get relation for } \mathcal{E} \text{ ending in R}\\
\texttt{rewriter} &\triangleq \mathcal{C} \to \mathcal{E} \to \mathcal{R} \to \texttt{option}\,(\mathcal{C} \times \mathcal{E}) && \text{perform rewrites}
\end{aligned}$$

Builders

$$\begin{aligned}
\texttt{do_proper} &: \texttt{list}\,(\mathcal{E} \times \mathcal{R}) \to \texttt{properness}\\
\texttt{rewrite_db} &: \texttt{list}\,(\texttt{rw_lemma} \times \texttt{rtacK}) \to \texttt{rewriter}\\
\texttt{rw_repeat} &: \texttt{refl_dec} \to \texttt{trans_dec} \to \mathbb{N} \to \texttt{rewriter} \to \texttt{rewriter}\\
\texttt{rw_pre_simplify} &: (\mathcal{E} \to \mathcal{E}) \to \texttt{rewriter} \to \texttt{rewriter}\\
\texttt{rw_post_simplify} &: (\mathcal{E} \to \mathcal{E}) \to \texttt{rewriter} \to \texttt{rewriter}\\
\texttt{bottom_up} &: \texttt{refl_dec} \to \texttt{trans_dec} \to \texttt{properness} \to \texttt{rewriter} \to \texttt{rewriter}\\
\texttt{setoid_rewrite} &: \mathcal{R} \to \texttt{rewriter} \to \texttt{rtac}
\end{aligned}$$

Fig. 7. The interface to the rewriter.

of these types has a corresponding soundness property similar to `rewrite_sound`.

$$\begin{aligned}
\mathsf{refl_dec_sound}\ f &\triangleq \forall r, fr = \texttt{true} \to \texttt{Reflexive}\ [\![R]\!]^{\mathcal{R}}\\
\mathsf{trans_dec_sound}\ f &\triangleq \forall r, fr = \texttt{true} \to \texttt{Transitive}\ [\![R]\!]^{\mathcal{R}}\\
\mathsf{properness_sound}\ f &\triangleq \forall e\, r\, rs, f\, e\, r = \mathsf{Some}\, rs \to \texttt{Proper}\ [\![rs \Longrightarrow r]\!]^{\mathcal{R}}\ {}^{\epsilon}_{\epsilon}[\![e]\!]_t
\end{aligned}$$

where $rs \Longrightarrow r$ builds a respectful morphism from rs. For example, $[r_1, r_2] \Longrightarrow r$ equals $r_1 \Longrightarrow r_2 \Longrightarrow r$.

The helper functions provide simple ways to construct sound values of these types from simpler pieces. As with the $\mathcal{R}_{tac}$ tactics, their soundness theorems are essentially all type-directed. For example, `do_proper` constructs a `properness` from a list of expressions and relations where each expression is proper with respect to the relation. Formally,

$$\begin{aligned}
&\texttt{do_proper_sound} : \forall er,\\
&\quad \texttt{Forall}(\lambda(e, r).\mathsf{Proper}\ [\![r]\!]^{\mathcal{R}}\ {}^{\epsilon}_{\epsilon}[\![e]\!]_{e_t})\, er \to \mathsf{properness_sound}\,(\texttt{do_proper}\, er)
\end{aligned}$$

where e_t represents the type of e. The current implementation looks up the expression and relation in the list and returns a list of the relations that the arguments must respect; however, more sophisticated implementations could use discrimination trees keyed on either the expression or the relation.

`rewrite_db` is similar in many regards to `APPLY` in $\mathcal{R}_{tac}$. It takes a list of rewriting lemmas (`rw_lemma`), which are specialized lemmas that have conclusions of the type $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$, and tactic continuations used to solve the premises and builds a rewriter that tries to rewrite with each lemma sequentially. The soundness theorem is similar to the soundness of `do_proper`.

$$\begin{aligned}
&\texttt{rewrite_db_sound} : \forall r,\\
&\texttt{Forall}(\lambda(l, t).[\![l]\!]_{\mathsf{rw_lemma}} \wedge \mathsf{rtacK_sound}\, t)\, r \to \mathsf{rewrite_sound}\,(\texttt{rewrite_db}\, r)
\end{aligned}$$

The `rw_pre_simplify` and `rw_post_simplify` tactics require that the function argument produces a term that is equal to the input term and are useful

when we need to reduce terms to put them back into a normal form. Finally, the `setoid_rewrite` tactic converts a rewriter into an $\mathcal{R}_{tac}$ tactic. Due to the nature of rewriting in the goal, the relation to rewrite by must be a sub-relation of reverse implication. Formally,

$$\texttt{setoid\_rewrite\_sound} : \forall r\, w,\\ (\llbracket r \rrbracket^{\mathcal{R}} \subseteq \rightarrow) \rightarrow \mathsf{rewrite\_sound}\, w \rightarrow \mathsf{rtac\_sound}\, (\texttt{setoid\_rewrite}\, r\, w)$$

7 Case Studies

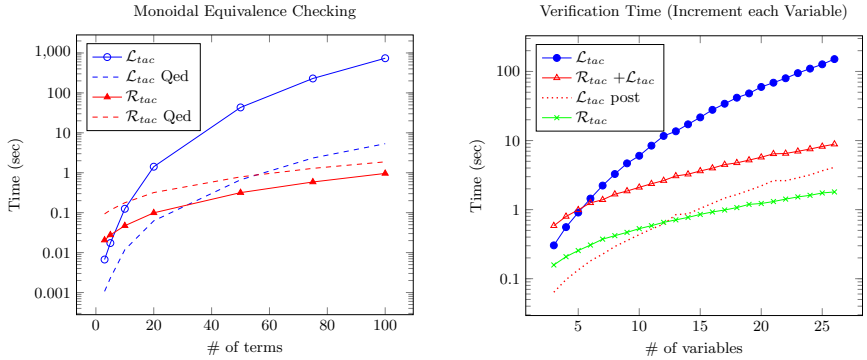
Using $\mathcal{R}_{tac}$, we have built several pieces of automation that perform interesting reasoning and have completely automatic proofs. Beyond these small case studies, $\mathcal{R}_{tac}$ has also been used for automating a larger program logic [**?**] and we are currently using it to verify Java programs with Charge! [**?**].

We performed our evaluation on a 2.7Ghz Core i7 running Linux and Coq 8.5rc1. Our benchmarks compare our reflective automation to similar $\mathcal{L}_{tac}$ automation. Unless otherwise noted, both implementations use the same algorithms and the $\mathcal{R}_{tac}$ implementations use only the generic tactics. We benchmark two phases: proof generation and proof checking (notated by -Qed). In the $\mathcal{L}_{tac}$ implementations, proof generation is the time it takes to interpret the tactics and construct the proof object, and the *Qed* time is the time it takes Coq to check the final proof. Note that when using $\mathcal{L}_{tac}$, the proof checking does *not* include the search necessary to find the proof. In the $\mathcal{R}_{tac}$ implementation, proof generation counts the cost to construct the syntactic representation of the goal and perform the computation on Coq's byte-code virtual machine [**?**]. During Qed time the syntactic representation is type-checked, its denotation is computed and checked to be equal to the proposition that needs to be checked, and the tactic is re-executed to ensure that the computation is correct.

Monoidal Equivalence Our first case study is the monoidal equivalence checker from Section 3. The graph in Figure **??** shows how the $\mathcal{R}_{tac}$ implementation scales compared to the $\mathcal{L}_{tac}$ implementation. Despite the fact that both the $\mathcal{R}_{tac}$ and the $\mathcal{L}_{tac}$ automation perform exactly the same search, the $\mathcal{R}_{tac}$ implementation scales significantly better than the $\mathcal{L}_{tac}$ implementation. The break-even point–where $\mathcal{R}_{tac}$ and $\mathcal{L}_{tac}$ are equally fast—is at roughly 8 terms where the proof size begins to increase dramatically compared to the problem size.

Even the `Qed` for $\mathcal{R}_{tac}$ becomes faster than the `Qed` for $\mathcal{L}_{tac}$, though this happens only for much larger problems. The intersection point for the `Qed` lines corresponds to the size of the problem where re-performing the entire search and type checking the syntactic problem becomes faster than checking just the final proof term. Memoizing the correct choices made during execution of the tactic in the style of Cybele [**?**] could further decrease `Qed` time. However, doing this would require embedding the simulation into the final proof.

Post-condition Verification We developed a simple program verifier in $\mathcal{R}_{tac}$ for a trivial imperative programming language, which includes assignments, addition,



(a) (b)

Fig. 8. Performance of (a) monoidal equivalence checking, and (b) a simple imperative program verifier.

and sequencing. All of the automation is based heavily on a simple $\mathcal{L}_{tac}$ implementation. For example, the lemmas that we reify are the same lemmas that are applied by the $\mathcal{L}_{tac}$ implementation. We built the automation incrementally, first automating only sequencing and copies and later integrating handling of simple reasoning about addition.

The graph in Figure **??** compares the performance between pure $\mathcal{L}_{tac}$ ($\mathcal{L}_{tac}$), pure $\mathcal{R}_{tac}$ ($\mathcal{R}_{tac}$), and a hybrid of $\mathcal{L}_{tac}$ and $\mathcal{R}_{tac}$ ($\mathcal{L}_{tac}$+$\mathcal{R}_{tac}$) implementation. The problem instances increment each of n variables with the post-condition stating that each variable has been incremented. The x axis shows the number of variables (which is directly related to both the size of the program and the pre- and post-conditions) and the y axis is the verification time in seconds. There are two sub-problems in the verifier: first, the post-condition needs to be computed, and second, the entailment between the computed post-condition and the stated post-condition is checked. The blue ($\mathcal{L}_{tac}$) line automates both sub-problems in $\mathcal{L}_{tac}$. Converting the post-condition calculation tactic to $\mathcal{R}_{tac}$ and leaving the entailment checking to $\mathcal{L}_{tac}$ already gets us a substantial performance improvement for larger problems, e.g. a 31x reduction in verification time for 26 variables. The red dotted line ($\mathcal{L}_{tac}$ post) shows the amount of that time that is spent checking the final entailment in $\mathcal{L}_{tac}$. Converting the entailment checker into a reflective procedure results in another 2.6x speedup bringing the tactic to the green line at the bottom of the graph. Overall, the translation from pure $\mathcal{L}_{tac}$ to pure $\mathcal{R}_{tac}$ leads to an almost 84x reduction in the total verification time from 151 seconds to less than 2. In addition to good performance on both sub-problems, solving the entire verification with a single reflective tactic avoids the need to leave the syntactic representation which is often accounts for a large portion of the time in reflective automation [**?**].

For this simple language, the entire translation from $\mathcal{L}_{tac}$ to $\mathcal{R}_{tac}$ took a little over a day. We encountered three main stumbling blocks in our development. First, the meta-theory for our language is built on the Charge! library [**?**]

```
Def the_rewrites : rewriter :=
   rw_post_simplify simple_reduce
      (rw_pre_simplify beta (rewrite_db the_lemmas)).

Def pull_all_quant : rewriter :=
  rw_repeat is_refl is_trans 300
    (bottom_up is_refl is_trans
       get_proper_only_all_ex
     the_rewrites).

Def quant_pull : rewriter :=
  bottom_up is_refl is_trans
    get_proper pull_all_quant.

Def qp_tac : rtac :=
  setoid_rewrite quant_pull.
```

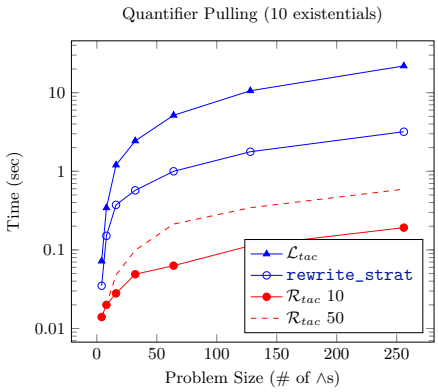


Fig. 9. Tasking the rewriter to lift quantifiers; and its scaling properties.

which relies heavily on type-classes which are not automatically handled by MirrorCore and $\mathcal{R}_{tac}$. Second, solving the entailment requires reasoning about arithmetic. In $\mathcal{L}_{tac}$, we can use Coq's built-in `omega` tactic which discharges obligations in Presburger arithmetic. Since we are building our automation reflectively, we lose access to generate-and-check-style automation. We solve this problem by writing custom $\mathcal{R}_{tac}$ to discharge the regular form of the goals that we need for this example, but more robust automation is clearly preferable and would likely benefit many clients. Finally, reasoning steps in $\mathcal{L}_{tac}$ rely on Coq's reduction mechanism. Since computational reflection treats symbols opaquely by default we needed to write reflective unfolders which replace symbols with their implementations. This is currently a quite manual task, though we believe that is could be simplified with some additional development.

For exposition purposes we have purposefully kept the automation simple. Adding support for additional language constructs is quite simple assuming they have good reasoning principles. In general, we reify the lemma and add new arms to the `FIRST` tactic to apply them. To extend the automation to a more realistic language we need to adapt the language and the automation to support heap operations and a rich separation logic. Our initial work on this suggests that this extension crucially relies on a good separation logic entailment checker. We believe that our monoidal equivalence checker is a first step in this direction, but more work is necessary to handle abstract representation predicates and more clever solving of unification variables that represent frame conditions.

Quantifier Pulling with Rewriting To demonstrate the rewriter, we use it to lift existential quantifiers over conjunctions as we did in Section **??**. Lifting of this

sort is a common operation when verifying interesting programs since existential quantifiers are often buried inside representation predicates.

To perform lifting, we instantiate the rewriter using the definitions in Figure **??**. The core traversal is `quant_pull` which uses the bottom-up traversal. When the head symbol is a conjunction the recursive rewrites pull quantifiers to the top, which produces a formula similar to: $(\exists x : \mathbb{N}, P\,x) \wedge (\exists y : \mathbb{N}, \exists z : \mathbb{N}, Q\,y\,z)$. To lift *all* of the quantifiers from *both* sides of the conjunct, `pull_all_quant` repeatedly performs rewriting to lift the existentials over the conjunct. This rewriting also uses the bottom-up rewriter, but only uses the properness lemmas that allow the rewriter to descend into the body of existential quantifiers (`get_proper_only_all_ex`) to avoid descending into terms that we know do not contain quantifiers.

Figure **??** compares our reflective rewriter with two non-reflective strategies on a problem where 10 existential quantifiers are lifted from the leaves of a tree of conjuncts. The $\mathcal{L}_{tac}$ rewriter uses `repeat setoid_rewrite` since Coq's built-in `autorewrite` tactic does not rewrite under binders which is necessary to complete the problem. `rewrite_strat` uses Sozeau's new rewriter [**?**] which is more customizable and produces better proof terms. Even on small instances, the reflective rewriter is faster than both $\mathcal{L}_{tac}$ and `rewrite_strat`, e.g. at 16 conjuncts the reflective implementation is 13x faster than the strategy rewriter and 42x faster than the $\mathcal{L}_{tac}$ implementation. The performance improvement is due to the large proof terms that setoid rewriting requires.

8 Related & Future Work

The idea of computational reflection has been around since Nuprl [**?**,**?**] and also arose later in LEGO [**?**]. Besson [**?**] first demonstrated the ideas in Coq reasoning about Peano arithmetic. Since then there have been a variety of procedures that use computational reflection. Braibant's AAC tactics [**?**] perform reflective reasoning on associative and commutative structures such as rings. Lescuyer [**?**] developed a simple, reflective, SMT solver in Coq. There has also been work on reflectively checking proof traces produced by external tools in Coq [**?**,**?**]. The development of powerful reflective tactics has been facilitated by fast reduction mechanisms in Coq [**?**,**?**]. Our work tackles the problem of making computational reflection more accessible to ordinary users of Coq by making it easier to write and prove reflective automation sound, which is the main bottleneck in using computational reflection in practice.

Our work builds on the ideas for computational reflection developed by Malecha et al [**?**]. We extend that work by supporting a richer term and type language that is capable of representing higher-order problems and problems with local quantification[6] using a formalized version of Garillot's representation of simple types in type theory [**?**]. Our work differs from Garillot's by applying

[6] Malecha's work supports local quantification only in separation logic formulae and relies crucially on the type of separation logic formulae to be inhabited.

the representation to build reusable reflective automation. Some of the technical content in this paper is expanded in Malecha's dissertation [**?**].

Work by Keller describes an embedding of HOL lite in Coq [**?**] in order to transfer HOL proofs to Coq. Our representation in MirrorCore is very close to their work since, like MirrorCore, HOL lite does not support dependent types. Their work even discusses some of the challenges in compressing the terms that they import. For us, computational reflection solves the problem of large proof terms, though we did have to tune our representation to shrink proof terms. Separately, Fallenstein and Kumar [**?**] have applied reflection in HOL focusing on building self-evolving systems using logics based on large cardinals.

Recent work [**?**,**?**] has built reflective tactics in the Agda proof assistant. Unlike our work, this work axiomatizes the denotation function for syntax and relies on these axioms to prove the soundness of tactics. They argue that this axiomatization is reasonable by restricting it to only reduce on values in some cases which is sufficient for computational reflection. The reason for axiomatizing the denotation function is to avoid the overwhelming complexity of embedding a language as rich as dependent type theory within itself [**?**,**?**,**?**,**?**,**?**,**?**].

Kokke and Swierstra [**?**] have developed an implementation of `auto` in Agda using Agda's support for computational reflection. Their work also includes a reflective implementation of unification similar to our own. Their implementation abstracts over the search strategy allowing them to support heuristic search strategies, e.g. breadth- and depth-first search. Developing their procedure using only $\mathcal{R}_{tac}$ would be difficult because $\mathcal{R}_{tac}$'s design follows $\mathcal{L}_{tac}$'s model and only supports depth-first search. However, we can still implement a custom tactic similar to our rewriting tactic that interfaces with $\mathcal{R}_{tac}$.

Beyond computational reflection, there has been a growing body of work on proof engineering both in Coq and other proof assistants. Ziliani developed $\mathcal{M}_{tac}$ [**?**], a proof-generating tactic language that manipulates proofs explicitly. While it does generate proofs and thus is not truly reflective, it does provide a cleaner way to develop proofs. Prior to $\mathcal{M}_{tac}$, Gonthier [**?**] demonstrated how to use Coq's canonical structures to approximate disciplined proof search. Canonical structures are the workhorse of automation in the SSreflect tactic library [**?**] which uses them as "small scale reflection." Our approach is based on large-scale computational reflection, seeking to integrate reflective automation to build automation capable of combining many steps of reasoning.

Escaping some of the complexity of dependent type theories, Stampoulis's VeriML [**?**] provides a way to write verified tactics within a simpler type theory. Like $\mathcal{R}_{tac}$, VeriML tactics are verified once meaning that there results do not need to be checked each time. VeriML is an entirely new proof assistant with accompanying meta-theory. Our implementation of $\mathcal{R}_{tac}$ is built directly within Coq and is being used alongside Coq's rich dependent type theory.

Future Work The primary limitation in MirrorCore is the fact that the term representation supports only simple types. For the time being we have been able to get around this limitation by using meta-level parameterization to represent, and reason about, type constructors, polymorphism, and dependent types. En-

riching MirrorCore to support these features in a first-class way would make it easier to write generic automation that can reason about these features, for example applying polymorphic lemmas. The primary limitation to doing this comes from the need to have decidable type checking for the term language. With simple types, decidable equality is simply syntactic equality, but when the type language contains functions type checking requires reduction.

Currently, the main cost of switching to reflective tactics is the loss of powerful tactics such as `omega` and `psatz`. While pieces of these tactics are reflective, integrating them into larger developments requires that they use with extensible representations. Porting these tactics to work on MirrorCore would be a step towards making them usable within $\mathcal{R}_{tac}$. Unfortunately, many of these plugins rely on interesting OCaml programs to do the proof search and then emit witnesses that are checked reflectively. Accommodating this kind of computation within $\mathcal{R}_{tac}$ would require native support for invoking external procedures and reconstructing the results in Coq *a la* Claret's work [**?**].

9 Conclusion

We built a framework for easily building efficient automation in Coq using computational reflection. Our framework consists of MirrorCore, an extensible embedding of the simply-typed λ-calculus inside of Coq, which we use as our core syntactic representation. On top of MirrorCore, we built the reflective tactic language $\mathcal{R}_{tac}$. $\mathcal{R}_{tac}$ is modeled on $\mathcal{L}_{tac}$ which makes it easy to port simple $\mathcal{L}_{tac}$ tactics directly to $\mathcal{R}_{tac}$. Our case studies show that even naïve $\mathcal{R}_{tac}$ implementations can be nearly 2 orders of magnitude faster than their corresponding $\mathcal{L}_{tac}$ tactics. To demonstrate the extensible nature of $\mathcal{R}_{tac}$, we built a bottom-up setoid rewriter capable of inter-operating with $\mathcal{R}_{tac}$ tactics while retaining the ability to use custom data types and a global view of computation for efficiency.

The combination of MirrorCore and $\mathcal{R}_{tac}$ opens the doors to larger formalisms within dependent type theories that need to construct proof objects. $\mathcal{R}_{tac}$ allows us to combine individual reflective tactics into higher-level automation which allows us to further amortize the reification overhead across more reasoning steps. We believe that fully reflective automation will enable more and larger applications of the rich program logics currently being developed.