$\mathcal{R}$tac: A Fully Reflective Tactic Language

Gregory M. Malecha
gmalecha@cs.harvard.edu
Harvard SEAS

Jesper Bengtson
jebe@itu.dk
IT University Denmark

Computational reflection is a useful technique for avoiding the overhead inherent in constructing large proof objects. However, to date it is significantly more time consuming to write reflective procedures than the equivalent tactics. Can we build a lightweight tactic language for building reflective procedures easily? To this end, we present $\mathcal{R}$tac, a lightweight, work-in-progress, tactic language built on top of MirrorCore, a parametric framework for writing reflective decision procedures in Coq.

1 MirrorCore

MirrorCore provides a syntactic, simply-typed lambda calculus which serves as a language to embed the "interesting" symbols that our reflective procedures aim to reason about. In Coq, the syntactic representation is the following:

$$\begin{array}{rccl} \text{(types)} & \tau & ::= & \tau_1 \to \tau_2 | ... \\ \text{(expressions)} & e & ::= & e_1 e_2 | \lambda \tau . e \| x | \lceil b \rceil | ?n \end{array}$$

In the definition we use $\lceil b \rceil$ to inject our special symbols into the language and $?n$ to represent unification variables. Both of which are important building blocks as shown in MirrorShard [4].

The generality of the open definitions allows MirrorCore to support meta-level polymorphic and dependent types by including recursive and parametrized constructors in the type algebra. For example, we can reason about polymorphic lists by including a constructor tyList : $\tau \to \tau$ and we can support bit-vectors of constants sizes by including a constructor tyBv : $\mathbb{N} \to \tau$.

To make reflective procedures more generally applicable, we implement and verify them with respect to extensional characterizations of the type algebra. For example, if we require a representation of Props, we parameterize them over a Typ0 record that contains data relevant to the representation:

```
Class Typ0 : Type :=
{ typ0 : typ
; typ0_cast : typD typ0 = F
; typ0_match : ∀ (T : Type → Type) t,
  T F → T (typD t) → T (typD t) }.
```

Along with appropriate reasoning principles.

The approach allows us a considerable amount of flexibility when reusing existing automation in new domains. For example, an entailment checker for arbitrary separation logics can be applied both to a logic specialized to Imp and a logic specialized to Java as implemented in the Charge! framework [1].

2 $\mathcal{R}$tac

To ease the development of reflective procedures we are experimenting building a small tactic language modeled loosely on $\mathcal{L}$tac. To see an example, consider the following simple goal.

$$\forall x, \forall y, x \to y \to x \land y$$

The current version of $\mathcal{R}$tac can solve this goal with the following tactic.

```
Def tac : rtac :=
   REPEAT INTRO ; APPLY and_intro ; ASSUMPTION.
Thm tac_sound : rtac_sound tac.
```

While a simple proof to construct, proving the tactics sound in a compositional way has been difficult.

To explain more requires a little bit more detail related to the representation and denotation of goals and tactics. Tactics are functions from a context (representing the "above the line" facts), a substitution (representing the instantiation of unification variables in the current context), and a MirrorCore expression (representing the goal) to a result consisting of either failure or a new substitution and a new goal which implies the initial goal under the *new* substitution.

$$\begin{array}{rccl} & rtac & \equiv & \mathsf{goal} \to \mathsf{option\ goal} \\ \text{(Goals)} & g & ::= & \forall \tau . g | \exists \overline{\tau = e?} . g | e \to g \\ & & & | g_1 \land g_2 | e | \top \\ \text{(Contexts)} & C & ::= & \bullet | C \exists \overline{\tau_i} | C \forall \tau | C \, e \to \end{array}$$

Ignoring the complexity of binding variables, goals denote to propositions and contexts (along with their substitutions) denote to functions from propositions to propositions. The reason that the denotation of a context requires a substitution is that the denotation will add equations from the substitution next to existentially quantified variables. For example, the denotation of universal and existential quantifiers in the context are respectively:

$$\begin{array}{lcl} [\![C\forall\tau, S]\!]P & \equiv & [\![C]\!](\forall x : [\![\tau]\!].P..) \\ [\![C\exists\overline{\tau_i}, S]\!]P & \equiv & [\![C]\!](\exists\overline{x_i : \tau_i}.\overline{x_i = [\![S[i]]\!]..} \wedge P..) \end{array}$$

While the above definitions are the desired ones from a global perspective, they are not strong enough to prove theorems about applying tactics under binders to side-by-side goals. The intuition for this is best seen when thinking about the proof being constructed in the soundness theorem. When a tactic succeeds in solving a goal, it is guaranteeing a proof for *any* values of the environments that are consistent with the substitution. To express this, we convert the existential quantifiers into universal quantifiers and move the implication into the context.

$$[\![C, S']\!]^p(P' \to P)$$

This formulation almost gets us through the entire proof. All that is left is expressing the fact that the context can only grow monotonically by adding new constraints, old constraints can not be removed or contradicted. At first glace, this seems to be provable by expressing the following fact:

$$\forall P, [\![C, S]\!]^p P \to [\![C, S']\!]^p P$$

but this is too weak to allow popping out from under a binder. To see the problem, consider the following inconsistent context: $\forall x : \mathbb{N}, \forall y : \emptyset$. With this environment, Coq permits two ways to prove the above theorem

$$\forall P, (\forall x : \mathbb{N}, \forall y : \emptyset, P) \to (\forall x : \mathbb{N}, \forall y : \emptyset, P)$$

The obvious (and good) proof is the identity function. The "bad" one matches on the `x :` $\emptyset$. We think of this one as bad because it uses "deeper" information to prove a "shallower" property; i.e. it uses $\emptyset$ to avoid constructing a function to thread `x` through the proof.

This type of problem is usually solved by syntactic methods, which essentially require constraints to match up point-wise, but the semantic formulation of MirrorCore seems to make these overly restrictive. More semantic methods rely on parametricity effectively requiring the above proof to hold for any types (as long as they are replaced consistently). These approaches are similar in spirit to PHOAS [2] and logical relations [3] and likely related, in some way, to extensional type theories since one can view Coq as the meta-logic for MirrorCore.

Our current solution side-steps the problem by requiring a transport proof at every new quantifier. Since the proofs are not manifest during execution, this should not affect performance of running tactics.

3 Future Work

While MirrorCore is mostly stable at this point, $\mathcal{R}$tac is still in a state of flux. The proofs have been proven more subtle than initially anticipated but the definitions described above are promising and the proofs are very near completion.

Besides proofs, performance has been a difficult thing to control. `vm_compute` is essential for making things fast, but it still seems like low-level implementation details can drastically affect performance, especially spurious matches. It is possible that manual optimization via cps-conversion would be helpful, but it further complicates proofs. I am optimistic that Denes' work on refinement, proofs, and `native_compute` could further assist in achieving performance while controlling complexity.

`native_compute` however may be too general purpose. In computational reflection, there are often two properties that make evaluation simple. First, terms are closed. Only the denotation function which is not given to the reflective procedure mentions quantified terms. And second, the majority of the code is known statically and could potentially be assembled and optimized ahead-of-time for maximum performance. Combining this optimized binary with a font-end that could read the raw kernel representation and translate it into values could be a way to achieve ACL2-like automation at native performance while sacrificing relatively little in terms of soundness, especially in the light of current developments on verified Coq compilers.

References

[1] J. Bengtson, J.B. Jensen, and L. Birkedal. Charge!: a framework for higher-order separation logic in Coq. In *Proc. ITP*, 2012.

[2] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proc. ICFP*, 2008.

[3] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In *Proc. POPL*, 2012.

[4] Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In *Interactive Theorem Proving*, 2014.