Reification by Parametricity

Fast Setup for Proof by Reflection, in Two Lines of Ltac

Jason Gross, Andres Erbsen, and Adam Chlipala

MIT CSAIL, Cambridge, MA, USA
jgross@mit.edu, andreser@mit.edu, adamc@csail.mit.edu

Abstract. We present a new strategy for performing reification in Coq. That is, we show how to generate first-class abstract syntax trees from "native" terms of Coq's logic, suitable as inputs to verified compilers or procedures in the *proof-by-reflection* style. Our new strategy, based on simple generalization of subterms as variables, is straightforward, short, and fast. In its pure form, it is only complete for constants and function applications, but "let" binders, eliminators, lambdas, and quantifiers can be accommodated through lightweight coding conventions or preprocessing.
We survey the existing methods of reification across multiple Coq metaprogramming facilities, describing various design choices and tricks that can be used to speed them up, as well as various limitations. We report benchmarking results for 18 variants, in addition to our own, finding that our own reification outperforms 16 of these methods in all cases, and one additional method in some cases; writing an OCaml plugin is the only method tested to be faster. Our method is the most concise of the strategies we considered, reifying terms using only two to four lines of Ltac—beyond lists of the identifiers to reify and their reified variants. Additionally, our strategy automatically provides error messages that are no less helpful than Coq's own error messages.

1 Introduction

Proof by reflection [2] is an established method for employing verified proof procedures, within larger proofs. There are a number of benefits to using verified functional programs written in the proof assistant's logic, instead of tactic scripts. We can often prove that procedures always terminate without attempting fallacious proof steps, and perhaps we can even prove that a procedure gives logically complete answers, for instance telling us definitively whether a proposition is true or false. In contrast, tactic-based procedures may encounter runtime errors or loop forever. As a consequence, those procedures must output proof terms, justifying their decisions, and these terms can grow large, making for slower proving and requiring transmission of large proof terms to be checked slowly by others. A verified procedure need not generate a certificate for each invocation.

The starting point for proof by reflection is *reification*: translating a "native" term of the logic into an explicit abstract syntax tree. We may then feed that tree

to verified procedures or any other functional programs in the logic. The benefits listed above are particularly appealing in domains where goals are very large. For instance, consider verification of large software systems, where we might want to reify thousands of lines of source code. Popular methods turn out to be surprisingly slow, often to the point where, counter-intuitively, the majority of proof-execution time is spent in reification – unless the proof engineer invests in writing a plugin directly in the proof assistant's metalanguage (e.g., OCaml for Coq).

In this paper, we show that reification can be both simpler and faster than with standard methods. Perhaps surprisingly, we demonstrate how to reify terms almost entirely through reduction in the logic, with a small amount of tactic code for setup and no ML programming. Though our techniques should be broadly applicable, especially in proof assistants based on type theory, our experience is with Coq, and we review the requisite background in the remainder of this introduction. In section 2, we summarize our survey into prior approaches to reification and provide high-quality implementations and documentation for them, serving a tutorial function independent of our new contributions. Experts on the subject might want to skip directly to section 3, which explains our alternative technique. We benchmark our approach against 18 competitors in section 4.

1.1 Proof-Script Primer

Basic Coq proofs are often written as lists of steps such as `induction` on some structure, `rewrite` using a known equivalence, or `unfold` of a definition. Very quickly, proofs can become long and tedious, both to write and to read, and hence Coq provides LTAC, a scripting language for proofs. As theorems and proofs grow in complexity, users frequently run into performance and maintainability issues with LTAC. Consider the case where we want to prove that a large algebraic expression, involving many `let ... in ...` expressions, is even:

```
Inductive is_even : nat -> Prop :=
| even_O : is_even O
| even_SS : forall x, is_even x -> is_even (S (S x)).
Goal is_even (let x := 100 * 100 * 100 * 100 in
              let y := x * x * x * x in
              y * y * y * y).
```

Coq stack-overflows if we try to reduce this goal. As a workaround, we might write a lemma that talks about evenness of `let ... in ...`, plus one about evenness of multiplication, and we might then write a tactic that composes such lemmas.

Even on smaller terms, though, proof size can quickly become an issue. If we give a naive proof that 7000 is even, the proof term will contain all of the even numbers between 0 and 7000, giving a proof-term-size blow-up at least quadratic in size (recalling that natural numbers are represented in unary; the challenges remain for more efficient base encodings). Clever readers will notice that Coq

could share subterms in the proof tree, recovering a term that is linear in the size of the goal. However, such sharing would have to be preserved very carefully, to prevent size blow-up from unexpected loss of sharing, and today's Coq version does not do that sharing. Even if it did, tactics that rely on assumptions about Coq's sharing strategy become harder to debug, rather than easier.

1.2 Reflective-Automation Primer

Enter reflective automation, which simultaneously solves both the problem of performance and the problem of debuggability. Proof terms, in a sense, are traces of a proof script. They provide Coq's kernel with a term that it can check to verify that no illegal steps were taken. Listing every step results in large traces.

The idea of reflective automation is that, if we can get a formal encoding of our goal, plus an algorithm to *check* the property we care about, then we can do much better than storing the entire trace of the program. We can prove that our checker is correct once and for all, removing the need to trace its steps.

A simple evenness checker can just operate on the unary encoding of natural numbers (Figure 1). We can use its correctness theorem to prove goals much more quickly:

```
Fixpoint check_is_even
  (n : nat) : bool
  := match n with
     | 0 => true
     | 1 => false
     | S (S n)
       => check_is_even n
     end.
```

Fig. 1. Evenness Checking

```
Theorem soundness : forall n, check_is_even n = true -> is_even n.
Goal is_even 2000.
  Time repeat (apply even_SS || apply even_O). (* 1.8 s *)
  Undo.
  Time apply soundness; vm_compute; reflexivity. (* 0.004 s *)
```

The tactic `vm_compute` tells Coq to use its virtual machine for reduction, to compute the value of `check_is_even 2000`, after which `reflexivity` proves that `true = true`. Note how much faster this method is. In fact, even the asymptotic complexity is better; this new algorithm is linear rather than quadratic in `n`.

However, even this procedure takes a bit over three minutes to prove `is_even (10 * 10 * 10 * 10 * 10 * 10 * 10 * 10 * 10)`. To do better, we need a formal representation of terms or expressions.

1.3 Reflective-Syntax Primer

Sometimes, to achieve faster proofs, we must be able to tell, for example, whether we got a term by multiplication or by addition, and not merely whether its normal form is 0 or a successor.

A reflective automation procedure generally has two steps. The first step is to *reify* the goal into some abstract syntactic representation, which we call the *term language* or

```
Inductive expr :=
| NatO : expr
| NatS (x : expr) : expr
| NatMul (x y : expr) : expr.
```

Fig. 2. Simple Expressions

an *expression language*. The second step is to run the algorithm on the reified syntax.

What should our expression language include? At a bare minimum, we must have multiplication nodes, and we must have `nat` literals. If we encode `S` and `O` separately, a decision that will become important later in section 3, we get the inductive type of Figure 2.

Before diving into methods of reification, let us write the evenness checker.

```
Fixpoint check_is_even_expr (t : expr) : bool
  := match t with
     | NatO => true
     | NatS x => negb (check_is_even_expr x)
     | NatMul x y => orb (check_is_even_expr x) (check_is_even_expr y)
     end.
```

Before we can state the soundness theorem (whenever this checker returns `true`, the represented number is even), we must write the function that tells us what number our expression represents, called *denotation* or *interpretation*:

```
Fixpoint denote (t : expr) : nat
  := match t with
     | NatO => O
     | NatS x => S (denote x)
     | NatMul x y => denote x * denote y
     end.

Theorem check_is_even_expr_sound (e : expr)
  : check_is_even_expr e = true -> is_even (denote e).
```

Given a tactic `Reify` to produce a reified term from a `nat`, we can time `check_is_even_expr`. It is instant on the last example.

Before we proceed to reification, we will introduce one more complexity. If we want to support our initial example with `let ... in ...` efficiently, we must also have `let`-expressions. Our current procedure that inlines `let`-expressions takes 19 seconds, for example, on `let x0 := 10 * 10 in let x1 := x0 * x0 in ... let x24 := x23 * x23 in x24`. The choices of representation include higher-order abstract syntax (HOAS) [11], parametric higher-order abstract syntax (PHOAS) [4], and de Bruijn indices [3]. The PHOAS representation is particularly convenient. In PHOAS, expression binders are represented by binders in Gallina, the functional language of Coq, and the expression language is parameterized over the type of the binder. Let us define a constant and notation for `let` expressions as definitions (a common choice in real Coq developments, to block Coq's default behavior of inlining `let` binders silently; the same choice will also turn out to be useful for reification later). We thus have:

```
Inductive expr {var : Type} :=
| NatO : expr
| NatS : expr -> expr
```

```
| NatMul : expr -> expr -> expr
| Var : var -> expr
| LetIn : expr -> (var -> expr) -> expr.
Definition Let_In {A B} (v : A) (f : A -> B) := let x := v in f x.
Notation "'dlet' x := v 'in' f" := (Let_In v (fun x => f)).
Notation "'elet' x := v 'in' f" := (LetIn v (fun x => f)).
Fixpoint denote (t : @expr nat) : nat
  := match t with
     | NatO => O
     | NatS x => S (denote x)
     | NatMul x y => denote x * denote y
     | Var v => v
     | LetIn v f => dlet x := denote v in denote (f x)
     end.
```

A full treatment of evenness checking for PHOAS would require proving well-formedness of syntactic expressions; for a more complete discussion of PHOAS, we refer the reader elsewhere [4]. Using `Wf` to denote the well-formedness predicate, we could prove a theorem

```
Theorem check_is_even_expr_sound (e : ∀ var, @expr var) (H : Wf e)
: check_is_even_expr (e bool) = true -> is_even (denote (e nat)).
```

To complete the picture, we would need a tactic `Reify` which took in a term of type `nat` and gave back a term of type `forall var, @expr var`, plus a tactic `prove_wf` which solved a goal of the form `Wf e` by repeated application of constructors. Given these, we could solve an evenness goal by writing[1]

```
match goal with
| [ |- is_even ?v ]
  => let e := Reify v in
     refine (check_is_even_expr_sound e _ _);
     [ prove_wf | vm_compute; reflexivity ]
end.
```

2 Methods of Reification

We implemented reification in 18 different ways, using 6 different metaprogramming facilities in the Coq ecosystem: Ltac, Ltac2, Mtac [8], type classes [12], canonical structures [7], and reification-specific OCaml plugins (quote [5], template-coq [1], ours). Figure 3 displays the simplest case: an Ltac script to reify a tree of function applications and constants. Unfortunately, all methods we surveyed become drastically more complicated or slower (and usually both) when adapted to reify terms with variable bindings such as `let-in` or λ nodes.

[1] Note that for the `refine` to be fast, we must issue something like `Strategy -10 [denote]` to tell Coq to unfold `denote` before `Let_In`.

We have made detailed walkthroughs and source code of these implementations available[2] in hope that they will be useful for others considering implementing reification using one of these metaprogramming mechanisms, instructive as nontrivial examples of multiple metaprogramming facilities, or helpful as a case study in Coq performance engineering. However, we do *not* recommend reading these out of general interest: most of the complexity in the described implementations strikes us as needless, with significant aspects of the design being driven by surprising behaviors, misfeatures, bugs, and performance bottlenecks of the underlying machinery as opposed to the task of reification.

```
Ltac f v x := (* reify var term *)
  lazymatch x with
  | O => constr:(@NatO v)
  | S ?x => let X := f v x in
            constr:(@NatS v X)
  | ?x*?y => let X := f v x in
             let Y := f v y in
             constr:(@NatMul v X Y)
  end.
```

Fig. 3. Reification Without Binders in Ltac

3 Reification by Parametricity

We propose factoring reification into two passes, both of which essentially have robust, built-in implementations in Coq: *abstraction* or *generalization*, and *substitution* or *specialization*.

The key insight to this factoring is that the shape of a reified term is essentially the same as the shape of the term that we start with. We can make precise the way these shapes are the same by abstracting over the parts that are different, obtaining a function that can be specialized to give either the original term or the reified term.

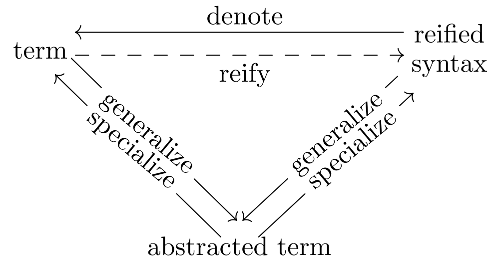


Fig. 4. Abstraction and Reification

That is, we have the commutative triangle in Figure 4.

3.1 Case-By-Case Walkthrough

Function Applications And Constants. Consider the example of reifying 2×2. In this case, the *term* is 2×2 or (mul (S (S O)) (S (S O))).

To reify, we first *generalize* or *abstract* the term 2×2 over the successor function S, the zero constructor O, the multiplication function mul, and the type $\mathbb{N}$ of natural numbers. We get a function taking one type argument and three value arguments:

$$\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N)\ (\mathrm{O} : N)\ (\mathrm{S} : N \to N).\ \textsc{Mul}\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{O}))\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{O}))$$

[2] https://github.com/mit-plv/reification-by-parametricity

We can now specialize this term in one of two ways: we may substitute $\mathbb{N}$, mul, O, and S, to get back the term we started with; or we may substitute `expr`, `NatMul`, `NatO`, and `NatS` to get the reified syntax tree

```
NatMul (NatS (NatS NatO)) (NatS (NatS NatO))
```

This simple two-step process is the core of our algorithm for reification: abstract over all identifiers (and key parts of their types) and specialize to syntax-tree constructors for these identifiers.

Wrapped Primitives: "Let" Binders, Eliminators, Quantifiers. The above procedure can be applied to a term that contains "let" binders to get a PHOAS syntax tree that represents the original term, but doing so would not capture sharing. The result would contain native "let" bindings of subexpressions, not PHOAS let expressions. Call-by-value evaluation of any procedure applied to the reification result would first substitute the let-bound subexpressions – leading to potentially exponential blowup and, in practice, memory exhaustion.

The abstraction mechanisms in all proof assistants (that we know about) only allow abstracting over terms, not language primitives. However, primitives can often be wrapped in explicit definitions, which we *can* abstract over. For example, we already used a wrapper for "let" binders, and terms that use it can be reified by abstracting over that definition. If we start with the expression

$$\texttt{dlet}\ a := 1\ \texttt{in}\ a \times a$$

and abstract over (`@Let_In` $\mathbb{N}$ $\mathbb{N}$), S, O, mul, and $\mathbb{N}$, we get a function of one type argument and four value arguments:

$$\Lambda N.\, \lambda\, (\textsc{Mul} : N \to N \to N).\ \lambda(\mathrm{O} : N).\ \lambda(\mathrm{S} : N \to N).$$
$$\lambda(\textsc{LetIn} : N \to (N \to N) \to N).\ \textsc{LetIn}\ (\mathrm{S}\ \mathrm{O})\ (\lambda a.\ \textsc{Mul}\ a\ a)$$

We may once again specialize this term to obtain either our original term or the reified syntax. Note that to obtain reified PHOAS syntax, we must include a `Var` node in the `LetIn` expression; we substitute $(\lambda x\ f.\ \texttt{LetIn}\ x\ (\lambda v.\ f\ (\texttt{Var}\ v)))$ for LetIn to obtain the PHOAS syntax tree

$$\texttt{LetIn (NatS NatO)}\ (\lambda v.\ \texttt{NatMul (Var}\ v\texttt{) (Var}\ v\texttt{))}$$

Wrapping a metalanguage primitive in a definition in the code to be reified is in general sufficient for reification by parametricity. Pattern matching and recursion cannot be abstracted over directly, but if the same code is expressed using eliminators, these can be handled like other functions. Similarly, even though $\forall/\Pi$ cannot be abstracted over, proof automation that itself introduces universal quantifiers before reification can easily wrap them in a marker definition (`_forall T P := forall (x:T), P x`) that can be. Existential quantifiers are not primitive in Coq and can be reified directly.

Lambdas. While it would be sufficient to require that, in code to be reified, we write all lambdas with a named wrapper function, that would significantly

clutter the code. We can do better by making use of the fact that a PHOAS object-language lambda (`Abs` node) consists of a metalanguage lambda that binds a value of type `var`, which can be used in expressions through constructor `Var` : `var` $\rightarrow$ `expr`. Naive reification by parametricity would turn a lambda of type $N \rightarrow N$ into a lambda of type `expr` $\rightarrow$ `expr`. A reification procedure that explicitly recurses over the metalanguage syntax could just precompose this recursive-call result with `Var` to get the desired object-language encoding of the lambda, but handling lambdas specially does not fit in the framework of abstraction and specialization.

First, let us handle the common case of lambdas that appear as arguments to higher-order functions. One easy approach: while the parametricity-based framework does not allow for special-casing lambdas, it is up to us to choose how to handle functions that we expect will take lambdas as arguments. We may replace each higher-order function with a metalanguage lambda that wraps the higher-order arguments in object-language lambdas, inserting `Var` nodes as appropriate. Code calling the function `sum_upto` $n\ f := f(0) + f(1) + \cdots + f(n)$ can be reified by abstracting over relevant definitions and substituting $(\lambda n\ f.\ \texttt{SumUpTo}\ n\ (\texttt{Abs}\ (\lambda v.\ f\ (\texttt{Var}\ v))))$ for `sum_upto`. Note that the expression plugged in for `sum_upto` differs from the one plugged in for `Let_In` only in the use of a deeply embedded abstraction node. If we wanted to reify `LetIn` as just another higher-order function (as opposed to a distinguished wrapper for a primitive), the code would look identical to that for `sum_upto`.

It would be convenient if abstracting and substituting for functions that take higher-order arguments were enough to reify lambdas, but here is a counterexample.

$$\lambda\ x\ y.\ x \times ((\lambda\ z.\ z \times z)\ y)$$

$$\Lambda N.\ \lambda(\textsc{Mul} : N \rightarrow N \rightarrow N).\ \lambda\ (x\ y : N).\ \texttt{Mul}\ x\ ((\lambda\ (z : N).\ \texttt{Mul}\ z\ z)\ y)$$

$$\lambda\ (x\ y : \texttt{expr}).\ \texttt{NatMul}\ x\ (\texttt{NatMul}\ y\ y)$$

The result is not even a PHOAS expression. We claim a desirable reified form is

$$\texttt{Abs}(\lambda\ x.\ \texttt{Abs}(\lambda\ y.\ \texttt{NatMul}\ (\texttt{Var}\ x)\ (\texttt{NatMul}\ (\texttt{Var}\ y)\ (\texttt{Var}\ y))))$$

Admittedly, even our improved form is not quite precise: $\lambda\ z.\ z \times z$ has been lost. However, as almost all standard Coq tactics silently reduce applications of lambdas, working under the assumption that functions not wrapped in definitions will be arbitrarily evaluated during scripting is already the norm. Accepting that limitation, it remains to consider possible occurrences of metalanguage lambdas in normal forms of outputs of reification as described so far. As lambdas in `expr` nodes that take metalanguage functions as arguments (`LetIn`, `Abs`) are handled by the rules for these nodes, the remaining lambdas must be exactly at the head of the expression. Manipulating these is outside of the power of abstraction and specialization; we recommend postprocessing using a simple recursive tactic script.

3.2 Commuting Abstraction and Reduction

Sometimes, the term we want to reify is the result of reducing another term. For example, we might have a function that reduces to a term with a variable number of `let` binders.[3] We might have an inductive type that counts the number of `let ... in ...` nodes we want in our output.

```
Inductive count := none | one_more (how_many : count).
```

It is important that this type be syntactically distinct from $\mathbb{N}$ for reasons we will see shortly.

We can then define a recursive function that constructs some number of nested `let` binders:

```
Fixpoint big (x:nat) (n:count)
  : nat
  := match n with
     | none => x
     | one_more n'
      => dlet x' := x * x in
         big x' n'
     end.
```

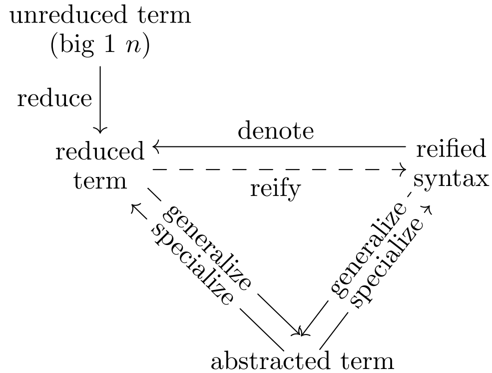
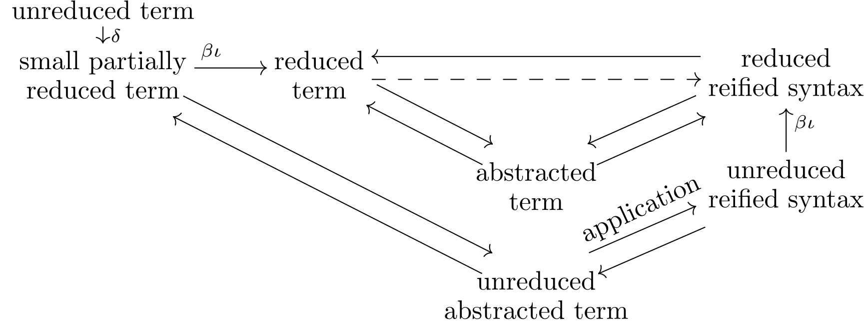


Fig. 5. Abstraction, Reification, Reduction

Our commutative diagram in Figure 4 now has an additional node, becoming Figure 5. Since generalization and specialization are proportional in speed to the size of the term begin handled, we can gain a significant performance boost by performing generalization before reduction. To explain why, we split apart the commutative diagram a bit more; in reduction, there is a δ or unfolding step, followed by a $\beta\iota$ step that reduces applications of λs and evaluates recursive calls. In specialization, there is an application step, where the λ is applied to arguments, and a β-reduction step, where the arguments are substituted. To obtain reified syntax, we may perform generalization after δ-reduction (before $\beta\iota$-reduction), and we are not required to perform the final β-reduction step of specialization to get a well-typed term. It is important that unfolding `big` results in exposing the body for generalization, which we accomplish in Coq by exposing the anonymous recursive function; in other languages, the result may be a primitive eliminator applied to the body of the fixpoint. Either way, our

[3] More realistically, we might have a function that represents big numbers using multiple words of a user-specified width. In this case, we may want to specialize the procedure to a couple of different bitwidths, then reifying the resulting partially reduced term.

commutative diagram thus becomes



Let us step through this alternative path of reduction using the example of the unreduced term `big 1 100`, where we take 100 to mean the term represented by $\underbrace{(\texttt{one_more} \cdots (\texttt{one_more}}_{100}\ \texttt{none}\ \underbrace{) \cdots)}_{100}$.

Our first step is to unfold `big`, rendered as the arrow labeled δ in the diagram. In Coq, the result is an anonymous fixpoint; here we will write it using the recursor `count_rec` of type $\forall T.\ T \to (\texttt{count} \to T \to T) \to \texttt{count} \to T$. Performing δ-reduction, that is, unfolding `big`, gives us the small partially reduced term

$$\big(\lambda(x : \mathbb{N}).\ \lambda(n : \texttt{count}).$$
$$\texttt{count_rec}\ (\mathbb{N} \to \mathbb{N})\ (\lambda x.\ x)\ (\lambda n'.\ \lambda \texttt{big}_{n'}.\ \lambda x.\ \texttt{dlet}\ x' := x{\times}x\ \texttt{ in big}_{n'}\ x')\big)\ 1\ 100$$

We call this term small, because performing $\beta\iota$ reduction gives us a much larger reduced term:

$$\texttt{dlet}\ x_1 := 1 \times 1\ \texttt{ in }\ \cdots\ \texttt{dlet}\ x_{100} := x_{99} \times x_{99}\ \texttt{ in }\ x_{100}$$

Abstracting the small partially reduced term over (`@Let_In` $\mathbb{N}$ $\mathbb{N}$), S, O, mul, and $\mathbb{N}$ gives us the abstracted unreduced term

$$\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N)(\mathrm{O} : N)(\mathrm{S} : N \to N)(\textsc{LetIn} : N \to (N \to N) \to N).$$
$$\big(\lambda(x : N).\ \lambda(n : \texttt{count}).\ \texttt{count_rec}\ (N \to N)\ (\lambda x.\ x)$$
$$(\lambda n'.\ \lambda \texttt{big}_{n'}.\ \lambda x.\ \textsc{LetIn}\ (\textsc{Mul}\ x\ x)\ (\lambda x'.\ \texttt{big}_{n'}\ x'))\big)$$
$$(\mathrm{S}\ \mathrm{O})\ \ 100$$

Note that it is essential here that `count` is not syntactically the same as $\mathbb{N}$; if they were the same, the abstraction would be ill-typed, as we have not abstracted over `count_rec`. More generally, it is essential that there is a clear separation between types that we reify and types that we do not, and we must reify *all* operations on the types that we reify.

We can now apply this term to `expr`, `NatMul`, `NatS`, `NatO`, and, finally, $(\lambda v\ f.\ \texttt{LetIn}\ v\ (\lambda x.\ f\ (\texttt{Var}\ x)))$. We get an unreduced reified syntax tree of type `expr`. If we now perform $\beta\iota$ reduction, we get our fully reduced reified term.

We take a moment to emphasize that this technique is not possible with any other method of reification. We could just as well have not specialized the function to the `count` of `100`, yielding a function of type `count` $\rightarrow$ `expr`, despite the fact that our reflective language knows nothing about `count`!

This technique is especially useful for terms that will not reduce without concrete parameters, but which should be reified for many different parameters. Running reduction once is slightly faster than running OCaml reification once, and it is more than twice as fast as running reduction followed by OCaml reification. For sufficiently large terms and sufficiently many parameter values, this performance beats even OCaml reification.[4]

3.3 Implementation in Ltac

`ExampleMoreParametricity.v` in the code supplement mirrors the development of reification by parametricity in subsection 3.1.

Unfortunately, Coq does not have a tactic that performs abstraction.[5] However, the `pattern` tactic suffices; it performs abstraction followed by application, making it a sort of one-sided inverse to β-reduction. By chaining `pattern` with an Ltac-`match` statement to peel off the application, we can get the abstracted function.

```
Ltac Reify x :=
match(eval pattern nat, Nat.mul, S, O, (@Let_In nat nat) in x)with
| ?rx _ _ _ _ _ =>
 constr:( fun var => rx (@expr var) NatMul NatS NatO
                        (fun v f => LetIn v (fun x => f (Var x))) )
end.
```

Note that if `@expr var` lives in `Type` rather than `Set`, an additional step involving retyping the term is needed; we refer the reader to `Parametricity.v` in the code supplement.

The error messages returned by the `pattern` tactic can be rather opaque at times; in `ExampleParametricityErrorMessages.v`, we provide a procedure for decoding the error messages.

Open Terms. At some level it is natural to ask about generalizing our method to reify open terms (i.e., with free variables), but we think such phrasing is a red herring. Any lemma statement about a procedure that acts on a representation of open terms would need to talk about how these terms would be closed. For example, solvers for algebraic goals without quantifiers treat free variables as implicitly universally quantified. The encodings are invariably ad-hoc: the free

[4] We discovered this method in the process of needing to reify implementations of cryptographic primitives [6] for a couple hundred different choices of numeric parameters (e.g., prime modulus of arithmetic). A couple hundred is enough to beat the overhead.

[5] The `generalize` tactic returns $\forall$ rather than λ, and it only works on types.

variables might be assigned unique numbers during reification, and the lemma statement would be quantified over a sufficiently long list that these numbers will be used to index into. Instead, we recommend directly reifying the natural encoding of the goal as interpreted by the solver, e.g. by adding new explicit quantifiers. Here is a hypothetical goal and a tactic script for this strategy:

```
(a b : nat) (H : 0 < b) |- ∃ q r, a = q × b + r ∧ r < b
```

```
repeat match goal with
       | n : nat |- ?P =>
         match eval pattern n in P with
         | ?P' _ => revert n; change (_forall nat P')
         end
       | H : ?A  |- ?B => revert H; change (impl A B)
       | |- ?G => (* ∀ a b, 0 < b -> ∃ q r, a = q × b + r ∧ r < b *)
         let rG := Reify G in
         refine (nonlinear_integer_solver_sound rG _ _);
         [ prove_wf | vm_compute; reflexivity ]
       end.
```

Briefly, this script replaced the context variables `a` and `b` with universal quantifiers in the conclusion, and it replaced the premise `H` with an implication in the conclusion. The syntax-tree datatype used in this example can be found in `ExampleMoreParametricity.v`.

3.4 Advantages and Disadvantages

This method is faster than all but Ltac2 and OCaml reification, and commuting reduction and abstraction makes this method faster even than the low-level Ltac2 reification in many cases. Additionally, this method is much more concise than nearly every other method we have examined, and it is very simple to implement.

We will emphasize here that this strategy shines when the initial term is small, the partially computed terms are big (and there are many of them), and the operations to evaluate are mostly well-separated by types (e.g., evaluate all of the `count` operations and none of the `nat` ones).

This strategy is not directly applicable for reification of `match` (rather than eliminators) or `let ... in ...` (rather than a definition that unfolds to `let ... in ...`), `forall` (rather than a definition that unfolds to `forall`), or when reification should not be modulo $\beta\iota\zeta$-reduction.

4 Performance Comparison

We have done a performance comparison of the various methods of reification to the PHOAS language `@expr var` from Figure 1.3 in Coq 8.7.1. A typical reification routine will obtain the term to be reified from the goal, reify it,

run `transitivity (denote reified_term)` (possibly after normalizing the reified term), and solve the side condition with something like `lazy [denote]; reflexivity`. Our testing on a few samples indicated that using `change` rather than `transitivity; lazy [denote]; reflexivity` can be around 3X slower; note that we do not test the time of `Defined`.

There are two interesting metrics to consider: (1) how long does it take to reify the term? and (2) how long does it take to get a normalized reified term, i.e., how long does it take both to reify the term and normalize the reified term? We have chosen to consider (1), because it provides the most fine-grained analysis of the actual reification method.

4.1 Without Binders

We look at terms of the form `1 * 1 * 1 * ...` where multiplication is associated to create a balanced binary tree. We say that the *size of the term* is the number of 1s. We refer the reader to the attached code for the exact test cases and the code of each reification method being tested.

We found that the performance of all methods is linear in term size.

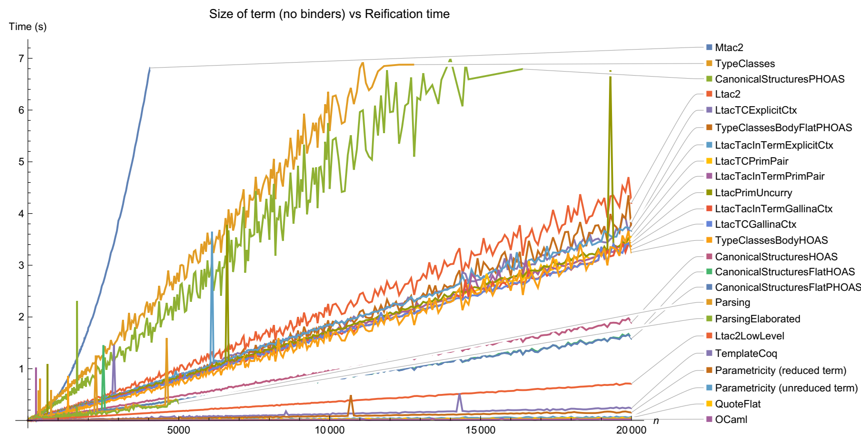


Fig. 6. Performance of Reification without Binders

Sorted from slowest to fastest, most of the labels in Figure 6 should be self-explanatory and are found in similarly named `.v` files in the associated code; we call out a few potentially confusing ones:

– The "Parsing" benchmark is "reification by copy-paste": a script generates a `.v` file with notation for an already-reified term; we benchmark the amount of time it takes to parse and typecheck that term. The "ParsingElaborated" benchmark is similar, but instead of giving notation for an already-reified term, we give the complete syntax tree, including arguments normally left

implicit. Note that these benchmarks cut off at around 5000 rather than at around 20 000, because on large terms, Coq crashes with a stack overflow in parsing.
– We have four variants starting with "CanonicalStructures" here. The Flat variants reify to `@expr nat` rather than to `forall var, @expr var` and benefit from fewer function binders and application nodes. The HOAS variants do not include a case for `let ... in ...` nodes, while the PHOAS variants do. Unlike most other reification methods, there is a significant cost associated with handling more sorts of identifiers in canonical structures.

We note that on this benchmark our method is slightly faster than template-coq, which reifies to de Bruijn indices, and slightly slower than the quote plugin in the standard library and the OCaml plugin we wrote by hand.

4.2 With Binders

We look at terms of the form `dlet` $\texttt{a}_1$ `:= 1 * 1 in dlet` $\texttt{a}_2$ `:=` $\texttt{a}_1$ `*` $\texttt{a}_1$ `in ... dlet` $\texttt{a}_n$ `:=` $\texttt{a}_{n-1}$ `*` $\texttt{a}_{n-1}$ `in` $\texttt{a}_n$, where n is the size of the term. The first graph shown here includes all of the reification variants at linear scale, while the next step zooms in on the highest-performance variants at log-log scale.

In addition to reification benchmarks, the graph in Figure 7 includes as a reference (1) the time it takes to run `lazy` reduction on a reified term already in normal form ("identity lazy") and (2) the time it takes to check that the reified term matches the original native term ("lazy Denote"). The former is just barely faster than OCaml reification; the latter often takes longer than reification itself. The line for the template-coq plugin cuts off at around 10 000 rather than around 20 000 because at that point template-coq starts crashing with stack overflows.

A nontrivial portion of the cost of "Parametricity (reduced term)" seems to be due to the fact that looking up the type of a binder is linear in the number of binders in the context, thus resulting in quadratic behavior of the retyping step that comes after abstraction in the `pattern` tactic. In Coq 8.8, this lookup will be $\log n$, and so reification will become even faster [10].

5 Future Work, Concluding Remarks

We identify one remaining open question with this method that has the potential of removing the next largest bottleneck in reification: using reduction to show that the reified term is correct.

Recall our reification procedure and the associated diagram, from Figure 3.2. We perform δ on an unreduced term to obtain a small, partially reduced term; we then perform abstraction to get

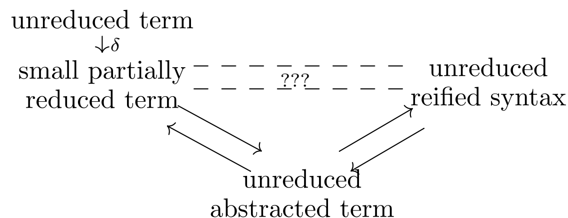


Fig. 8. Completing the commutative triangle

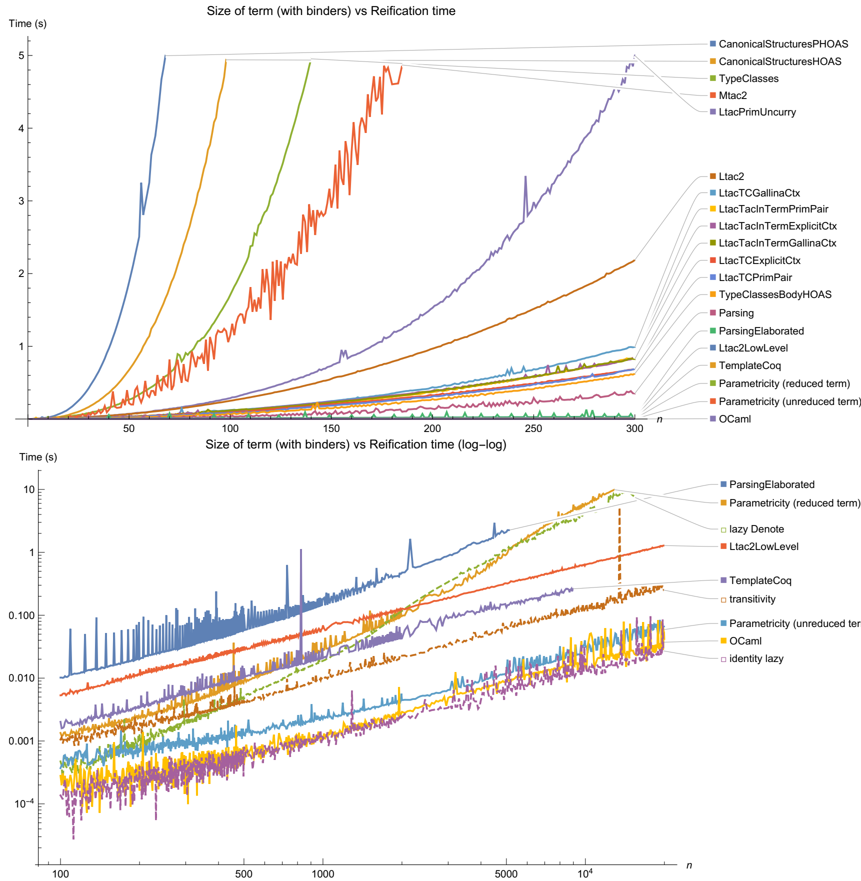


Fig. 7. Performance of Reification with Binders

an abstracted, unreduced term, followed by application to get unreduced reified syntax. These steps are all fast. Finally, we perform $\beta\iota$-reduction to get reduced, reified syntax and perform $\beta\iota\delta$ reduction to get back a reduced form of our original term. These steps are slow, but we must do them if we are to have verified reflective automation.

It would be nice if we could prove this equality without ever reducing our term. That is, it would be nice if we could have the diagram in Figure 8.

The question, then, is how to connect the small partially reduced term with `denote` applied to the unreduced reified syntax. That is, letting F denote the unreduced abstracted term, how can we prove, without reducing F, that

$$F\ \mathbb{N}\ \text{Mul O S}\ (@\text{Let_In}\ \mathbb{N}\ \mathbb{N}) = \texttt{denote}\ \ (F\ \texttt{expr NatMul NatO NatS LetIn})$$

We hypothesize that a form of internalized parametricity would suffice for proving this lemma. In particular, we could specialize F's type argument with $\mathbb{N} \times$ `expr`. Then we would need a proof that for any function F of type

$$\forall (T : \texttt{Type}), (T \to T \to T) \to T \to (T \to T) \to (T \to (T \to T) \to T) \to T$$

and any types A and B, and any terms $f_A : A \to A \to A$, $f_B : B \to B \to B$, $a : A$, $b : B$, $g_A : A \to A$, $g_B : B \to B$, $h_A : A \to (A \to A) \to A$, and $h_B : B \to (B \to B) \to B$, using $f \times g$ to denote lifting a pair of functions to a function over pairs:

$$\texttt{fst}\ (F\ (A \times B)\ (f_A \times f_B)\ (a, b)\ (g_A \times g_B)\ (h_A \times h_B)) = F\ A\ f_A\ a\ g_A\ h_A \land$$
$$\texttt{snd}\ (F\ (A \times B)\ (f_A \times f_B)\ (a, b)\ (g_A \times g_B)\ (h_A \times h_B)) = F\ B\ f_B\ b\ g_B\ h_B$$

This theorem is a sort of parametricity theorem.

Despite this remaining open question, we hope that our performance results make a strong case for our method of reification; it is fast, concise, and robust.

6 Acknowledgments and Historical Notes

We would like to thank Hugo Herbelin for sharing the trick with `type of` to propagate universe constraints[6] as well as useful conversations on Coq's bug tracker that allowed us to track down performance issues.[7] We would like to thank Pierre-Marie Pédrot for conversations on Coq's Gitter and his help in tracking down performance bottlenecks in earlier versions of our reification scripts and in Coq's tactics. We would like to thank Beta Ziliani for his help in using Mtac2, as well as his invaluable guidance in figuring out how to use canonical structures to reify to PHOAS. We also thank John Wiegley for feedback on the paper.

For those interested in history, our method of reification by parametricity was inspired by the `evm_compute` tactic [9]. We first made use of `pattern` to allow `vm_compute` to replace `cbv`-with-an-explicit-blacklist when we discovered `cbv` was too slow and the blacklist too hard to maintain. We then noticed that in the sequence of doing abstraction; `vm_compute`; application; β-reduction; reification, we could move β-reduction to the end of the sequence if we fused reification with application, and thus reification by parametricity was born.

This work was supported in part by a Google Research Award and National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584.

References

1. Anand, A., Boulier, S., Tabareau, N., Sozeau, M.: Typed Template Coq. CoqPL 2018 (Jan 2018), https://popl18.sigplan.org/event/coqpl-2018-typed-template-coq

[6] https://github.com/coq/coq/issues/5996#issuecomment-338405694
[7] https://github.com/coq/coq/issues/6252

2. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Abadi, M., Ito, T. (eds.) Theoretical Aspects of Computer Software. pp. 515–529. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
3. de Bruijn, N.G.: Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) **34(5)**, 381–392 (1972). https://doi.org/https://doi.org/10.1016/1385-7258(72)90034-0, `http://www.sciencedirect.com/science/article/pii/1385725872900340`
4. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (Sep 2008), `http://adam.chlipala.net/papers/PhoasICFP08/`
5. Coq Development Team: The Coq Proof Assistant Reference Manual, chap. 10.3 Detailed examples of tactics (quote). INRIA, 8.7.1 edn. (2017), `https://coq.inria.fr/distrib/V8.7.1/refman/tactic-examples.html#quote-examples`
6. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In: Proc. IEEE Symposium on Security & Privacy (May 2019)
7. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Tech. rep., Inria Saclay Ile de France (Nov 2016), `https://hal.inria.fr/inria-00258384/`
8. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. Journal of Functional Programming **23**(4), 357–401 (2013). https://doi.org/10.1017/S0956796813000051, `https://people.mpi-sws.org/~beta/lessadhoc/lessadhoc-extended.pdf`
9. Malecha, G., Chlipala, A., Braibant, T.: Compositional computational reflection. In: ITP'14: Proceedings of the 5th International Conference on Interactive Theorem Proving (2014), `http://adam.chlipala.net/papers/MirrorShardITP14/`
10. Pédrot, P.M.: Fast rel lookup #6506 (Dec 2017), `https://github.com/coq/coq/pull/6506`
11. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proc. PLDI. pp. 199–208 (1988), `https://www.cs.cmu.edu/~fp/papers/pldi88.pdf`
12. Sozeau, M., Oury, N.: First-class type classes. Lecture Notes in Computer Science **5170**, 278–293 (2008), `https://www.irif.fr/~sozeau/research/publications/First-Class_Type_Classes.pdf`