Reification by Parametricity

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

Jason Gross[1], Andres Erbsen[2], and Adam Chlipala[3]

[1] MIT CSAIL, Cambridge, MA, USA
jgross@mit.edu
[2] MIT CSAIL, Cambridge, MA, USA
andreser@mit.edu
[3] MIT CSAIL, Cambridge, MA, USA
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 proof procedures in the *proof by reflection* style. Our new strategy, based on the `pattern` tactic, is simple, short, and fast. We survey the existing methods of reification, describing various design choices and tricks that can be used to speed them up, as well as various limitations. Our strategy is not a good fit, for example, when a term must be reified without performing $\beta\iota\zeta$ reduction. We describe the results of benchmarking 18 variants of reification, 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; the fastest method of reification we tested is writing an OCaml plugin. Our method is the most concise of the strategies we considered, requiring only two to four lines of Ltac—beyond lists of the identifiers to reify and their reified variants—to reify a term. Additionally, our strategy automatically provides error messages which 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 then survey prior approaches to reification, 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_0 : is_even 0
| 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 ...`, and 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 very carefully preserved, to prevent unexpected 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, and 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.

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

Fig. 1. Eveness Checking

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:

```
Theorem soundness : forall n, check_is_even n = true -> is_even n.
Goal is_even 2000.
  Time repeat (apply even_SS || apply even_0). (* 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, the asymptotic complexity is better; this new algorithm is linear in `n`, rather than quadratic.

However, even this procedure takes a bit over three minutes to prove `is_even (10 * 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.

Reflective automation procedures generally have 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 which 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, that 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` which produces 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 which 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) [13], 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. Finally, because much of Coq ζ-reduces terms freely (i.e., inlines `let` binders), we define a constant and notation for `let` expressions as definitions. 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].

2 Methods of Reification

We proceed now to discussing the various ways of reifying terms. See section 4 for a performance comparison of these methods. We expect the reader to be struck by the seemingly needless complexity of some of these methods, even for the few where we have chosen to show moderate code detail here; we feel the same way! It is certainly fine to give up on following some of these details, in advance of reaching the much simpler presentation of our new method in section 3. Our supplementary code, most of which can also be found in Appendix E, contains full, commented implementations of this whole menagerie, both for educational purposes and to use in our performance experiments.

2.1 Typeclasses

Coq's *typeclasses* [14] provide a mechanism for running tactics during type inference. It turns out that typeclass-based reification is one of the simplest methods (even simpler than Ltac when binders are required). It allows more open or modular reification, but it can be harder to debug.

```
Local Generalizable Variables x y rx ry f rf.
Section with_var.
  Context {var : Type}.
  Class reify_of (term : nat) (rterm : @expr var) := {}.
  Global Instance reify_O                  : reify_of O NatO.
  Global Instance reify_S `{reify_of x rx} : reify_of (S x) (NatS rx).
  Global Instance reify_NatMul `{reify_of x rx, reify_of y ry}
                               : reify_of (x * y) (NatMul rx ry).
  Global Instance reify_LetIn `{reify_of x rx}
```

```
        `{forall y ry, reify_of y (Var ry) -> reify_of (f y) (rf ry)}
        : reify_of (dlet y := x in f y) (elet ry := rx in rf ry).
End with_var.
Ltac reify var x :=
  let c := constr:(_ : @reify_of var x _) in
  lazymatch type of c with reify_of _ ?rx => rx end.
```

Unfortunately, the size of the output of this typeclass resolution is quadratic in the size of the input, making it slower than most alternative strategies.

2.2 Ltac

Ltac reification is one of the simplest methods of reification, if binders are not needed. The idea is to recurse over the structure of the term, in Ltac. These procedures are generally easy to write and suffer mainly in performance and complexity around reifying binders. Most of the time in binder-heavy code seems to result from the overhead of switching back and forth between term checking and tactic evaluation. If we do not need binders, we can write the Ltac in Figure 3. Note that we use `lazymatch` rather than `match` to propagate error messages and prevent unwanted backtracking.

However, if we want binders, we must recurse under binders, and we have seven options. One option is to uncurry functions as we go, so that we always reify functions of exactly one binder. If we do not do that, we have to make two choices: (1) how do we recurse under binders? and (2) how do we keep track of the PHOAS `Var` node corresponding to each binder? We present what we believe is the simplest of these, and then we briefly describe the others.

```
Ltac reify var x :=
  lazymatch x with
  | O
    => constr:(@NatO var)
  | S ?x
    => let rx := reify var x in
       constr:(@NatS var rx)
  | ?x * ?y
    => let rx := reify var x in
       let ry := reify var y in
       constr:(@NatMul var rx ry)
  end.
```

Fig. 3. Ltac Reification

Tracking variables with Coq hypotheses and using typeclasses to recurse under binders (`LtacTCGallinaCtx.v`). We have already seen how to do reification with typeclasses. We can instead use typeclasses only for recursing under binders. Much like typeclass-based reification, we track which `nat` binders reify to which `var` binders by adding instances of a marker definition `var_for` in the context. We proceed in three steps: we must declare the class; we must write the code to invoke the typeclass to recurse under binders; and we must add a typeclass resolution hint to invoke our tactic to solve this class.

```
Class reify_helper_cls (var : Type) (term : nat)
  := do_reify_helper : @expr var.
Definition var_for {var : Type} (n : nat) (v : var) := False.
```

```
Ltac reify var term :=
  lazymatch goal with H:var_for term ?v |-_ => constr:(@Var var v)
  |_ => lazymatch term with
  | (dlet x := ?v in ?f) =>
    let rv := reify var v in
    let not_x := fresh in
    let rf := lazymatch constr:(_ : forall (x : nat) (not_x : var)
                (_ : @var_for var x not_x), @reify_cls var f) with
        | fun _ v' _ => @?f v' => f
        | ?f => error_cant_elim_deps f
        end in constr:(@LetIn var rv rf)
  | (* ... non-binders reified as in Ltac ... *) end end.
Global Hint Extern 0 (@reify_helper_cls _ _)
 => (intros; lazymatch goal with |- @reify_helper_cls ?var ?term
     => let res := reify var term in exact res end) : typeclass_instances.
```

The last `lazymatch` in `reify` serves two purposes: (1) it removes the `nat` and `var_for` binders, which are required for recursive reification but which must be unused in the resulting term if the reified term is to be valid; and (2) it strips off the type cast used to invoke typeclass resolution, keeping term size linear.

Other Methods For Recursing Under Binders

Uncurrying (`LtacPrimUncurry.v`). As in CPDT [5], we can uncurry functions on the fly and use `@?` patterns to look under a single binder. For speed, we avoid implicit argument inference, use primitive projections [6], and trigger local β-reduction by feeding the Coq elaborator single-case matches [10].

Recursing under binders with typeclasses (`LtacTC*.v`). As explained above. If we choose to track variables with explicit contexts, the context will be an argument of the type class.

Recursing under binders with tactics in terms (`LtacTacInTerm*.v`). Since Coq 8.5, `ltac:(...)` can be used insert tactic-generated terms directly into Gallina code, even under binders. There are a number of bugs and misfeatures to be wary of when using this method; the attached implementations were developed by working around one unexpected behavior after another.

Other Methods For Tracking Variables

Pairs (`Ltac*PrimPair.v`).
We can handle binders by pairing variables with newly introduced binders of type `var` and then reify `fst (?term, ?v)` to `@Var var v` whenever possible.

Coq hypotheses (`Ltac*GallinaCtx.v`). As in typeclass-based reification, we can introduce a definition to track which `nat` binders reify to which `var` binders, searching the context for such hypotheses.

Explicit context (`Ltac*ExplicitCtx.v`).
Finally, rather than relying on Coq's contexts, we can pass around an association list explicitly.

2.3 Canonical Structures

Automation via canonical structures was pioneered by Gonthier et. al. [9] It is fairly concise but an enormous pain to debug[4] and takes a bit of work to wrap one's head around. The basic idea is that, when Coq's unification engine encounters a unification problem of the form `projection e = term`, if `e` is a term of record type with holes, and if there is a canonical structure declared for that record with the head constant of `term` in the field `projection`, then Coq will try to solve `?e` by unifying `term` with an application of the canonical structure.

Canonical-structure reification is much faster than Ltac reification for small terms without binders. For large terms, we get bitten by the fact that canonical-structure resolution generates quadratically sized terms, much like typeclass resolution. For more detail on canonical-structure reification, see Appendix C. For full implementations, see Appendix E.8, the `CanonicalStructures*.v` files.

2.4 Mtac (`Mtac2.v`)

Created by the authors of *How to Make Ad Hoc Proof Automation Less Ad Hoc* [9] in part to deal with the fact that canonical structures are painful, Mtac [15] is a monadic tactic language whose tactics are Gallina terms. Inexperienced as we are with Mtac, we have only two suggestions for speeding up Mtac-based reification: (1) avoid unnecessary normalization; and (2) handle fresh binder names manually rather than invoking `M.fresh_binder_name`, which produces a string with length linear in the number of times it has been called so far, rather than logarithmic.

2.5 Ltac2 (`Ltac2LowLevel.v`, `Ltac2.v`)

In upcoming versions of Coq, there is a new, saner replacement for Ltac called Ltac2. Its main benefits are a less exciting execution model, more fine-grained control over manipulation of terms, and static typechecking.

A relatively straightforward transcription to Ltac2 of the reification routine that explicitly tracks variables and uses tactics-in-terms to recurse is about 2X slower than the corresponding Ltac, likely due to allocating twice as many evars. However, the real benefit of Ltac2 comes from being able to write low-level term-manipulation code without incurring the overhead normally associated with term manipulation in Ltac1.[5] One key insight here is that we do not need to track variable contexts *at all*! We can instead retype the same binders with type `var`. By writing such low-level code, we get a 50X speedup over naive Ltac2 code.

[4] Primarily due to an inability to insert print statements, plus near incomprehensibility of `Set Debug Unification`. We have heard that UniCoq [16] makes unification more debuggable, though we have not yet tried it ourselves.

[5] It is not entirely clear to us where this overhead comes from. Our hypotheses, from dialogue with Coq developers, include (re)typechecking terms, memory allocation, and *evar normalization*, a procedure in which Coq makes sure to give a consistent view of which evars have been filled and which have not.

2.6 OCaml

The upper bound on reification performance (as for so many other Coq procedures) is attained by writing an OCaml plugin. Faster even than parsing a pre-reified term, a line-by-line translation of the low-level Ltac2 reification procedure into OCaml results in another 50X speedup. Pierre-Marie Pédrot, the author of Ltac2, said that essentially all of the slowness of Ltac2 over OCaml comes from the overhead of Ltac2 being interpreted. We look forward to the day when this straightforward compilation is built into the Ltac2 plugin.

One might ask: why not write all reification in OCaml? Our answers are that:

1. Historically, Coq's OCaml API has been rather unstable between versions. The situation is improving, but we are not yet at a point where porting a plugin from one version of Coq to another is easy to do without knowledge of the arcana of Coq's internals.
2. Coq gives interactive, line-by-line feedback on tactic scripts.

In our comparison in section 4, we include two additional OCaml plugins of note. The standard library's `quote` plugin [7] inverts a simple denotation function to construct an OCaml reification routine; it does not handle binders. The template-coq plugin [1] is an OCaml reification plugin to an inductive datatype that mirrors Coq's underlying representation of terms. The biggest overhead in this method of reification is allocation, but reifying to template-coq's de Bruijn AST and then compiling from there to PHOAS is still quite fast.

3 Reification by Parametricity

So far, all of the reification we have seen operates by walking the Gallina syntax tree, reifiying each node into a PHOAS syntax tree node. There is a way of factoring this process into two passes over the syntax tree, 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.

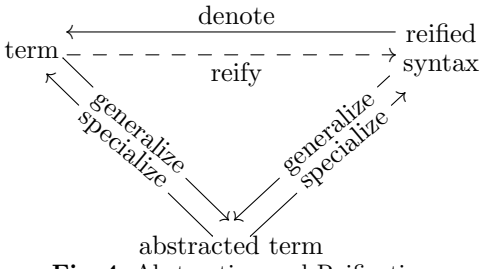


Fig. 4. Abstraction and Reification

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

3.1 Explanation By Examples

Reification Without Binders. 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:

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

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.

Reification With Binders. We can also reify expressions involving binders with this method. If we start with the expression

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

we can take advantage of the fact that we are using a definition for `let ... in ...`. We can abstract over our definition (`@Let_In` $\mathbb{N}$ $\mathbb{N}$), S, O, mul, and $\mathbb{N}$. We get a function of one type argument and four value arguments:

$$\begin{aligned}\Lambda N.\, \lambda\ &(\textsc{Mul} : N \to N \to N).\ \lambda(\mathrm{O} : N).\ \lambda(\mathrm{S} : N \to N).\\ &\lambda(\textsc{LetIn} : N \to (N \to N) \to N).\ \textsc{LetIn}\ (\mathrm{S}\ \mathrm{O})\ (\lambda x.\ \textsc{Mul}\ x\ x)\end{aligned}$$

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 $(\lambda v\ f.\ \texttt{LetIn}\ v\ (\lambda x.\ f\ (\texttt{Var}\ x)))$ for LetIn to obtain the PHOAS syntax tree

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

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.[6] 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.

[6] 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, and then reify the resulting partially reduced term.

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.
```

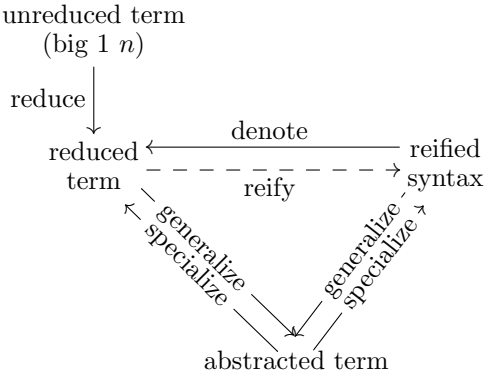
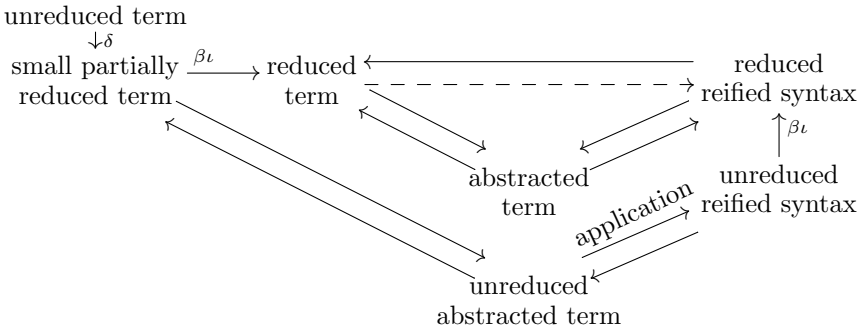


Fig. 5. Abstraction, Reification, Reduction

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, and a $\beta\iota$ step that reduces applications of λs to variables 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 commutative diagram thus becomes



Let us step through this alternate 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 $\Lambda T.\ T \to (\texttt{count} \to T \to T) \to \texttt{count} \to T$. Per-

forming δ-reduction, that is, unfolding `big`, gives us the small partially reduced term

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

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

$$\begin{aligned}&\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N)(\textsc{O} : N)(\textsc{S} : N \to N)(\textsc{LetIn} : N \to (N \to N) \to N).\\ &\quad\big(\lambda(x : N).\ \lambda(n : \texttt{count}).\ \texttt{count\_rec}\ (N \to N)\ (\lambda x.\ x)\\ &\qquad(\lambda n'.\ \lambda \texttt{big}_{n'}.\ \lambda x.\ \textsc{LetIn}\ (\textsc{Mul}\ x\ x)\ (\lambda x'.\ \texttt{big}_{n'}\ x'))\big)\\ &\quad(\textsc{S}\ \textsc{O})\ \ 100\end{aligned}$$

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` $\to$ `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.[7]

3.3 Implementation in Ltac

Unfortunately, Coq does not have a tactic that performs abstraction.[8] However, the `pattern` tactic suffices; it performs abstraction followed by application, and

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

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

is 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 Appendix D.

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).

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

4 Performance Comparison

We have performed a performance comparison of the various methods of reification to the PHOAS language `@expr var` from Figure 1.3. 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 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 tarball or to Appendix E for the exact code of each reification method being tested; the definition `big_flat` in Appendix E.17 (`BenchmarkUtil.v`) defines the term being reified.

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

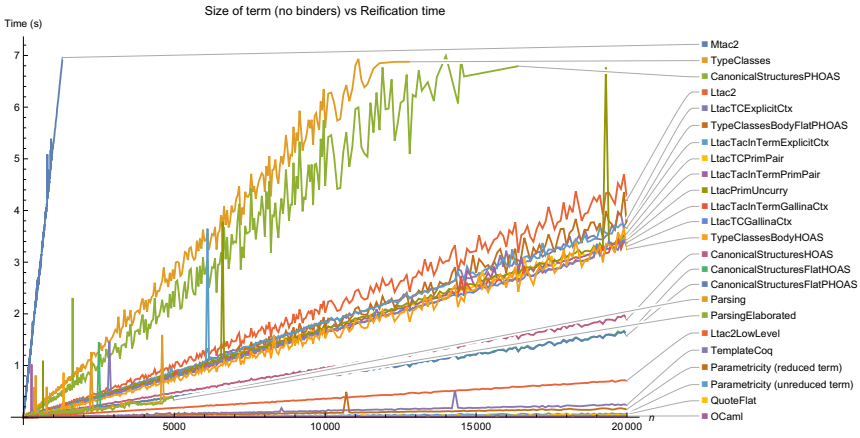


Fig. 6. Performance of Reification without Binders

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 tarball and Appendix E; 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 with no holes. 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.

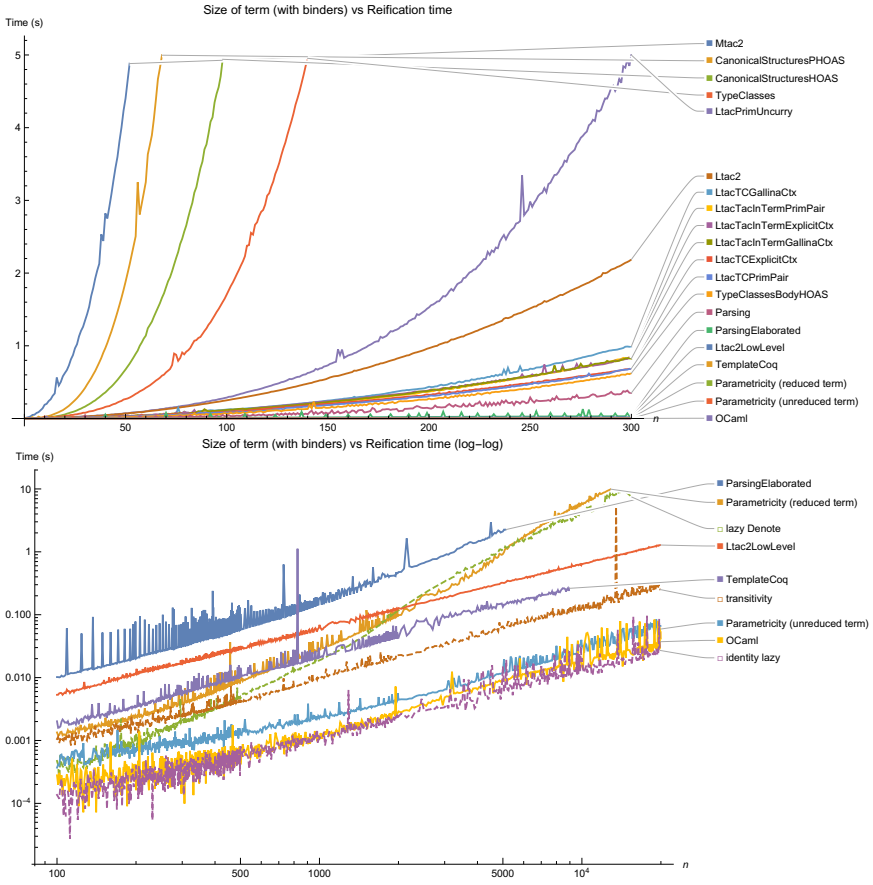


Fig. 7. Performance of Reification with Binders

4.2 With Binders

We look at terms of the form `dlet` a_1 `:= 1 * 1 in dlet` a_2 `:=` a_1 `*` a_1 `in ... dlet` a_n `:=` a_{n-1} `*` a_{n-1} `in` 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 overflow.

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 [12].

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.

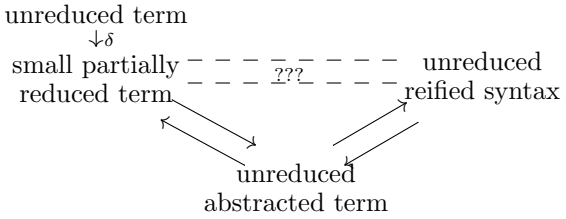


Fig. 8. Completing the commutative triangle

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 an abstracted, unreduced term, and 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

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

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

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

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

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

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.

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
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), 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. Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (December 2013), http://adam.chlipala.net/cpdt/
6. Coq Development Team: The Coq Proof Assistant Reference Manual, chap. 2.1.1 Extensions of Gallina, Record Types (Primitive Projections). INRIA, 8.7.1 edn. (2017), https://coq.inria.fr/distrib/V8.7.1/refman/gallina-ext.html#sec65
7. 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
8. 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/
9. 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://people.mpi-sws.org/~beta/lessadhoc/lessadhoc-extended.pdf
10. Leivent, J.: [coqdev] beta1 and zeta1 reduction (Jan 2016), https://sympa.inria.fr/sympa/arc/coqdev/2016-01/msg00060.html
11. Malecha, G., Chlipala, A., Braibant, T., Hulin, P., Yang, E.Z.: Mirrorshard: Proof by computational reflection with verified hints. CoRR abs/1305.6543 (2013), http://arxiv.org/abs/1305.6543
12. Pédrot, P.M.: Fast rel lookup #6506 (Dec 2017), https://github.com/coq/coq/pull/6506
13. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proc. PLDI. pp. 199–208 (1988), https://www.cs.cmu.edu/~fp/papers/pldi88.pdf
14. 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
15. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in Coq. Journal of Functional Programming 25 (2015), http://plv.mpi-sws.org/mtac/journal-draft.pdf
16. Ziliani, B., Sozeau, M.: A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. Journal of Functional Programming 27 (2017), https://people.mpi-sws.org/~beta/papers/unicoq-journal.pdf

A Acknowledgments and Historical Notes

We would like to thank Hugo Herbelin for sharing the trick with `type of` to propagate universe constraints[9] as well as useful conversations on Coq's bug tracker that allowed us to track down performance issues.[10] 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.

For those interested in history, our method of reification by parametricity was inspired by the `evm_compute` tactic. [11] 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 reification by parametricity was born.

B More Detailed Performance Graphs

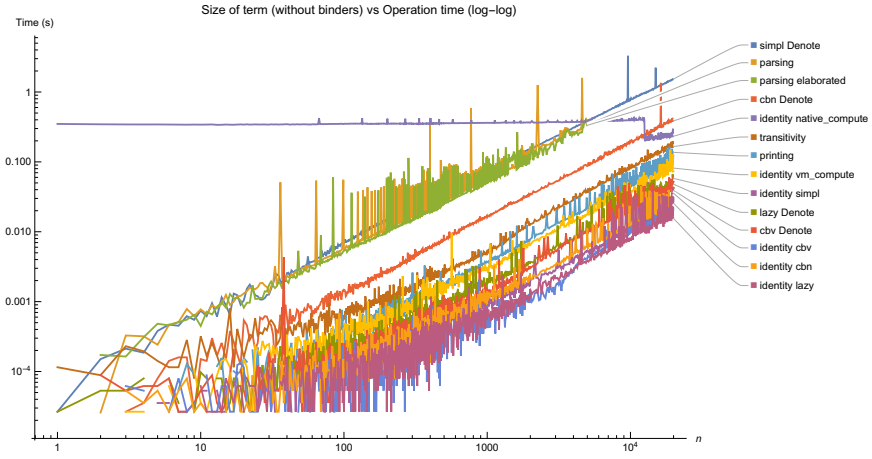
B.1 Without Binders

We look at terms of the form `1 * 1 * 1 * ...`. When we use n to denote the *size of the term*, we are taking n to be the number of 1s.

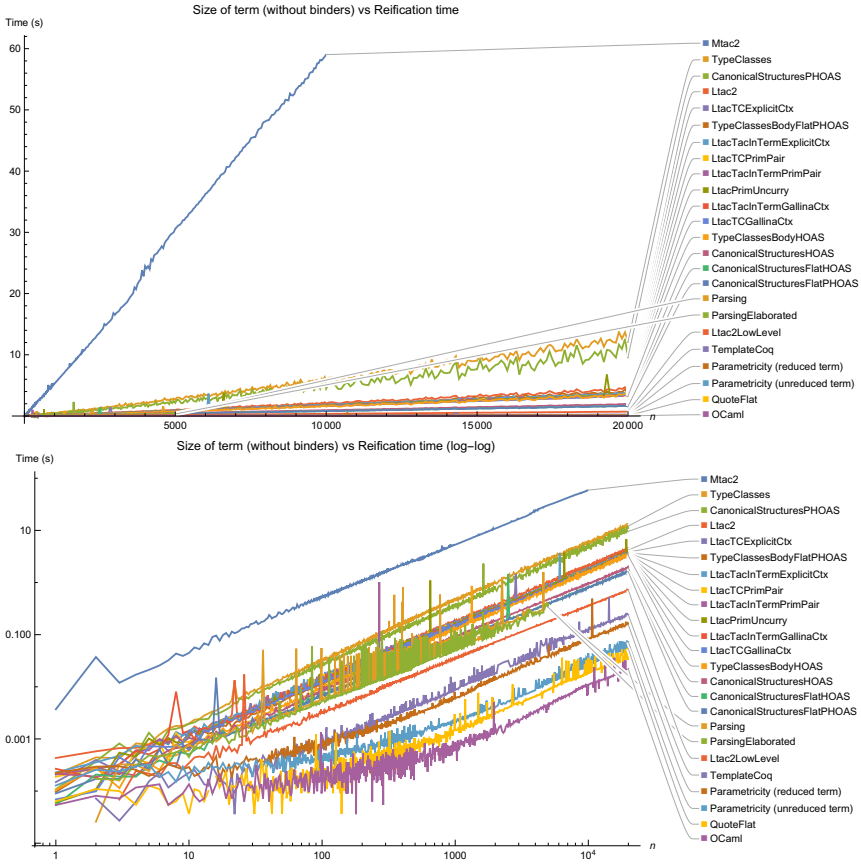
B.1.1 Baseline. We start by taking a look at some standard operations, to establish a baseline.

We can time the various reduction machines (`cbv`, `lazy`, `simpl`, `cbn`, `vm_compute`, `native_compute`) on an already-fully-normalized reified term. We can time the various customizable reduction machines (`cbv`, `lazy`, `simpl`, `cbn`) on unfolding the denotation function. We can time the `transitivity` step mentioned above. We can time how long it takes to parse and typecheck a reified term ("reification by copy-paste"), either when we have fully elaborated it to have no holes or when we have left holes and given it to Coq via notations. Finally, we can time how long it takes to print (e.g., with `idtac`) a reified term.

[9] `https://github.com/coq/coq/issues/5996#issuecomment-338405694`
[10] `https://github.com/coq/coq/issues/6252`



B.1.2 Reification We present more complete versions (both log-log and linear) of the graph in subsection 4.1.

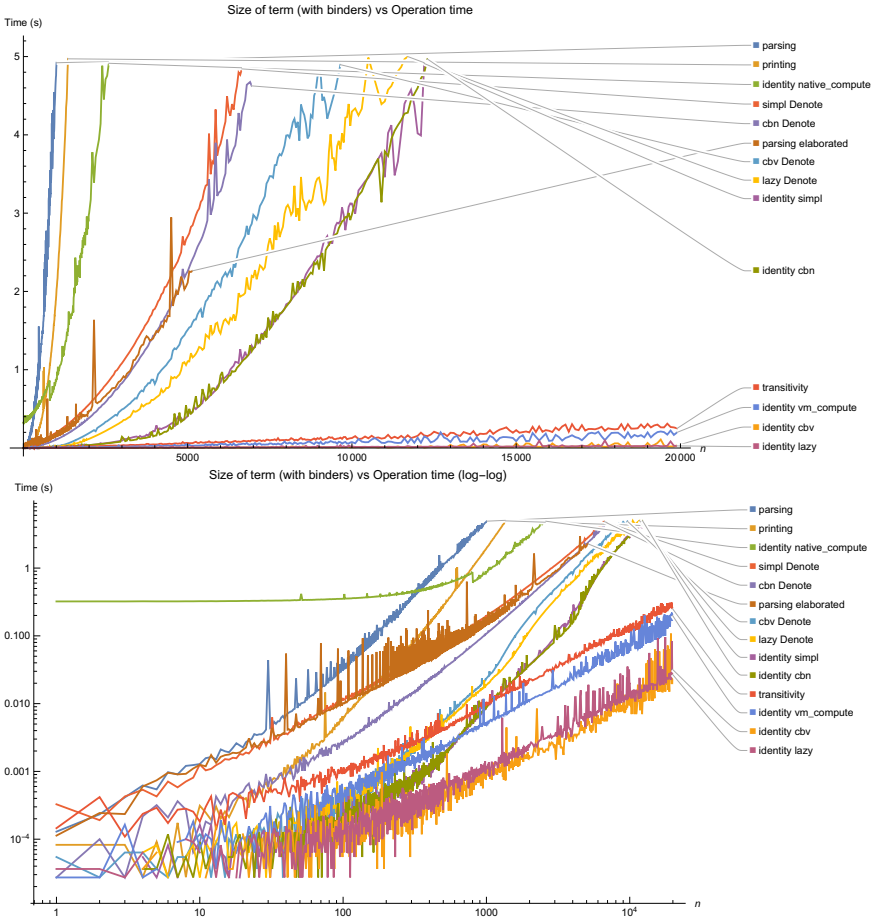


Note that `native_compute` is roughly constant, before it drops at large terms. This drop corresponds to one of the term size where we restart Coq before continuing (12 500), but we have no explanation for this behavior.

B.2 With Binders

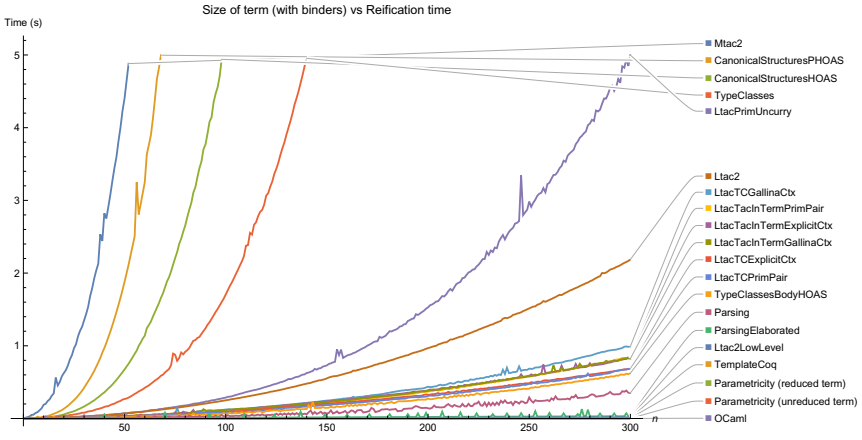
We look at terms of the form `dlet` a_1 `:= 1 * 1 in dlet` a_2 `:=` a_1 `*` a_1 `in ... dlet` a_n `:=` a_{n-1} `*` a_{n-1} `in` a_n.

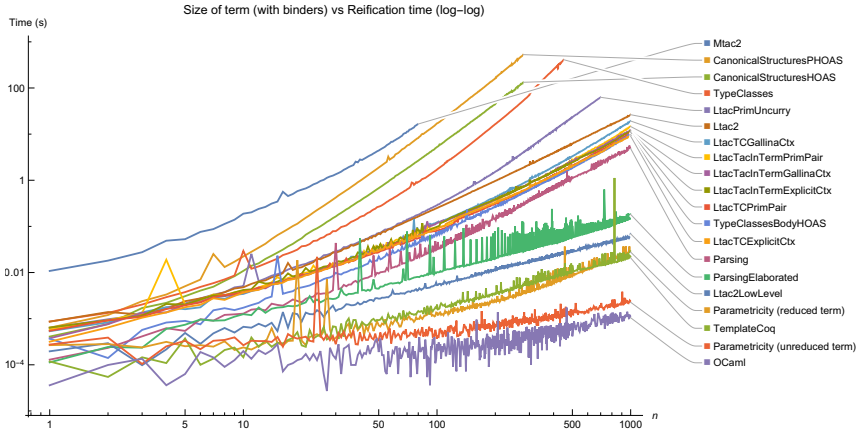
B.2.1 Baseline. We once again look at the baseline operations.

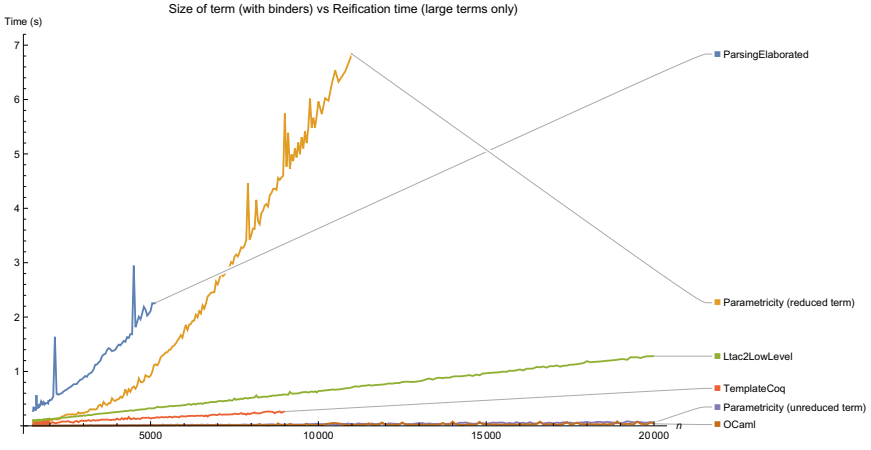


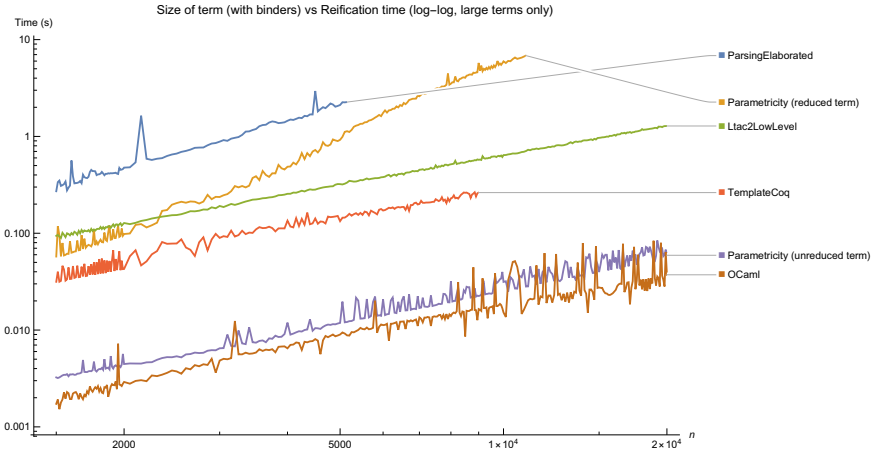
Note that `native_compute` once again has discontinuous drops at the location where we restart Coq (term size 800).

B.2.2 Reification We present more complete versions (both log-log and linear) of the graphs in subsection 4.2.


Size of term (with binders) vs Reification time
Time (s)
5
4
3
2
1
50
100
150
200
250
300
n
Mtac2
CanonicalStructuresPHOAS
CanonicalStructuresHOAS
TypeClasses
LtacPrimUncurry
Ltac2
LtacTCGallinaCtx
LtacTacInTermPrimPair
LtacTacInTermExplicitCtx
LtacTacInTermGallinaCtx
LtacTCExplicitCtx
LtacTCPrimPair
TypeClassesBodyHOAS
Parsing
ParsingElaborated
Ltac2LowLevel
TemplateCoq
Parametricity (reduced term)
Parametricity (unreduced term)
OCaml

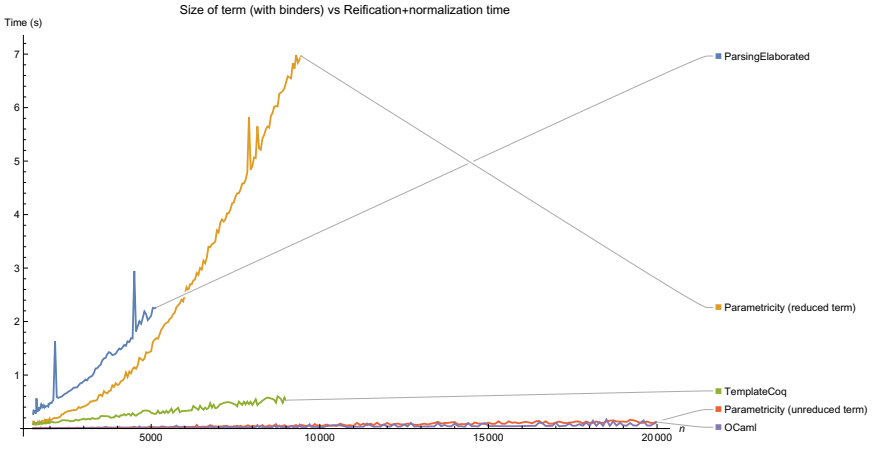

Size of term (with binders) vs Reification time (log–log)
Time (s)
100
1
0.01
10^-4
1
5
10
50
100
500
1000
n
Mtac2
CanonicalStructuresPHOAS
CanonicalStructuresHOAS
TypeClasses
LtacPrimUncurry
Ltac2
LtacTCGallinaCtx
LtacTacInTermPrimPair
LtacTacInTermGallinaCtx
LtacTacInTermExplicitCtx
LtacTCPrimPair
TypeClassesBodyHOAS
LtacTCExplicitCtx
Parsing
ParsingElaborated
Ltac2LowLevel
Parametricity (reduced term)
TemplateCoq
Parametricity (unreduced term)
OCaml

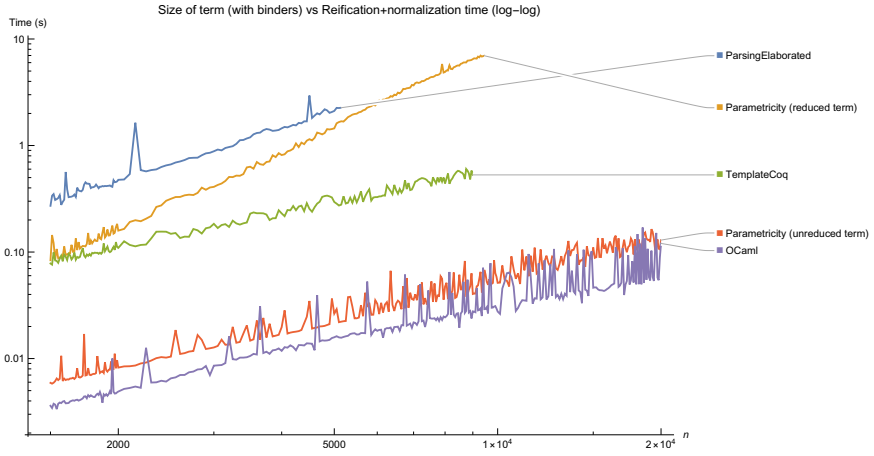

Size of term (with binders) vs Reification time (large terms only)
Time (s)
7
6
5
4
3
2
1
5000
10000
15000
20000
n
ParsingElaborated
Parametricity (reduced term)
Ltac2LowLevel
TemplateCoq
Parametricity (unreduced term)
OCaml



Note that Ltac2 is about 2X slower than Ltac reificiation; we hypothesize that this difference is due to the overhead of allocating twice as many evars (`Constr.in_context` allocates two evars, while we are careful to only allocate one evar per recursive call in the Ltac reification methods).

B.2.3 Reification to normal form We present versions of the graphs from subsection 4.2 which include the time it takes to normalize the reified term.





C Canonical Structures Reification in Detail

Let us walk through a simple reification problem. Note that canonical structures cannot be overlapping: for each field, the head constant of that field must be distinct across all definitions declared canonical for that field. We set up a number of tags and structures (code also in `CanonicalStructuresFlatHOAS.v`):

```
Structure tagged_nat := tag { untag : nat }.
Structure reified_of var :=
  reify { nat_of : tagged_nat ; reified_nat_of : @expr var }.
(* tags to control the order of application *)
Definition S_tag := tag.
Definition O_tag := S_tag.
Canonical Structure mul_tag n := O_tag n.
Canonical Structure reify_O var
  := reify (O_tag O) (@NatO var).
Canonical Structure reify_S var x
  := reify (S_tag (S (untag (nat_of x)))) (@NatS var (reified_nat_of x)).
Canonical Structure reify_mul var x y
  := reify (mul_tag (untag (nat_of x) * untag (nat_of y)))
           (@NatMul var (reified_nat_of x) (reified_nat_of y)).
```

Suppose we are faced with the goal

```
untag (@nat_of var ?e) = S (0 * 0)
```

We solve this goal with `refine eq_refl`, but let us step through the beginning of what Coq does.

1. It sees `untag _= _` and so applies the canonical structure `mul_tag` to the right hand side, trying to solve the unification problem:

2. `nat_of ?e = mul_tag (S (0 * 0))` results in attempting to apply the canonical structure `reify_mul`. However, `mul_tag (untag (nat_of ?x) * untag (nat_of ?y))` does not unify with `mul_tag (S (0 * 0))`, and so unification backtracks, unfolding `mul_tag`, giving:
3. `nat_of ?e = O_tag (S (0 * 0))`, which results in attempting to apply the canonical structure `reify_O` for `O_tag`. This, too, fails unification, and so `O_tag` is unfolded, giving:
4. `nat_of ?e = S_tag (S (0 * 0))`, which results in attempting to apply the canonical structure `reify_S` for `S_tag`. This succeeds at the first level, instantiating `?e` with `S_tag (S (untag (nat_of ?x)))` for a fresh evar `?x`, spawning a new unification problem:
5. `untag (nat_of ?x) = 0 * 0`. Here we again insert the canonical structure `mul_tag` and again try the canonical structure `reify_mul`, which this time succeeds, spawning two new problems where `untag (nat_of _)` must be unified with `0`. These, too, eventually succeed, through a somewhat drawn-out (but very fast) process.

To reify binders, we can do something similar but pass around an explicit list of binders that exist in the context. Note that the order of tags is important; `var_tl_tag` should be the last one checked, because it is the only one with a recursive call that can be expected to fail. Note that reification by canonical structures for terms with binders tends to be fairly slow. Additionally, it suffers from an additional complexity: reifying `let ... in ...` nodes seems to require "locking" the definition for `Let_In`.[11] If we do not block reduction with an opaque constant, Coq's unification will loop on inferring the identity function again and again.

As we are inexperienced with reification by canonical structures, especially in the presence of binders, there are likely more opportunities to optimize this code, especially by playing with the tag order. Code can be found in Appendix E.8.4.

D Reification by Parametricity in Ltac

We have glossed over two points in this description.

First: if the type to be reified lives in `Set` while the expression type lives in `Type`, Coq will report a universe inconsistency. Thus we have to dynamically change the `Set` binder for a `Type` binder, after which we invoke Ltac's `type of` construct for its side effect of propagating universe constraints.

This gives us the tactic:

```
Ltac Reify x :=
  let rx :=
   lazymatch (eval pattern nat, O, S, Nat.mul, (@Let_In nat nat) in x) with
   | ?rx _ _ _ _ _ => rx
```

[11] See "Locking, unlocking" in *A Small Scale Reflection Extension for the* Coq *system.* [8]

```
   end in
  let rx :=
   lazymatch rx with fun N : Set => ?rx => constr:(fun N : Type => rx) end in
  let __ := type of rx in (* propagate universe constraints *)
  let rx := constr:(fun var : Type
   => rx (@expr var) (@NatO var) (@NatS var) (@NatMul var)
       (fun x' f' => @LetIn var x' (fun v => f' (@Var var v)))) in
  rx.
```

Second: we have tacitly taken advantage of the fact that we had constructors for `O` and `S`. That is, we have taken advantage of the fact that our expression language is

```
Inductive expr {var : Type} :=
| NatO : expr
| NatS : expr -> expr
| NatMul : expr -> expr -> expr
| Var : var -> expr
| LetIn : expr -> (var -> expr) -> expr.
```

rather than

```
Inductive expr {var : Type} :=
| NatConst : nat -> expr
| NatMul : expr -> expr -> expr
| Var : var -> expr
| LetIn : expr -> (var -> expr) -> expr.
```

This language gives us a bit more trouble, because there are some terms that show up in our pre-reified expression which we do not directly reify (for example `S`). Thus, if we simply try to abstract over `S` and `O`, we cannot find reified terms to substitute them with, and if we do not abstract over `S`, then we need to list out all of the constants that show up.

For completeness, we present a recursive Ltac routine that does this, though we advise users of this method to cleanly separate between recursive types that are reified and recursive types that are not.[12]

```
(** expects:
    - [var] - the PHOAS var type
    - [find_const term found_tac not_found_tac], a tactical which
      looks for constants in [term], invokes [found_tac] with the
      constant if it finds one, and invokes [not_found_tac ()] if it
      finds none.
    - [plug_const var term const], a tactic which takes a term and a
```

[12] For example, if we had been reifying `Z` expressions rather than `nat` expressions, we could have a single constant node, and reify the constructors `Z0`, `Zpos`, and `Zneg`, without reifying `positive`.

```
      constant, and plugs in the reified version of [const] *)
Ltac reify_with_consts var find_const plug_const term :=
  find_const
    term
    ltac:(fun c
          => let rx := lazymatch (eval pattern c in term) with
                       | ?term _ => term
                       end in
             let rx := reify_with_consts find_const plug_const term in
             plug_const var rx c)
           ltac:(fun _
  => let rx :=
      match (eval pattern nat, Nat.mul, O, S, (@Let_In nat nat) in term) with
      | ?rx _ _ _ _ _ => rx
      end in
     let rx :=
      lazymatch rx with fun N : Set => ?rx => constr:(fun N : Type => rx) end in
     let __ := type of rx in (* propagate universe constraints *)
     constr:(rx (@expr var) (@NatMul var) (@NatO var) (@NatS var)
                (fun x' f' => @LetIn var x' (fun v => f' (@Var var v))))).
```

As we are advising against using recursion to reify constants, we do not evaluate the performance of this more complicated variant of reification by parametricity.

E Detailed Code Examples

All code found in this section is available in the associated code tarball.

For completeness, we include the introductory code necessary to compile the rest of the code.

E.1 Common Notations for Reification By Parametricity (`Common.v`)

E.1.1 Introductory Notations

```
Global Set Implicit Arguments.
Reserved Notation "'dlet' x := v 'in' f"
         (at level 200, f at level 200,
          format "'dlet'  x  :=  v  'in' '//' f").
Reserved Notation "'nllet' x := v 'in' f"
         (at level 200, f at level 200,
          format "'nllet'  x  :=  v  'in' '//' f").
Reserved Notation "'elet' x := v 'in' f"
         (at level 200, f at level 200,
          format "'elet'  x  :=  v  'in' '//' f").
Definition Let_In {A B} (v : A) (f : A → B) : B
  := let x := v in f x.
Notation "'dlet' x := v 'in' f" := (Let_In v (fun x ⇒ f)).
Definition key : unit. exact tt. Qed.
Definition lock {A} (v : A) : A := match key with tt ⇒ v end.
Lemma unlock {A} (v : A) : lock v = v.
Proof. unfold lock; destruct key; reflexivity. Qed.
Definition LockedLet_In_nat : nat → (nat → nat) → nat
  := lock (@Let_In nat nat).
Definition locked_nat_mul := lock Nat.mul.
Notation "'nllet' x := v 'in' f"
  := (LockedLet_In_nat v (fun x ⇒ f)).
Definition lock_Let_In_nat : @Let_In nat nat = LockedLet_In_nat
  := eq_sym (unlock _).
Definition lock_Nat_mul : Nat.mul = locked_nat_mul
  := eq_sym (unlock _).
```

E.2 Expression trees in PHOAS (PHOAS.v)

```
Require Import Reify.Common.

Inductive expr {var : Type} : Type :=
| NatO : expr
| NatS : expr → expr
| LetIn (v : expr) (f : var → expr)
| Var (v : var)
| NatMul (x y : expr).

Bind Scope expr_scope with expr.
Delimit Scope expr_scope with expr.

Infix "*" := NatMul : expr_scope.
Notation "'elet' x := v 'in' f" := (LetIn v (fun x ⇒ f%expr)) : expr_scope.
Notation "$$ x" := (Var x) (at level 9, format "$$ x") : expr_scope.

Fixpoint denote (e : @expr nat) : nat
  := match e with
     | NatO ⇒ O
     | NatS x ⇒ S (denote x)
     | LetIn v f ⇒ dlet x := denote v in denote (f x)
     | Var v ⇒ v
     | NatMul x y ⇒ denote x × denote y
     end.

Definition Expr := ∀ var, @expr var.
Definition Denote (e : Expr) := denote (e _).
```

E.3 Factored code common to many variants of reification (ReifyCommon.v)

```
Require Import Reify.NamedTimers.
Require Export Reify.Common.
Require Export Reify.PHOAS.

Notation do_transitivity := false (only parsing).
```

We provide a tactic to run a tactic in a constr context.

```
Ltac crun tac :=
  match goal with
  | _ ⇒ tac
  end.
```

Note: If you want to preserve variable names on reification, there are many hoops to jump through. We write a `refresh` tactic which permits preserving binder names at a nontrivial performance overhead.

c.f. https://github.com/coq/coq/issues/5448, https://github.com/coq/coq/issues/6315, https://github.com/coq/coq/issues/6559

```
Ltac require_same_var n1 n2 :=
  let c1 := constr:(fun n1 n2 : Set ⇒ ltac:(exact n1)) in
  let c2 := constr:(fun n1 n2 : Set ⇒ ltac:(exact n2)) in
  first [ constr_eq c1 c2
        | fail 1 "Not the same var:" n1 "and" n2 "(via constr_eq" c1 c2 ")" ].
Ltac is_same_var n1 n2 :=
  match goal with
  | _ ⇒ let __ := match goal with _ ⇒ require_same_var n1 n2 end in
        true
  | _ ⇒ false
  end.
Ltac is_underscore v :=
  let v' := fresh v in
  let v' := fresh v' in
  is_same_var v v'.
```

Note that `fresh_tac` must be `ltac:(fun n ⇒ fresh n)`; c.f. https://github.com/coq/coq/issues/6559

```
Ltac refresh n fresh_tac :=
  let n_is_underscore := is_underscore n in
  let n' := lazymatch n_is_underscore with
            | true ⇒ fresh
            | false ⇒ fresh_tac n
            end in
  let n' := fresh_tac n' in
  n'.
```

However, this comes at a significant cost in speed, so we do not try to preserve variable names, and this tactic is unused in our benchmark.

```
Ltac Reify_of reify x :=
  constr:(fun var : Type ⇒ ltac:(let v := reify var x in exact v)).

Ltac if_doing_trans tac :=
  let do_trans := constr:(do_transitivity) in
  lazymatch do_trans with
  | true ⇒ tac ()
  | false ⇒ idtac
  end.
```

We ask for dummy arguments for most things, because it is good practice to indicate that this tactic should not be run at the call-site (when it's passed to another tactic), but at the use-site.

```
Ltac do_Reify_rhs_of_cps_with_denote Reify_cps Denote _ :=
  let v := lazymatch goal with ⊢ ?LHS = ?v ⇒ v end in
  let __ := crun ltac:(restart_timer "norm reif") in
  let __ := crun ltac:(restart_timer "actual reif") in
  Reify_cps v ltac:(
    fun rv
    ⇒ let __ := crun ltac:(finish_timing ("Tactic call") "actual reif") in
      let __ := crun ltac:(restart_timer "eval lazy") in
      let rv := (eval lazy in rv) in
      let __ := crun ltac:(finish_timing ("Tactic call") "eval lazy") in
      let __ := crun ltac:(finish_timing ("Tactic call") "norm reif") in
      time "lazy beta iota" lazy beta iota;
      if_doing_trans
        ltac:(fun _
              ⇒ time "transitivity (Denote rv)"
                     transitivity (Denote rv))).
Ltac do_Reify_rhs_of_cps Reify_cps _ :=
  do_Reify_rhs_of_cps_with_denote Reify_cps Denote ().
Ltac do_Reify_rhs_of_with_denote Reify Denote _ :=
  do_Reify_rhs_of_cps_with_denote
    ltac:(fun v tac ⇒ let rv := Reify v in tac rv) Denote ().
Ltac do_Reify_rhs_of Reify _ :=
  do_Reify_rhs_of_with_denote Reify Denote ().
Ltac post_Reify_rhs _ :=
  [ > ..
  | if_doing_trans ltac:(fun _ ⇒ lazy [Denote denote]; reflexivity) ].
Ltac Reify_rhs_of_cps Reify_cps _ :=
  do_Reify_rhs_of_cps Reify_cps (); post_Reify_rhs ().
Ltac Reify_rhs_of Reify _ :=
  do_Reify_rhs_of Reify (); post_Reify_rhs ().

Ltac error_cant_elim_deps f :=
  let __ := match goal with
            | _ ⇒ idtac "Failed to eliminate functional dependencies in" f
```

```
            end in
  constr:(I : I).

Ltac error_bad_function f :=
  let __ := match goal with
            | _ ⇒ idtac "Bad let-in function" f
            end in
  constr:(I : I).

Ltac error_bad_term term :=
  let __ := match goal with
            | _ ⇒ idtac "Unrecognized term:" term
            end in
  let ret := constr:(term : I) in
  constr:(I : I).
```

E.4 Define a primitive pairing type (PrimPair.v)

```
Set Primitive Projections.
Record prod A B := pair { fst : A ; snd : B }.
Add Printing Let prod.

Arguments pair {A B} _ _.
Arguments fst {A B} _.
Arguments snd {A B} _.

Notation "x * y" := (prod x y) : type_scope.
Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) : core_scope.
```

E.5 Reification by canonical structures (CanonicalStructuresReifyCommon.v)

```
Require Import Reify.NamedTimers.
Require Import Reify.Common.
Require Export Reify.ReifyCommon.
Require Import Reify.PHOAS.
```

Take care of initial locking of mul, letin, etc.

```
Ltac make_pre_Reify_rhs nat_of untag do_lock_letin do_lock_natmul :=
  let RHS := lazymatch goal with ⊢ _ = ?RHS ⇒ RHS end in
  let e := fresh "e" in
  let T := fresh in
  evar (T : Type);
  evar (e : T);
  subst T;
  cut (untag (nat_of e) = RHS);
  [ subst e
  | lazymatch do_lock_letin with
    | true ⇒ rewrite ?lock_Let_In_nat
    | false ⇒ idtac
    end;
    lazymatch do_lock_natmul with
    | true ⇒ rewrite ?lock_Nat_mul
    | false ⇒ idtac
    end;
    cbv [e]; clear e ].
```

N.B. we must thunk the constants so as to not focus the goal

```
Ltac make_do_Reify_rhs denote reified_nat_of postprocess :=
  [ >
  | restart_timer "norm reif";
    time "actual reif" refine eq_refl ];
  let denote := denote () in
  let reified_nat_of := reified_nat_of () in
  let e := lazymatch goal with ⊢ ?untag (?nat_of ?e) = _ → ?LHS = _ ⇒ e end in
  let __ := crun ltac:(restart_timer "eval lazy") in
  let e' := (eval lazy in (reified_nat_of e)) in
  let __ := crun ltac:(finish_timing ("Tactic call") "eval lazy") in
  let __ := crun ltac:(restart_timer "postprocess") in
  let e' := postprocess e' in
  let __ := crun ltac:(finish_timing ("Tactic call") "postprocess") in
  let __ := crun ltac:(finish_timing ("Tactic call") "norm reif") in
  time "intros _" intros _;
  time "lazy beta iota" lazy beta iota;
  if_doing_trans ltac:(fun _ ⇒ time "transitivity (Denote rv)"
                                    transitivity (denote e')).
```

E.6 Typeclass-based Reification

E.6.1 Typeclass-based reification (TypeClasses.v)

```
Require Import Reify.ReifyCommon.

Local Generalizable Variables x y rx ry f rf.
Section with_var.
  Context {var : Type}.

  Class reify_of (term : nat) (rterm : @expr var) := {}.
  Global Instance reify_NatMul `{reify_of x rx, reify_of y ry}
    : reify_of (x × y) (rx × ry).
  Global Instance reify_LetIn `{reify_of x rx}
         `{∀ y ry, reify_of y (Var ry) → reify_of (f y) (rf ry)}
    : reify_of (dlet y := x in f y) (elet ry := rx in rf ry).
  Global Instance reify_S `{reify_of x rx}
    : reify_of (S x) (NatS rx).
  Global Instance reify_O
    : reify_of O NatO.
End with_var.
```

This must be commented out pre-8.6; it tells Coq to not try to infer reifications if it doesn't fully know what term it's reifying.

```
Hint Mode reify_of - ! - : typeclass_instances.
Hint Opaque Nat.mul Let_In : typeclass_instances.

Ltac reify var x :=
  let c := constr:(_ : @reify_of var x _) in
  lazymatch type of c with
  | reify_of _ ?rx ⇒ rx
  end.

Ltac Reify x :=
  let c := constr:(fun var ⇒ (_ : @reify_of var x _)) in
  lazymatch type of c with
  | ∀ var, reify_of _ (@?rx var) ⇒ rx
  end.

Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.6.2 Typeclass-based reification (TypeClassesBodyFlatPHOAS.v)

We can also do typeclass-based reification where we return the reified term in the body rather than in the type. However, this method does not work well with binders, because there's no easy way to eliminate the dependency on the unreified binder when reifying to PHOAS.

```
Require Import Reify.ReifyCommon.

Local Generalizable Variables x y rx ry f rf.
Section with_var.
  Context {var : Type}.

  Class reify_of (term : nat) := rterm : @expr var.
  Global Instance reify_NatMul `{rx : reify_of x, ry : reify_of y}
    : reify_of (x × y) := (rx × ry)%expr.
  Global Instance reify_S `{rx : reify_of x}
    : reify_of (S x) := NatS rx.
  Global Instance reify_O
    : reify_of O := NatO.
End with_var.
```

This must be commented out pre-8.6; it tells Coq to not try to infer reifications if it doesn't fully know what term it's reifying.

```
Hint Mode reify_of - ! : typeclass_instances.
Hint Opaque Nat.mul : typeclass_instances.

Ltac reify var x :=
  constr:(_ : @reify_of var x).

Ltac Reify x :=
  constr:(_ : ∀ var, @reify_of var x).

Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.6.3 Typeclass-based reification (TypeClassesBodyHOAS.v)

We can also do typeclass-based reification where we return the reified term in the body rather than in the type. However, this method does not work well with PHOAS binders, because there's no easy way to eliminate the dependency on the unreified binder when reifying to PHOAS.

```
Require Import Reify.ReifyCommon.

Local Generalizable Variables x y rx ry f rf.
Class reify_of (term : nat) := rterm : @expr nat.
```

We use | 100 so this gets triggered late.

```
Global Instance reify_Var {x} : reify_of x | 100 := Var x.
Global Instance reify_NatMul `{rx : reify_of x, ry : reify_of y}
  : reify_of (x × y) := (rx × ry)%expr.
Global Instance reify_S `{rx : reify_of x}
  : reify_of (S x) := NatS rx.
Global Instance reify_O
  : reify_of O := NatO.
Global Instance reify_LetIn `{rx : reify_of x}
       `{rf : ∀ y, reify_of (f y)}
  : reify_of (dlet y := x in f y) := (elet ry := rx in rf ry)%expr.
```

This must be commented out pre-8.6; it tells Coq to not try to infer reifications if it doesn't fully know what term it's reifying.

```
Hint Mode reify_of ! : typeclass_instances.
Hint Opaque Nat.mul Let_In : typeclass_instances.

Ltac Reify x :=
  constr:(_ : @reify_of x).

Ltac do_Reify_rhs _ := do_Reify_rhs_of_with_denote Reify denote ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.7 Ltac Reification

E.7.1 Ltac-based reification, using uncurrying to reucurse under binders (LtacPrimUncurry.v)

```
Require Import Reify.ReifyCommon.
Require Import Reify.PrimPair.
```

Points of note:

- We use primitive projections for pairing to speed up typing.
- Because we track variables by pairing `nat` binders with fresh `var` nodes, we use a `phantom` axiom of type `nat` to fill in the now-unused `nat` binder after reification.
- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- We make use of a trick from "[coqdev] beta1 and zeta1 reduction"[13] to bind names with a single-branch `match` statement without incurring extra β or ζ reductions.
- We give the `return` clause on the `match` statement explicitly to work around `https://github.com/coq/coq/issues/6252#issuecomment-347041995` and prevent extra backtracking, as well as preventing extra evar allocation.

```
Axiom phantom : nat.

Ltac reify var term :=
  let reify_rec term := reify var term in
  lazymatch term with
  | (fun args : ?T ⇒ O)
    ⇒ constr:(fun args : T ⇒ @NatO var)
  | (fun args : ?T ⇒ S (@?x args))
    ⇒ let rx := reify_rec x in
      constr:(fun args : T ⇒ @NatS var (rx args))
  | fun args : ?T ⇒ @?x args × @?y args
    ⇒ let rx := reify_rec x in
      let ry := reify_rec y in
      constr:(fun args : T ⇒ @NatMul var (rx args) (ry args))
  | (fun args : ?T ⇒ dlet x := @?v args in ?f)
    ⇒ let rv := reify_rec v in
      let args2 := fresh in
      let rf :=
          reify_rec
            (fun args2 : (nat × var) × T
             ⇒ match @snd (nat × var) T args2,
                      @fst nat var (@fst (nat × var) T args2)
                      return nat
```

[13] `https://sympa.inria.fr/sympa/arc/coqdev/2016-01/msg00060.html`

```
                with
                | args, x ⇒ f
                end) in
      constr:(fun args : T
              ⇒ @LetIn
                  var
                  (rv args)
                  (fun x : var
                   ⇒ rf (@pair (nat × var) T (@pair nat var phantom x) args)))
  | (fun args : ?T ⇒ @fst ?A ?B (@fst ?C ?D ?args'))
    ⇒ constr:(fun args : T ⇒ @Var var (@snd A B (@fst C D args')))
  | (fun args : ?T ⇒ _)
    ⇒ error_bad_term term
  | ?v
    ⇒ let rv := reify_rec (fun dummy : unit ⇒ v) in
      (eval lazy beta iota delta [fst snd] in (rv tt))
  end.
Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.7.2 Recursing under binders with typeclasses, tracking variables by pairing (LtacTCPrimPair.v)

```
Require Import Reify.ReifyCommon.
Require Import Reify.PrimPair.
```

Points of note:

- We use primitive projections for pairing to speed up typing.
- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- We make use of a trick from "[coqdev] beta1 and zeta1 reduction"[14] to bind names with a single-branch `match` statement without incurring extra β or ζ reductions.
- We give the `return` clause on the `match` statement explicitly to work around `https://github.com/coq/coq/issues/6252#issuecomment-347041995` and prevent extra backtracking, as well as preventing extra evar allocation.
- In the `Hint` used to tie the recursive knot, we run `intros` before binding any terms to avoid playing fast and loose with binders, because we will sometimes be presented with goals with unintroduced binders. If we did not call `intros` first, instead binding ?`var` and ?`term` in the hint pattern rule, they might contain unbound identifiers, causing reification to fail when it tried to deal with them.

```
Class reify_cls (var : Type) (term : nat) := do_reify : @expr var.

Ltac reify var term :=
  let reify_rec term := reify var term in
  lazymatch term with
  | fst (?term, ?v)
    ⇒ constr:(@Var var v)
  | _
    ⇒
    lazymatch term with
    | O ⇒ constr:(@NatO var)
    | S ?x
      ⇒ let rx := reify_rec x in
        constr:(@NatS var rx)
    | ?x × ?y
      ⇒ let rx := reify_rec x in
        let ry := reify_rec y in
        constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      ⇒ let rv := reify_rec v in
        let not_x := fresh in
        let not_x2 := fresh in
```

[14] `https://sympa.inria.fr/sympa/arc/coqdev/2016-01/msg00060.html`

```
          let rf
              :=
              lazymatch
                constr:(_ : ∀ (not_x : nat) (not_x2 : var),
                           @reify_cls
                             var
                             match @fst nat var (@pair nat var not_x not_x2)
                                   return nat
                             with
                             | x ⇒ f
                             end)
              with
              | fun _ ⇒ ?f ⇒ f
              | ?f ⇒ error_cant_elim_deps f
              end in
          constr:(@LetIn var rv rf)
      | ?v
        ⇒ error_bad_term v
      end
    end.
Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().

Global Hint Extern 0 (@reify_cls _ _)
⇒ (intros;
      lazymatch goal with
      | [ ⊢ @reify_cls ?var ?term ]
        ⇒ let res := reify var term in
          exact res
      end) : typeclass_instances.
```

E.7.3 Recursing under binders with typeclasses, tracking variables with Gallina contexts (`LtacTCGallinaCtx.v`)

```
Require Import Reify.ReifyCommon.
```

Points of note:

- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- We do not use a typeclass for the variable case to avoid typeclass search when it's not needed.
- In the `Hint` used to tie the recursive knot, we run `intros` before binding any terms to avoid playing fast and loose with binders, because we will sometimes be presented with goals with unintroduced binders. If we did not call `intros` first, instead binding ?`var` and ?`term` in the hint pattern rule, they might contain unbound identifiers, causing reification to fail when it tried to deal with them.

```
Class reify_cls (var : Type) (term : nat) := do_reify : @expr var.
```

Much like typeclass-based reification, we introduce a definition to track which `nat` binders reify to which `var` binders, searching the context for such hypotheses.

```
Definition var_for {var : Type} (n : nat) (v : var) := False.

Ltac reify var term :=
  let reify_rec term := reify var term in
  lazymatch goal with
  | [ H : var_for term ?v ⊢ _ ]
    ⇒ constr:(@Var var v)
  | _
    ⇒
    lazymatch term with
    | O ⇒ constr:(@NatO var)
    | S ?x
      ⇒ let rx := reify_rec x in
        constr:(@NatS var rx)
    | ?x × ?y
      ⇒ let rx := reify_rec x in
        let ry := reify_rec y in
        constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      ⇒ let rv := reify_rec v in
        let not_x := fresh in
        let rf
            :=
            lazymatch
              constr:(_ : ∀ (x : nat) (not_x : var)
```

```
                                    (_ : @var_for var x not_x),
                            @reify_cls var f)
                with
                | fun _ v' _ ⇒ @?f v' ⇒ f
                | ?f ⇒ error_cant_elim_deps f
                end in
            constr:(@LetIn var rv rf)
      | ?v
        ⇒ error_bad_term v
      end
  end.

Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().

Global Hint Extern 0 (@reify_cls _ _)
⇒ (intros;
      lazymatch goal with
      | [ ⊢ @reify_cls ?var ?term ]
        ⇒ let res := reify var term in
          exact res
      end) : typeclass_instances.
```

E.7.4 Recursing under binders with typeclasses, tracking variables with explicit contexts (`LtacTCExplicitCtx.v`)

```
Require Import Reify.ReifyCommon.
```

Points of note:

- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- In the `Hint` used to tie the recursive knot, we run `intros` before binding any terms to avoid playing fast and loose with binders, because we will sometimes be presented with goals with unintroduced binders. If we did not call `intros` first, instead binding ?`var` and ?`term` in the hint pattern rule, they might contain unbound identifiers, causing reification to fail when it tried to deal with them.

```
Module var_context.
  Inductive var_context {var : Type} :=
  | nil
  | cons (n : nat) (v : var) (xs : var_context).
End var_context.

Class reify_helper_cls (var : Type) (term : nat)
      (ctx : @var_context.var_context var)
  := do_reify_helper : @expr var.

Ltac reify_helper var term ctx :=
  let reify_rec term := reify_helper var term ctx in
  lazymatch ctx with
  | context[var_context.cons term ?v _]
    ⇒ constr:(@Var var v)
  | _
    ⇒
    lazymatch term with
    | O ⇒ constr:(@NatO var)
    | S ?x
      ⇒ let rx := reify_rec x in
        constr:(@NatS var rx)
    | ?x × ?y
      ⇒ let rx := reify_rec x in
        let ry := reify_rec y in
        constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      ⇒ let rv := reify_rec v in
        let not_x := fresh in
        let rf
            :=
            lazymatch
```

```
                constr:(_ : ∀ (x : nat) (not_x : var),
                           @reify_helper_cls
                             var f (@var_context.cons var x not_x ctx))
              with
              | fun _ ⇒ ?f ⇒ f
              | ?f ⇒ error_cant_elim_deps f
              end in
          constr:(@LetIn var rv rf)
    | ?v
      ⇒ error_bad_term v
    end
  end.
Ltac reify var x :=
  reify_helper var x (@var_context.nil var).
Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
Global Hint Extern 0 (@reify_helper_cls _ _ _)
⇒ (intros;
      lazymatch goal with
      | [ ⊢ @reify_helper_cls ?var ?term ?ctx ]
        ⇒ let res := reify_helper var term ctx in
          exact res
      end) : typeclass_instances.
```

E.7.5 Recursing under binders with tactics in terms, tracking variables by pairing (LtacTacInTermPrimPair.v)

```
Require Import Reify.ReifyCommon.
Require Import Reify.PrimPair.
```

Points of note:

- We use primitive projections for pairing to speed up typing.
- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- We must bind open terms to fresh variable names to work around the fact that tactics in terms do not correctly support open terms.[15]
- We make use of a trick from "[coqdev] beta1 and zeta1 reduction"[16] to bind names with a single-branch `match` statement without incurring extra β or ζ reductions.
- We must unfold aliases bound with this `match` statement trick (substitution does not happen until after typechecking), and if we are not careful with how we use `fresh`, Coq will stack overflow on `cbv delta` or otherwise misbehave.[17]
- We give the `return` clause on the `match` statement explicitly. Without the explicit return clause, Coq would backtrack on failure and attempt a second way of elaborating the `match` branches, resulting in a blowup on failure that is exponential in the recursive depth of the failure.[18] If we used `return _`, rather than specifying the type explicitly, we incur the cost of allocating an additional evar, which is linear in the size of the context. (This performance statistic courtesy of conversations with Pierre-Marie Pédrot on Coq's gitter.)
- We explicitly `clear` variable bindings from the context before invoking the recursive call, because the cost of evars is proportional to the size of the context.

```
Ltac reify var term :=
  let reify_rec term := reify var term in
  lazymatch term with
  | fst (?term, ?v)
    ⇒ constr:(@Var var v)
  | _
    ⇒
    lazymatch term with
    | O ⇒ constr:(@NatO var)
    | S ?x
```

[15] https://github.com/coq/coq/issues/3248
[16] https://sympa.inria.fr/sympa/arc/coqdev/2016-01/msg00060.html
[17] See https://github.com/coq/coq/issues/5448, https://github.com/coq/coq/issues/6315, https://github.com/coq/coq/issues/6559.
[18] https://github.com/coq/coq/issues/6252#issuecomment-347041995

```
      ⇒ let rx := reify_rec x in
        constr:(@NatS var rx)
    | ?x × ?y
      ⇒ let rx := reify_rec x in
        let ry := reify_rec y in
        constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      ⇒ let rv := reify_rec v in
        let not_x := fresh in
        let not_x2 := fresh in
        let not_x3 := fresh in
        let rf
            :=
            lazymatch
              constr:(
                fun (not_x : nat) (not_x2 : var)
                ⇒ match @fst nat var (@pair nat var not_x not_x2)
                        return @expr var
                   with
                   | x
                     ⇒ match f return @expr var with
                       | not_x3
                         ⇒ ltac:(
                             let fx := (eval cbv delta [not_x3 x] in not_x3) in
                             clear x not_x3;
                             let rf := reify_rec fx in
                             exact rf)
                       end
                   end)
            with
            | fun _ ⇒ ?f ⇒ f
            | ?f ⇒ error_cant_elim_deps f
            end in
        constr:(@LetIn var rv rf)
    | ?v
      ⇒ error_bad_term v
    end
  end.
Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.7.6 Recursing under binders with tactics in terms, tracking variables with Gallina contexts (`LtacTacInTermGallinaCtx.v`)

```
Require Import Reify.ReifyCommon.
```

Points of note:

- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- We must bind open terms to fresh variable names to work around the fact that tactics in terms do not correctly support open terms (see COQBUG(https://github.com/coq/coq/issues/3248)).
- We make use of a trick from "[coqdev] beta1 and zeta1 reduction"[19] to bind names with a single-branch `match` statement without incurring extra β or ζ reductions.
- We must unfold aliases bound with this `match` statement trick (substitution does not happen until after typechecking), and if we are not careful with how we use `fresh`, Coq will stack overflow on `cbv delta` or otherwise misbehave.[20]
- We give the `return` clause on the `match` statement explicitly. Without the explicit return clause, Coq would backtrack on failure and attempt a second way of elaborating the `match` branches, resulting in a blowup on failure that is exponential in the recursive depth of the failure.[21] If we used `return _`, rather than specifying the type explicitly, we incur the cost of allocating an additional evar, which is linear in the size of the context. (This performance statistic courtesy of conversations with Pierre-Marie Pédrot on Coq's gitter.)
- We explicitly `clear` variable bindings from the context before invoking the recursive call, because the cost of evars is proportional to the size of the context.

Much like typeclass-based reification, we introduce a definition to track which `nat` binders reify to which `var` binders, searching the context for such hypotheses.

```
Definition var_for {var : Type} (n : nat) (v : var) := False.

Ltac reify var term :=
  let reify_rec term := reify var term in
  lazymatch goal with
  | [ H : var_for term ?v ⊢ _ ]
    ⇒ constr:(@Var var v)
  | _
    ⇒
    lazymatch term with
```

[19] https://sympa.inria.fr/sympa/arc/coqdev/2016-01/msg00060.html

[20] See https://github.com/coq/coq/issues/5448, https://github.com/coq/coq/issues/6315, https://github.com/coq/coq/issues/6559.

[21] https://github.com/coq/coq/issues/6252#issuecomment-347041995

```
    | O ⇒ constr:(@NatO var)
    | S ?x
      ⇒ let rx := reify_rec x in
        constr:(@NatS var rx)
    | ?x × ?y
      ⇒ let rx := reify_rec x in
        let ry := reify_rec y in
        constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      ⇒ let rv := reify_rec v in
        let not_x := fresh  in
        let not_x2 := fresh  in
        let rf
            :=
            lazymatch
              constr:(
                fun (x : nat) (not_x : var) (_ : @var_for var x not_x)
                ⇒ match f return @expr var with
                  | not_x2
                    ⇒ ltac:(let fx := (eval cbv delta [not_x2] in not_x2) in
                              clear not_x2;
                              let rf := reify_rec fx in
                              exact rf)
                  end)
            with
            | fun _ v' _ ⇒ @?f v' ⇒ f
            | ?f ⇒ error_cant_elim_deps f
            end in
        constr:(@LetIn var rv rf)
    | ?v
      ⇒ error_bad_term v
    end
  end.
Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.7.7 Recursing under binders with tactics in terms, tracking variables with explicit contexts (LtacTacInTermExplicitCtx.v)

```
Require Import Reify.ReifyCommon.
```

Points of note:

- We make sure to fill in all implicit arguments explicitly, to minimize the number of evars generated; evars are one of the main bottlenecks.
- We must bind open terms to fresh variable names to work around the fact that tactics in terms do not correctly support open terms.[22]
- We make use of a trick from "[coqdev] beta1 and zeta1 reduction"[23] to bind names with a single-branch `match` statement without incurring extra β or ζ reductions.
- We must unfold aliases bound with this `match` statement trick (substitution does not happen until after typechecking), and if we are not careful with how we use `fresh`, Coq will stack overflow on `cbv delta` or otherwise misbehave.[24]
- We give the `return` clause on the `match` statement explicitly. Without the explicit return clause, Coq would backtrack on failure and attempt a second way of elaborating the `match` branches, resulting in a blowup on failure that is exponential in the recursive depth of the failure.[25] If we used `return _`, rather than specifying the type explicitly, we incur the cost of allocating an additional evar, which is linear in the size of the context. (This performance statistic courtesy of conversations with Pierre-Marie Pédrot on Coq's gitter.)
- We explicitly `clear` variable bindings from the context before invoking the recursive call, because the cost of evars is proportional to the size of the context.
- Note that we `match`-bind the new context because `x` shows up in it.[26]

```
Module var_context.
  Inductive var_context {var : Type} :=
  | nil
  | cons (n : nat) (v : var) (xs : var_context).
End var_context.

Ltac reify_helper var term ctx :=
  let reify_rec term := reify_helper var term ctx in
  lazymatch ctx with
  | context[var_context.cons term ?v _]
    ⇒ constr:(@Var var v)
```

[22] https://github.com/coq/coq/issues/3248
[23] https://sympa.inria.fr/sympa/arc/coqdev/2016-01/msg00060.html
[24] See https://github.com/coq/coq/issues/5448, https://github.com/coq/coq/issues/6315, https://github.com/coq/coq/issues/6559.
[25] https://github.com/coq/coq/issues/6252#issuecomment-347041995
[26] https://github.com/coq/coq/issues/3248

```
  | _
    ⇒
    lazymatch term with
    | O ⇒ constr:(@NatO var)
    | S ?x
      ⇒ let rx := reify_rec x in
        constr:(@NatS var rx)
    | ?x × ?y
      ⇒ let rx := reify_rec x in
        let ry := reify_rec y in
        constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      ⇒ let rv := reify_rec v in
        let not_x := fresh in
        let not_x2 := fresh in
        let not_x3 := fresh in
        let rf
            :=
            lazymatch
              constr:(
                fun (x : nat) (not_x : var)
                ⇒ match f, @var_context.cons var x not_x ctx
                          return @expr var
                  with
                  | not_x2, not_x3
                    ⇒ ltac:(let fx := (eval cbv delta [not_x2] in not_x2) in
                              let ctx := (eval cbv delta [not_x3] in not_x3) in
                              clear not_x2 not_x3;
                              let rf := reify_helper var fx ctx in
                              exact rf)
                  end)
            with
            | fun _ ⇒ ?f ⇒ f
            | ?f ⇒ error_cant_elim_deps f
            end in
        constr:(@LetIn var rv rf)
    | ?v
      ⇒ error_bad_term v
    end
  end.

Ltac reify var x :=
  reify_helper var x (@var_context.nil var).
Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
```

```
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.8 Canonical Structures Reification

E.8.1 Canonical-structure based reification (CanonicalStructuresFlatHOAS.v)

This version reifies to @expr nat, and does not support let-binders.

```
Require Import Reify.CanonicalStructuresReifyCommon.
```

structure for packaging a nat expr and its reification

```
Structure tagged_nat := tag { untag :> nat }.
Structure reified_of :=
  reify { nat_of : tagged_nat ; reified_nat_of :> @expr nat }.
```

tags to control the order of application `Definition S_tag := tag.`

```
Definition O_tag := S_tag.
```

N.B. Canonical structures follow Import, so they must be imported for reification to work.

```
Module Export Exports.
  Canonical Structure mul_tag n := O_tag n.
  Canonical Structure reify_O
    := reify (O_tag O) (@NatO nat).
  Canonical Structure reify_S x
    := reify (S_tag (S (nat_of x))) (@NatS nat x).
  Canonical Structure reify_mul x y
    := reify (mul_tag (nat_of x × nat_of y))
             (@NatMul nat x y).
End Exports.
```

We take advantage of not needing to lock Let_In to avoid a rewrite by passing false to the do_lock_letin argument of make_pre_Reify_rhs

```
Ltac pre_Reify_rhs _ := make_pre_Reify_rhs nat_of untag false false.
```

N.B. we must thunk the constants so as to not focus the goal

```
Ltac do_Reify_rhs _ := make_do_Reify_rhs ltac:(fun _ ⇒ denote)
                                         ltac:(fun _ ⇒ reified_nat_of)
                                         ltac:(fun x ⇒ x).
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := pre_Reify_rhs (); do_Reify_rhs (); post_Reify_rhs ().
```

E.8.2 Canonical-structure based reification (CanonicalStructuresFlatPHOAS.v)

This version reifies to Expr, and does not support let-binders.

```
Require Import Reify.CanonicalStructuresReifyCommon.
```

structure for packaging a nat expr and its reification

```
Structure tagged_nat := tag { untag :> nat }.
Structure reified_of :=
  reify { nat_of : tagged_nat ; reified_nat_of :> Expr }.
```

tags to control the order of application

```
Definition S_tag := tag.
Definition O_tag := S_tag.
```

N.B. Canonical structures follow Import, so they must be imported for reification to work.

```
Module Export Exports.
  Canonical Structure mul_tag n := O_tag n.
  Canonical Structure reify_O
    := reify (O_tag 0) (@NatO).
  Canonical Structure reify_S x
    := reify (S_tag (S (nat_of x))) (fun var ⇒ @NatS var (reified_nat_of x var)).
  Canonical Structure reify_mul x y
    := reify (mul_tag (nat_of x × nat_of y))
             (fun var ⇒ @NatMul var (reified_nat_of x var) (reified_nat_of y var)).
End Exports.
```

We take advantage of not needing to lock Let_In to avoid a rewrite by passing false to the do_lock_letin argument of make_pre_Reify_rhs

```
Ltac pre_Reify_rhs _ := make_pre_Reify_rhs nat_of untag false false.
```

N.B. we must thunk the constants so as to not focus the goal

```
Ltac do_Reify_rhs _ := make_do_Reify_rhs ltac:(fun _ ⇒ Denote)
                                         ltac:(fun _ ⇒ reified_nat_of)
                                         ltac:(fun x ⇒ x).
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := pre_Reify_rhs (); do_Reify_rhs (); post_Reify_rhs ().
```

E.8.3 Canonical-structure based reification (CanonicalStructuresHOAS.v)

This version reifies to @expr nat, and supports let-binders.

```
Require Import Reify.CanonicalStructuresReifyCommon.
```

structure for packaging a nat expr and its reification

```
Structure tagged_nat := tag { untag :> nat }.
Structure reified_of :=
  reify { nat_of : tagged_nat ; reified_nat_of :> @expr nat }.
```

tags to control the order of application

```
Definition var_tag := tag.
Definition S_tag := var_tag.
Definition O_tag := S_tag.
Definition let_in_tag := O_tag.
```

N.B. Canonical structures follow Import, so they must be imported for reification to work.

```
Module Export Exports.
  Canonical Structure mul_tag n := let_in_tag n.
  Canonical Structure reify_var n
    := reify (var_tag n) (@Var nat n).
  Canonical Structure reify_O
    := reify (O_tag O) (@NatO nat).
  Canonical Structure reify_S x
    := reify (S_tag (S (nat_of x))) (@NatS nat x).
  Canonical Structure reify_mul x y
    := reify (mul_tag (nat_of x × nat_of y))
             (@NatMul nat x y).
  Canonical Structure reify_let_in v f
    := reify (let_in_tag (nllet x := untag (nat_of v) in nat_of (f x)))
             (elet x := reified_nat_of v in reified_nat_of (f x)).
End Exports.

Ltac pre_Reify_rhs _ := make_pre_Reify_rhs nat_of untag true false.
```

N.B. we must thunk the constants so as to not focus the goal

```
Ltac do_Reify_rhs _ := make_do_Reify_rhs ltac:(fun _ ⇒ denote)
                                         ltac:(fun _ ⇒ reified_nat_of)
                                         ltac:(fun x ⇒ x).
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := pre_Reify_rhs (); do_Reify_rhs (); post_Reify_rhs ().
```

E.8.4 Canonical-structure based reification (CanonicalStructuresPHOAS.v)

This version reifies to Expr, and supports let-binders.

```
Require Import Coq.Lists.List.
Require Import Reify.CanonicalStructuresReifyCommon.

Local Notation context := (list nat).

Structure tagged_nat (ctx : context) := tag { untag :> nat }.

Structure reified_of (ctx : context) :=
  reify { nat_of : tagged_nat ctx ;
          reified_nat_of :> ∀ var, list var → (∀ T, T) → @expr var }.

Definition var_tl_tag := tag.
Definition var_hd_tag := var_tl_tag.
Definition S_tag := var_hd_tag.
Definition O_tag := S_tag.
Definition mul_tag := O_tag.
```

N.B. Canonical structures follow Import, so they must be imported for reification to work.

```
Module Export Exports.
  Canonical Structure letin_tag ctx n := mul_tag ctx n.

  Canonical Structure reify_O ctx
    := reify (O_tag ctx 0) (fun var _ _ ⇒ @NatO var).
  Canonical Structure reify_S ctx x
    := reify (@S_tag ctx (S (@nat_of ctx x)))
             (fun var vs phantom ⇒ @NatS var (x var vs phantom)).
  Canonical Structure reify_mul ctx x y
    := reify (@mul_tag ctx (@nat_of ctx x × @nat_of ctx y))
             (fun var vs phantom ⇒ @NatMul var (x var vs phantom) (y var vs phantom)).
  Canonical Structure reify_var_hd n ctx
    := reify (var_hd_tag (n :: ctx) n)
             (fun var vs phantom ⇒ @Var var (List.hd (phantom _) vs)).
  Canonical Structure reify_var_tl n ctx x
    := reify (var_tl_tag (n :: ctx) (@nat_of ctx x))
             (fun var vs phantom ⇒ reified_nat_of x (List.tl vs) phantom).
  Canonical Structure reify_letin ctx v f
    := reify (letin_tag
                ctx
                (nllet x := @nat_of ctx v in
                 @nat_of (x :: ctx) (f x)))
             (fun var vs phantom
              ⇒ elet x := reified_nat_of v vs phantom in
                reified_nat_of (f (phantom _)) (x :: vs) phantom)%expr.
End Exports.
```

```
Definition ReifiedNatOf (e : reified_of nil) : (∀ T, T) → Expr
  := fun phantom var ⇒ reified_nat_of e nil phantom.

Ltac pre_Reify_rhs _ := make_pre_Reify_rhs (@nat_of nil) (@untag nil) true false.
```

N.B. we must thunk the constants so as to not focus the goal

```
Ltac do_Reify_rhs _ :=
  make_do_Reify_rhs
    ltac:(fun _ ⇒ Denote) ltac:(fun _ ⇒ ReifiedNatOf)
    ltac:(fun e ⇒
            lazymatch e with
            | fun _ ⇒ ?e ⇒ e
            | _ ⇒ ReifyCommon.error_cant_elim_deps e
            end).
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := pre_Reify_rhs (); do_Reify_rhs (); post_Reify_rhs ().
```

E.9 Reification by Mtac2 (Mtac2.v)

```
Require Import Coq.Strings.String Coq.ZArith.ZArith.
Require Import Reify.ReifyCommon.
Require Import Mtac2.Mtac2.
Import M.notations.
```

Points of note:

- We use `=n>` to avoid unnecessary normalization / reduction
- We handle fresh binder names manually rather than invoking `M.fresh_binder_name`, which produces a string with length linear in the number of times it has been called so far, rather than logarithmic.

```
Module var_context.
  Inductive var_context {var : Type} :=
  | nil
  | cons (n : nat) (v : var) (xs : var_context).
End var_context.

Fixpoint string_of_pos_bin' (p : positive) (rest : string) : string
  := match p with
     | xI x ⇒ string_of_pos_bin' x (String "1" rest)
     | xO x ⇒ string_of_pos_bin' x (String "0" rest)
     | xH ⇒ String "0" (String "b" (String "1" rest))
     end.
Definition string_of_pos_bin (p : positive) : string
  := string_of_pos_bin' p EmptyString.
```

We'd like to just use `M.fresh_binder_name f`, but this incurs significant overhead (about 2X slower at 27 invocations) because it has strings linear in the number of repeated invocations, rather than logarithmic, so instead we handle binder names manually.

`norm_string` is useful for printing / debugging, but incurs a bit of overhead.

```
Definition norm_string (v : string) : M string :=
  (mfix1 norm_string (s : string) : M string :=
     (mmatch s with
     | EmptyString ⇒ M.ret EmptyString
     | [? a b] (String a b) ⇒ b' ← norm_string b; M.ret (String a b')
      end)) v.
Definition name_binder {A B} (f : A → B) (var_idx : positive) : M string
  := M.ret (String "x" (string_of_pos_bin var_idx)).
Definition find_in_ctx {var : Type} (term : nat)
           (ctx : @var_context.var_context var)
  : M (option var)
  := (mfix1 find_in_ctx (ctx : @var_context.var_context var) : M (option var) :=
        (mmatch ctx with
        | [? v xs] (var_context.cons term v xs)
```

```
          =n> M.ret (Some v)
        | [? x v xs] (var_context.cons x v xs)
          =n> find_in_ctx xs
        | _ ⇒ M.ret None
         end)) ctx.
Definition reify_helper {var : Type} (term : nat)
           (ctx : @var_context.var_context var)
  : M (@expr var)
  := ((mfix3 reify_helper (term : nat) (var_idx : positive)
             (ctx : @var_context.var_context var)
      : M (@expr var) :=
        lvar ← find_in_ctx term ctx;
          match lvar with
          | Some v ⇒ M.ret (@Var var v)
          | None
            ⇒
            (mmatch term with
            | O
              =n> M.ret (@NatO var)
            | [? x] (S x)
              =n> (rx ← reify_helper x var_idx ctx;
                     M.ret (@NatS var rx))
            | [? x y] (x × y)
              =n> (rx ← reify_helper x var_idx ctx;
                     ry ← reify_helper y var_idx ctx;
                     M.ret (@NatMul var rx ry))
            | [? v f] (@Let_In nat nat v f)
              =n> (rv ← reify_helper v var_idx ctx;
                   x ← name_binder f var_idx;
                   let vx := String.append "var_" x in
                   rf ← (M.nu
                           x mNone
                           (fun x : nat
                            ⇒ M.nu
                                 vx mNone
                                 (fun vx : var
                                  ⇒ let fx := reduce (RedWhd [rl:RedBeta]) (f x) in
                                     rf ← (reify_helper
                                             fx
                                             (Pos.succ var_idx)
                                             (var_context.cons x vx ctx));
                                     M.abs_fun vx rf)));
                   M.ret (@LetIn var rv rf))
             end)
          end) term 1%positive ctx).
```

```
Definition reify (var : Type) (term : nat) : M (@expr var)
  := reify_helper term var_context.nil.
Definition Reify (term : nat) : M Expr
  := \nu var:Type, r ← reify var term; M.abs_fun var r.

Ltac Reify' x := constr:(ltac:(mrun (@Reify x))).
Ltac Reify x := Reify' x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of ltac:(Reify) ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of ltac:(Reify) ().
```

E.10 Ltac2

E.10.1 Common utility definitions for Ltac2 (Ltac2Common.v)

```
Require Ltac2.Ltac2.
Import Ltac2.Init.
Import Ltac2.Notations.

Module List.
  Ltac2 rec map f ls :=
    match ls with
    | [] ⇒ []
    | l :: ls ⇒ f l :: map f ls
    end.
End List.
Module Ident.
  Ltac2 rec find_error id xs :=
    match xs with
    | [] ⇒ None
    | x :: xs
      ⇒ let ((id', val)) := x in
        match Ident.equal id id' with
        | true ⇒ Some val
        | false ⇒ find_error id xs
        end
    end.
  Ltac2 find id xs :=
    match find_error id xs with
    | None ⇒ Control.zero Not_found
    | Some val ⇒ val
    end.
End Ident.
Module Array.
  Ltac2 rec to_list_aux (ls : 'a array) (start : int) :=
    match Int.equal (Int.compare start (Array.length ls)) -1 with
    | true ⇒ Array.get ls start :: to_list_aux ls (Int.mul start 1)
    | false ⇒ []
    end.
  Ltac2 to_list (ls : 'a array) := to_list_aux ls 0.
End Array.
Module Constr.
  Ltac2 rec strip_casts term :=
    match Constr.Unsafe.kind term with
    | Constr.Unsafe.Cast term' _ _ ⇒ strip_casts term'
    | _ ⇒ term
    end.
  Module Unsafe.
```

```
    Ltac2 beta1 (c : constr) :=
      match Constr.Unsafe.kind c with
      | Constr.Unsafe.App f args
        ⇒ match Constr.Unsafe.kind f with
          | Constr.Unsafe.Lambda id ty f
            ⇒ Constr.Unsafe.substnl (Array.to_list args) 0 f
          | _ ⇒ c
          end
      | _ ⇒ c
      end.
    Ltac2 zeta1 (c : constr) :=
      match Constr.Unsafe.kind c with
      | Constr.Unsafe.LetIn id v ty f
        ⇒ Constr.Unsafe.substnl [v] 0 f
      | _ ⇒ c
      end.
  End Unsafe.
End Constr.
Module Ltac1.
  Class Ltac1Result {T} (v : T) := {}.
  Class Ltac1Results {T} (v : list T) := {}.
  Class Ltac2Result {T} (v : T) := {}.
  Ltac save_ltac1_result v :=
    match goal with
    | _ ⇒ assert (Ltac1Result v) by constructor
    end.
  Ltac clear_ltac1_results _ :=
    match goal with
    | _ ⇒ repeat match goal with
                 | [ H : Ltac1Result _ ⊢ _ ] ⇒ clear H
                 end
    end.
  Ltac2 get_ltac1_result () :=
    (lazy_match! goal with
    | [ id : Ltac1Result ?v ⊢ _ ]
      ⇒ Std.clear [id]; v
     end).

  Ltac save_ltac1_results v :=
    match goal with
    | _ ⇒ assert (Ltac1Result v) by constructor
    end.
  Ltac2 save_ltac2_result v :=
    Std.cut '(Ltac2Result $v);
      Control.dispatch
        [(fun ()
```

```
          ⇒ Std.intros false [Std.IntroNaming (Std.IntroFresh @res)])
        ;
        (fun () ⇒ Notations.constructor)].
  Ltac get_ltac2_result _ :=
    lazymatch goal with
    | [ res : Ltac2Result ?v ⊢ _ ]
      ⇒ let __ := match goal with
                  | _ ⇒ clear res
                  end in
        v
    end.
  Ltac2 from_ltac1 (save_args : constr) (tac : unit → unit) :=
    let beta_flag :=
        {
          Std.rBeta := true; Std.rMatch := false;
          Std.rFix := false; Std.rCofix := false;
          Std.rZeta := false; Std.rDelta := false; Std.rConst := [];
        } in
    let c := '(ltac2:(save_ltac2_result save_args;
                      tac ();
                      let v := get_ltac1_result () in
                      Control.refine (fun () ⇒ v))) in
    Constr.Unsafe.zeta1 (Constr.Unsafe.zeta1 (Std.eval_cbv beta_flag c)).
End Ltac1.
```

E.10.2 Reification by Ltac2, copying Ltac1 (Ltac2.v)

This file contains the Ltac2 version of Ltac1 reification, from LtacTacInTermExplicitCtx.v.

```
Require Import Reify.ReifyCommon.
Require Import Reify.Ltac2Common.
Import Ltac2.Init.
Import Ltac2.Notations.

Ltac2 rec reify_helper
      (var : constr)
      (term : constr)
      (ctx : (ident × ident) list)
  :=
    let reify_rec term := reify_helper var term ctx in
    Control.plus
      (fun ()
       ⇒ match Constr.Unsafe.kind (Constr.strip_casts term) with
         | Constr.Unsafe.Var x
           ⇒ let v := Ident.find x ctx in
             let v := Constr.Unsafe.make (Constr.Unsafe.Var v) in
             constr:(@Var $var $v)
         | _ ⇒ Control.zero Not_found
         end)
      (fun _
       ⇒ (lazy_match! term with
         | O ⇒ constr:(@NatO $var)
         | S ?x
           ⇒ let rx := reify_rec x in
             constr:(@NatS $var $rx)
         | ?x × ?y
           ⇒ let rx := reify_rec x in
             let ry := reify_rec y in
             constr:(@NatMul $var $rx $ry)
         | (dlet x := ?v in @?f x)
           ⇒ let rv := reify_rec v in
```

We assume the invariant that all bound variables show up as Rel nodes rather than Var nodes

```
             match Constr.Unsafe.kind f with
             | Constr.Unsafe.Lambda id t c
               ⇒ let c_set := Fresh.Free.of_ids
                                (List.map (fun (id, _, _) ⇒ id)
                                          (Control.hyps ())) in
                 let c_set := Fresh.Free.union
                                c_set
```

```
                    (Fresh.Free.of_constr c) in
      let x_base := match id with
                    | Some id ⇒ id
                    | None ⇒ @x
                    end in
      let x := Fresh.fresh c_set x_base in
      let c_set := Fresh.Free.union
                     c_set
                     (Fresh.Free.of_ids [x]) in
      let not_x := Fresh.fresh c_set x_base in
      let ctx := (x, not_x) :: ctx in
      let c := Constr.Unsafe.substnl
                 [Constr.Unsafe.make (Constr.Unsafe.Var x)]
                 0
                 c in
      let ret :=
          Constr.in_context
            x t
            (fun ()
             ⇒ let rf :=
                    Constr.in_context
                      not_x var
                      (fun ()
                       ⇒ let rf := reify_helper var c ctx in
                         Control.refine (fun () ⇒ rf)) in
               Control.refine (fun () ⇒ rf)) in
      (lazy_match! ret with
      | fun _ ⇒ ?rf
        ⇒ constr:(@LetIn $var $rv $rf)
      | ?f
        ⇒ let msg :=
              Message.concat
                (Message.of_string
                   "Failed to eliminate functional dependencies in ")
                (Message.of_constr f) in
          Message.print msg;
            Control.zero
              (Tactic_failure (Some msg))
       end)
| _ ⇒ let msg :=
           Message.concat
             (Message.of_string "Bad let-in function: ")
             (Message.of_constr f) in
      Message.print msg;
        Control.zero (Tactic_failure (Some msg))
```

```
            end
          | _
            ⇒ let msg := Message.concat
                           (Message.of_string "Unrecognized term: ")
                           (Message.of_constr term) in
              Message.print msg;
                Control.zero (Tactic_failure (Some msg))
           end)).

Ltac2 reify (var : constr) (term : constr) := reify_helper var term [].

Ltac reify var term :=
  let __ := Ltac1.save_ltac1_result (var, term) in
  let ret :=
      constr:(ltac2:(let args := Ltac1.get_ltac1_result () in
                     (lazy_match! args with
                     | (?var, ?term)
                       ⇒ let rv := reify var term in
                          Control.refine (fun () ⇒ rv)
                     | _ ⇒ Control.throw Not_found
                      end))) in
  let __ := Ltac1.clear_ltac1_results () in
  ret.

Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs := ReifyCommon.post_Reify_rhs.
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.10.3 Reification by Ltac2, using unsafe low-level primitives (`Ltac2LowLevel.v`)

```
Require Import Reify.ReifyCommon.
Require Import Reify.Ltac2Common.
Import Ltac2.Init.
Import Ltac2.Notations.

Ltac2 if_doing_trans (tac : unit → 'a) (default : 'a) :=
  let do_trans := '(do_transitivity) in
  (lazy_match! do_trans with
  | true ⇒ tac ()
  | false ⇒ default
  end).
```

This function is parameterized over the constants which we are reifying (`gO`, `gS`, `gNatMul`, `gLetIn`) and over Ltac2 functions that build applications of the reified versions of these functions to reified arguments. `Ltac2 rec unsafe_reify_helper`

```
      (mkVar : constr → 'a)
      (mkO : 'a)
      (mkS : 'a → 'a)
      (mkNatMul : 'a → 'a → 'a)
      (mkLetIn : 'a → ident option → constr → 'a → 'a)
      (gO : constr)
      (gS : constr)
      (gNatMul : constr)
      (gLetIn : constr)
      (unrecognized : constr → 'a)
      (term : constr)
  :=
    let reify_rec term :=
        unsafe_reify_helper
          mkVar mkO mkS mkNatMul mkLetIn gO gS gNatMul gLetIn unrecognized term in
    let kterm := Constr.Unsafe.kind term in
    match Constr.equal term gO with
    | true
      ⇒ mkO
    | false
      ⇒
      match kterm with
      | Constr.Unsafe.Rel _ ⇒ mkVar term
      | Constr.Unsafe.Var _ ⇒ mkVar term
      | Constr.Unsafe.Cast term _ _ ⇒ reify_rec term
      | Constr.Unsafe.App f args
        ⇒
        match Constr.equal f gS with
        | true
```

```
          ⇒ let x := Array.get args 0 in
            let rx := reify_rec x in
            mkS rx
        | false
          ⇒
          match Constr.equal f gNatMul with
          | true
            ⇒ let x := Array.get args 0 in
              let y := Array.get args 1 in
              let rx := reify_rec x in
              let ry := reify_rec y in
              mkNatMul rx ry
          | false
            ⇒
            match Constr.equal f gLetIn with
            | true
              ⇒ let x := Array.get args 2  in
                let f := Array.get args 3 in
                match Constr.Unsafe.kind f with
                | Constr.Unsafe.Lambda idx ty body
                  ⇒ let rx := reify_rec x in
                    let rf := reify_rec body in
                    mkLetIn rx idx ty rf
                | _ ⇒ unrecognized term
                end
            | false
              ⇒ unrecognized term
            end
          end
      | _
        ⇒ unrecognized term
      end
  end.

Ltac2 unsafe_reify (var : constr) (term : constr) :=
  let cVar := '@Var in
  let cO := '@NatO in
  let cS := '@NatS in
  let cNatMul := '@NatMul in
  let cLetIn := '@LetIn in
  let gO := 'O in
  let gS := 'S in
  let gNatMul := '@Nat.mul in
  let gLetIn := '@Let_In in
  let mkOVarArgs :=
```

```
      let args := Array.make 1 var in
      args in
  let mk1VarArgs (x : constr) :=
      let args := Array.make 2 var in
      let () := Array.set args 1 x in
      args in
  let mk2VarArgs (x : constr) (y : constr) :=
      let args := Array.make 3 var in
      let () := Array.set args 1 x in
      let () := Array.set args 2 y in
      args in
  let mkApp0 (f : constr) :=
      Constr.Unsafe.make (Constr.Unsafe.App f mk0VarArgs) in
  let mkApp1 (f : constr) (x : constr) :=
      Constr.Unsafe.make (Constr.Unsafe.App f (mk1VarArgs x)) in
  let mkApp2 (f : constr) (x : constr) (y : constr) :=
      Constr.Unsafe.make (Constr.Unsafe.App f (mk2VarArgs x y)) in
  let mkVar (v : constr) := mkApp1 cVar v in
  let mkO := mkApp0 cO in
  let mkS (v : constr) := mkApp1 cS v in
  let mkNatMul (x : constr) (y : constr) := mkApp2 cNatMul x y in
  let mkcLetIn (x : constr) (y : constr) := mkApp2 cLetIn x y in
  let mkLetIn (x : constr) (idx : ident option) (ty : constr) (fbody : constr)
      := mkcLetIn x (Constr.Unsafe.make (Constr.Unsafe.Lambda idx var fbody)) in
  let ret := unsafe_reify_helper
               mkVar mkO mkS mkNatMul mkLetIn gO gS gNatMul gLetIn
               (fun term ⇒ term)
               term in
  ret.
Ltac2 check_result (ret : constr) :=
  match Constr.Unsafe.check ret with
  | Val rterm ⇒ rterm
  | Err exn ⇒ Control.zero exn
  end.
Ltac2 reify (var : constr) (term : constr) :=
  check_result (unsafe_reify var term).

Ltac2 unsafe_Reify (term : constr) :=
  let fresh_set := Fresh.Free.of_constr term in
  let idvar := Fresh.fresh fresh_set @var in
  let var := Constr.Unsafe.make (Constr.Unsafe.Var idvar) in
  let rterm := unsafe_reify var term in
  let rterm := Constr.Unsafe.closenl [idvar] 1 rterm in
  Constr.Unsafe.make (Constr.Unsafe.Lambda (Some idvar) 'Type rterm).

Ltac2 do_Reify (term : constr) :=
  check_result (unsafe_Reify term).
```

```
Ltac2 unsafe_mkApp1 (f : constr) (x : constr) :=
  let args := Array.make 1 x in
  Constr.Unsafe.make (Constr.Unsafe.App f args).
Ltac2 mkApp1 (f : constr) (x : constr) :=
  check_result (unsafe_mkApp1 f x).

Ltac2 all_flags :=
  {
    Std.rBeta := true; Std.rMatch := true; Std.rFix := true; Std.rCofix := true;
    Std.rZeta := true; Std.rDelta := true; Std.rConst := [];
  }.
Ltac2 betaiota_flags :=
  {
    Std.rBeta := true; Std.rMatch := true; Std.rFix := true; Std.rCofix := true;
    Std.rZeta := false; Std.rDelta := false; Std.rConst := [];
  }.
Ltac2 in_goal :=
  { Std.on_hyps := None; Std.on_concl := Std.AllOccurrences }.

Ltac2 do_Reify_rhs_fast () :=
  let g := Control.goal () in
  match Constr.Unsafe.kind g with
  | Constr.Unsafe.App f args
    ⇒ let v := Array.get args 2 in
      let rv := Control.time (Some "actual reif")
                             (fun _ ⇒ unsafe_Reify v) in
      let rv := Control.time (Some "eval lazy")
                             (fun _ ⇒ Std.eval_lazy all_flags rv) in
      Control.time (Some "lazy beta iota")
                   (fun _ ⇒ Std.lazy betaiota_flags in_goal);
        if_doing_trans
          (fun _ ⇒ Control.time
                      (Some "transitivity (Denote rv)")
                      (fun _ ⇒ Std.transitivity (unsafe_mkApp1 'Denote rv))) ()
  | _
    ⇒ Control.zero
        (Tactic_failure
           (Some (Message.concat
                    (Message.of_string
                       "Invalid goal in Ltac2Unsafe.do_Reify_rhs_fast: ")
                    (Message.of_constr g))))
  end.

Ltac2 do_Reify_rhs () :=
  (lazy_match! goal with
  | [ ⊢ _ = ?v ]
    ⇒ let rv := do_Reify v in
      let rv := Std.eval_lazy all_flags rv in
```

```
      if_doing_trans (fun _ ⇒ Std.transitivity (mkApp1 'Denote rv)) ()
  | [ ⊢ ?g ] ⇒ Control.zero
                 (Tactic_failure
                    (Some (Message.concat
                             (Message.of_string
                                "Invalid goal in Ltac2Unsafe.do_Reify_rhs: ")
                             (Message.of_constr g))))
   end).

Ltac reify var term :=
  let __ := Ltac1.save_ltac1_result (var, term) in
  let ret :=
      constr:(ltac2:(let args := Ltac1.get_ltac1_result () in
                     (lazy_match! args with
                     | (?var, ?term)
                       ⇒ let rv := reify var term in
                          Control.refine (fun () ⇒ rv)
                     | _ ⇒ Control.throw Not_found
                      end))) in
  let __ := Ltac1.clear_ltac1_results () in
  ret.

Ltac Reify x := Reify_of reify x.
Ltac do_Reify_rhs _ := ltac2:(do_Reify_rhs_fast ()).
Ltac post_Reify_rhs := ReifyCommon.post_Reify_rhs.
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.11 OCaml

E.11.1 OCaml reification (reify_plugin.ml4)

```
(*i camlp4deps: "parsing/grammar.cma" i*)
(*i camlp4use: "pa_extend.cmp" i*)

open Names

let rec unsafe_reify_helper
        (mkVar : Constr.t -> 'a)
        (mkO : 'a)
        (mkS : 'a -> 'a)
        (mkOp : 'a -> 'a -> 'a)
        (mkLetIn : 'a -> Name.t -> Constr.t -> 'a -> 'a)
        (gO : Constr.t)
        (gS : Constr.t)
        (gOp : Constr.t)
        (gLetIn : Constr.t)
        (unrecognized : Constr.t -> 'a)
        (term : Constr.t)
    =
      let reify_rec term =
        unsafe_reify_helper
          mkVar mkO mkS mkOp mkLetIn gO gS gOp gLetIn unrecognized term in
      let kterm = Constr.kind term in
      if Constr.equal term gO
      then mkO
      else begin match kterm with
      | Term.Rel _ -> mkVar term
      | Term.Var _ -> mkVar term
      | Term.Cast (term, _, _) -> reify_rec term
      | Term.App (f, args)
        ->
        if Constr.equal f gS
        then let x = Array.get args 0 in
             let rx = reify_rec x in
             mkS rx
        else if Constr.equal f gOp
        then let x = Array.get args 0 in
             let y = Array.get args 1 in
             let rx = reify_rec x in
             let ry = reify_rec y in
             mkOp rx ry
        else if Constr.equal f gLetIn
        then let x = Array.get args 2 (* assume the first two args are type params *) in
```

```
              let f = Array.get args 3 in
              begin match Constr.kind f with
              | Term.Lambda (idx, ty, body)
                -> let rx = reify_rec x in
                   let rf = reify_rec body in
                   mkLetIn rx idx ty rf
              | _ -> unrecognized term
              end
          else unrecognized term
      | _
        -> unrecognized term
      end

let unsafe_reify
        (cVar : Constr.t)
        (c0 : Constr.t)
        (cS : Constr.t)
        (cOp : Constr.t)
        (cLetIn : Constr.t)
        (g0 : Constr.t)
        (gS : Constr.t)
        (gOp : Constr.t)
        (gLetIn : Constr.t)
        (var : Constr.t)
        (term : Constr.t) : Constr.t =
    let mkApp0 (f : Constr.t) =
        Constr.mkApp (f, [| var |]) in
    let mkApp1 (f : Constr.t) (x : Constr.t) =
        Constr.mkApp (f, [| var ; x |]) in
    let mkApp2 (f : Constr.t) (x : Constr.t) (y : Constr.t) =
        Constr.mkApp (f, [| var ; x ; y |]) in
    let mkVar (v : Constr.t) = mkApp1 cVar v in
    let mk0 = mkApp0 c0 in
    let mkS (v : Constr.t) = mkApp1 cS v in
    let mkOp (x : Constr.t) (y : Constr.t) = mkApp2 cOp x y in
    let mkcLetIn (x : Constr.t) (y : Constr.t) = mkApp2 cLetIn x y in
    let mkLetIn (x : Constr.t) (idx : Name.t) (ty : Constr.t) (fbody : Constr.t)
        = mkcLetIn x (Constr.mkLambda (idx, var, fbody)) in
    let ret = unsafe_reify_helper
              mkVar mk0 mkS mkOp mkLetIn g0 gS gOp gLetIn
              (fun term -> term)
              term in
    ret

let unsafe_Reify
```

```
        (cVar : Constr.t)
        (cO : Constr.t)
        (cS : Constr.t)
        (cOp : Constr.t)
        (cLetIn : Constr.t)
        (gO : Constr.t)
        (gS : Constr.t)
        (gOp : Constr.t)
        (gLetIn : Constr.t)
        (idvar : Id.t)
        (varty : Constr.t)
        (term : Constr.t) : Constr.t =
    let fresh_set = let rec fold accu c = match Constr.kind c with
      | Constr.Var id -> Id.Set.add id accu
      | _ -> Constr.fold fold accu c
      in
      fold Id.Set.empty term in
    let idvar = Namegen.next_ident_away_from
                  idvar
                  (fun id -> Id.Set.mem id fresh_set) in
    let var = Constr.mkVar idvar in
    let rterm = unsafe_reify cVar cO cS cOp cLetIn gO gS gOp gLetIn var term in
    let rterm = Vars.substn_vars 1 [idvar] rterm in
    Constr.mkLambda (Name.Name idvar, varty, rterm)

DECLARE PLUGIN "reify"

open Ltac_plugin
open Stdarg
open Tacarg
open Names

(** Stolen from plugins/setoid_ring/newring.ml *)
open Tacexpr
open Misctypes
open Tacinterp
(* Calling a locally bound tactic *)
let ltac_lcall tac args =
  TacArg(Loc.tag @@ TacCall (Loc.tag (ArgVar(Loc.tag @@ Id.of_string tac),args)))

let ltac_apply (f : Value.t) (args: Tacinterp.Value.t list) =
  let fold arg (i, vars, lfun) =
    let id = Id.of_string ("x" ^ string_of_int i) in
    let x = Reference (ArgVar (Loc.tag id)) in
    (succ i, x :: vars, Id.Map.add id arg lfun)
```

```
  in
  let (_, args, lfun) = List.fold_right fold args (0, [], Id.Map.empty) in
  let lfun = Id.Map.add (Id.of_string "F") f lfun in
  let ist = { (Tacinterp.default_ist ()) with Tacinterp.lfun = lfun; } in
  Tacinterp.eval_tactic_ist ist (ltac_lcall "F" args)

let to_ltac_val c = Tacinterp.Value.of_constr c

open Pp

TACTIC EXTEND quote_term_cps
    | [ "quote_term_cps" "[" ident(idvar) "," constr(varty) "]"
          constr(cVar) constr(c0) constr(cS) constr(cOp) constr(cLetIn)
          constr(g0) constr(gS) constr(gOp) constr(gLetIn)
          constr(term) tactic(tac) ] ->
      [ (** quote the given term, pass the result to t **)
  Proofview.Goal.enter begin fun gl ->
          let _ (*env*) = Proofview.Goal.env gl in
          let c = unsafe_Reify
                    (EConstr.Unsafe.to_constr cVar)
                    (EConstr.Unsafe.to_constr c0)
                    (EConstr.Unsafe.to_constr cS)
                    (EConstr.Unsafe.to_constr cOp)
                    (EConstr.Unsafe.to_constr cLetIn)
                    (EConstr.Unsafe.to_constr g0)
                    (EConstr.Unsafe.to_constr gS)
                    (EConstr.Unsafe.to_constr gOp)
                    (EConstr.Unsafe.to_constr gLetIn)
                    idvar
                    (EConstr.Unsafe.to_constr varty)
                    (EConstr.Unsafe.to_constr term) in
          ltac_apply tac (List.map to_ltac_val [EConstr.of_constr c])
  end ]
END;;
```

E.11.2 Reification in OCaml (OCamlReify.v)

```
Declare ML Module "reify".
```

E.11.3 Reification in OCaml (OCaml.v)

```
Require Import Reify.ReifyCommon.
Require Import Reify.OCamlReify.
```

See OCamlReify.v and reify_plugin.ml4 for the implementation code.

```
Ltac Reify_cps term tac :=
  quote_term_cps
    [var, Type] (@Var) (@NatO) (@NatS) (@NatMul) (@LetIn) O S Nat.mul (@Let_In)
    term tac.

Ltac reify_cps var term tac :=
  Reify_cps term ltac:(fun rt ⇒ tac (rt var)).

Ltac do_Reify_rhs _ := do_Reify_rhs_of_cps Reify_cps ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of_cps Reify_cps ().
```

E.12 Reification by template-coq (TemplateCoq.v)

```
Require Coq.Strings.String.
Require Import Reify.ReifyCommon.
Require Import Template.Ast.
Require Import Template.Template.

Module Compile.
  Import Coq.Strings.String.
  Scheme Equality for string.

  Section with_var.
    Context {var : Type}.
    Axiom bad : var.
    Fixpoint compile (e : term) (ctx : list var) : @expr var
      := match e with
         | tRel idx ⇒ Var (List.nth_default bad ctx idx)
         | tCast e _ _
           ⇒ compile e ctx
         | tConstruct (mkInd Bp 0) 0 _
           ⇒ @NatO var
         | tApp f4 (_ :: _ :: x :: tLambda _ _ f :: nil)
           ⇒ @LetIn var (compile x ctx)
                    (fun x' ⇒ compile f (x' :: ctx))
         | tApp f2 (x :: y :: nil)
           ⇒ @NatMul var (compile x ctx) (compile y ctx)
         | tApp f1 (x :: nil)
           ⇒ @NatS var (compile x ctx)
         | _
           ⇒ Var bad
         end%list.
  End with_var.
  Definition Compile (e : term) : Expr := fun var ⇒ @compile var e nil.
End Compile.

Ltac reify_cps var term tac :=
  quote_term term (fun v ⇒ tac (@Compile.compile var v)).

Ltac Reify_cps term tac :=
  quote_term term (fun v ⇒ tac (Compile.Compile v)).

Ltac do_Reify_rhs _ := do_Reify_rhs_of_cps Reify_cps ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of_cps Reify_cps ().
```

E.13 Reification by the quote plugin (QuoteFlat.v)

```
Require Import Coq.quote.Quote.
Require Import Reify.ReifyCommon.

Inductive qexpr : Set :=
| qNatO : qexpr
| qNatS : qexpr → qexpr
| qNatMul (x y : qexpr)
| qConst (k : nat).

Module Export Exports.
  Fixpoint qdenote (e : qexpr) : nat
    := match e with
       | qNatO ⇒ O
       | qNatS x ⇒ S (qdenote x)
       | qNatMul x y ⇒ Nat.mul (qdenote x) (qdenote y)
       | qConst k ⇒ k
       end.
End Exports.

Fixpoint compile_nat {var} (n : nat) : @expr var
  := match n with
     | O ⇒ NatO
     | S x ⇒ NatS (compile_nat x)
     end.

Fixpoint compile {var} (e : qexpr) : @expr var
  := match e with
     | qNatO ⇒ NatO
     | qNatS x ⇒ NatS (compile x)
     | qNatMul x y ⇒ NatMul (compile x) (compile y)
     | qConst k ⇒ compile_nat k
     end.

Definition Compile (e : qexpr) : Expr := fun var ⇒ compile e.

Ltac reify_cps var term tac :=
  quote qdenote [S O] in term using
    (fun v ⇒ lazymatch v with qdenote ?v ⇒ tac (@compile var v) end).

Ltac Reify_cps term tac :=
  quote qdenote [S O] in term using
    (fun v ⇒ lazymatch v with qdenote ?v ⇒ tac (@Compile v) end).

Ltac do_Reify_rhs _ := do_Reify_rhs_of_cps Reify_cps ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of_cps Reify_cps ().
```

E.14 Reification by parametricity (Parametricity.v)

```
Require Import Reify.ReifyCommon.

Ltac Reify x :=
  let rx := lazymatch (eval pattern nat, O, S, Nat.mul, (@Let_In nat nat) in x) with
            | ?rx _ _ _ _ _ ⇒ rx end in
  let rx := lazymatch rx with fun N : Set ⇒ ?rx ⇒ constr:(fun N : Type ⇒ rx) end in
  let __ := type of rx in
  constr:(fun var : Type ⇒ rx (@expr var) (@NatO var) (@NatS var) (@NatMul var)
                              (fun x' f' ⇒ @LetIn var x' (fun v ⇒ f' (@Var var v)))).

Ltac reify var x :=
  let rx := Reify x in
  constr:(rx var).

Ltac do_Reify_rhs _ := do_Reify_rhs_of Reify ().
Ltac post_Reify_rhs _ := ReifyCommon.post_Reify_rhs ().
Ltac Reify_rhs _ := Reify_rhs_of Reify ().
```

E.15 Reification by parametricity, with a routine for handling constants recursively (ParametricityWithConst.v)

```
Require Import Reify.ReifyCommon.
```

This file contains the extra code to handle constants recursively. We advise against using this code, and provide it as a proof of concept only.

expects:

- var - the PHOAS var type
- find_const term found_tac not_found_tac, a tactical which looks for constants in term, invokes found_tac with the constant if it finds one, and invokes not_found_tac () if it finds none.
- plug_const var term const, a tactic which takes a term and a constant, and plugs in the reified version of const

```
Ltac reify_with_consts var find_const plug_const term :=
  find_const
    term
    ltac:(fun c
          ⇒ let rx := lazymatch (eval pattern c in term) with
                      | ?term _ ⇒ term
                      end in
            let rx := reify_with_consts find_const plug_const term in
            plug_const var rx c)
    ltac:(fun _
          ⇒ let rx :=
                lazymatch
                  (eval pattern nat, Nat.mul, (@Let_In nat (fun _ ⇒ nat)), O, S
                    in term)
                with
                | ?rx _ _ _ _ _ ⇒ rx
                end in
            let rx := lazymatch rx with
                      | fun N : Set ⇒ ?rx ⇒ constr:(fun N : Type ⇒ rx)
                      end in
            let __ := type of rx in
            constr:(rx (@expr var) (@NatMul var)
                       (fun x' f'
                        ⇒ @LetIn var x'
                                   (fun v ⇒ f' (@Var var v)))
                       (@NatO var) (@NatS var))).
Ltac Reify_with_consts find_const plug_const term :=
  constr:(fun var : Type
          ⇒ ltac:(let rx := reify_with_consts var find_const plug_const term in
                  exact rx)).
```

E.16 Utility functions for PHOAS (PHOASUtil.v)

```
Require Import Reify.Common.
Require Import Reify.PHOAS.

Module PHOAS.
  Export Reify.PHOAS.

  Fixpoint beq_helper (e1 e2 : @expr nat) (base : nat) : bool
    := match e1, e2 with
       | LetIn v1 f1, LetIn v2 f2
         ⇒ if beq_helper v1 v2 base
           then beq_helper (f1 base) (f2 base) (S base)
           else false
       | Var v1, Var v2 ⇒ Nat.eqb v1 v2
       | NatO, NatO ⇒ true
       | NatS x, NatS y ⇒ beq_helper x y base
       | NatMul x1 y1, NatMul x2 y2
         ⇒ andb (beq_helper x1 x2 base) (beq_helper y1 y2 base)
       | LetIn _ _, _
       | Var _, _
       | NatO, _
       | NatS _, _
       | NatMul _ _, _
         ⇒ false
       end.
  Definition beq (e1 e2 : @expr nat) : bool := beq_helper e1 e2 O.
  Definition Beq (e1 e2 : Expr) : bool := beq (e1 _) (e2 _).
End PHOAS.
```

E.17 Various utilities for benchmarking (BenchmarkUtil.v)

```
Require Import Coq.ZArith.ZArith. Require Import Reify.Common.
Require Reify.PHOAS.
Require Import Reify.PHOASUtil.
```

E.17.1 Definition of the terms with which we build our benchmarking suite

```
Inductive count := none | one_more (how_many : count).
Fixpoint count_of_nat (v : nat) : count
  := match v with
     | O ⇒ none
     | S x ⇒ one_more (count_of_nat x)
     end.
Fixpoint nat_of_count (v : count) : nat
  := match v with
     | none ⇒ O
     | one_more x ⇒ S (nat_of_count x)
     end.
Fixpoint pos_of_succ_count (v : count) : positive
  := match v with
     | none ⇒ 1%positive
     | one_more x ⇒ Pos.succ (pos_of_succ_count x)
     end.
Definition Z_of_count (v : count) : Z
  := match v with
     | none ⇒ 0%Z
     | one_more x ⇒ Z.pos (pos_of_succ_count x)
     end.
Fixpoint big (x:nat) (n:count)
  : nat
  := match n with
     | none ⇒ x
     | one_more n'
       ⇒ dlet x' := x × x in
          big x' n'
     end.
Definition big_flat_op {T} (op : T → T → T) (a : T) (sz : count) : T
  := Eval cbv [Z_of_count pos_of_succ_count Pos.iter_op Pos.succ] in
      match Z_of_count sz with
      | Z0 ⇒ a
      | Zpos p ⇒ Pos.iter_op op p a
      | Zneg p ⇒ a
      end.
```

```
Definition big_flat (a : nat) (sz : count) : nat
  := big_flat_op Nat.mul a sz.

Ltac check_sane ref_PHOAS :=
  lazymatch goal with
  | [ ⊢ _ = PHOAS.Denote ?e ]
    ⇒ let val := (eval vm_compute in (PHOAS.Beq e ref_PHOAS)) in
      lazymatch val with
      | true ⇒ idtac
      | false ⇒ idtac "Error: Got" e "Expected:" ref_PHOAS; unify e ref_PHOAS
      end
  | [ ⊢ _ = PHOAS.denote ?e ]
    ⇒ let ref_HOAS := (eval lazy in (ref_PHOAS nat)) in
      let val := (eval vm_compute in (PHOAS.beq e ref_HOAS)) in
      lazymatch val with
      | true ⇒ idtac
      | false ⇒ idtac "Error: Got" e "Expected:" ref_HOAS; unify e ref_HOAS
      end
  | [ ⊢ _ = ?Denote ?e ]
    ⇒ fail 0 "Unrecognized denotation function" Denote
  end.
```

F Self-Contained Reification Example

F.1 Self-contained example of reification by parametricity on flat expressions (ExampleSelfContainedParametricity.v)

```
Require Import Coq.Bool.Bool.
Require Import Coq.Setoids.Setoid.

Inductive expr :=
| NatO : expr
| NatS : expr → expr
| NatMul : expr → expr → expr.
Fixpoint denote (t : expr) : nat
  := match t with
     | NatO ⇒ O
     | NatS x ⇒ S (denote x)
     | NatMul x y ⇒ denote x × denote y
     end.

Module Import EvennessChecking.
```

F.1.1 Theory on checking evenness of expressions

```
  Inductive is_even : nat → Prop :=
  | even_O : is_even O
  | even_SS : ∀ x, is_even x → is_even (S (S x)).

  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.

  Notation is_even_or_odd x := ({is_even x} + {~is_even x}).

  Lemma is_even_or_odd_x : ∀ x, (is_even x ∧ ¬is_even (S x))
                                  ∨ (~is_even x ∧ is_even (S x)).
  Proof.
    induction x as [|x [[IHx0 IHx1]|[IHx0 IHx1]]].
    { left; split; try constructor; intro H; inversion H. }
    { right; split; [ | constructor ]; assumption. }
    { left; split;
        [ assumption
        | intro H; inversion_clear H; apply IHx0; assumption ]. }
  Qed.

  Definition is_even_or_odd_S x (pf : is_even_or_odd x)
```

```
  : is_even_or_odd (S x).
Proof.
  destruct pf as [pf|pf]; [ right | left ];
    abstract (destruct (is_even_or_odd_x x) as [[H0 H1]|[H0 H1]]; tauto).
Defined.

Definition is_even_or_odd_sum
           x y
  : (is_even x ∧ is_even y ∧ is_even (x + y))
    ∨ (~is_even x ∧ ¬is_even y ∧ is_even (x + y))
    ∨ (is_even x ∧ ¬is_even y ∧ ¬is_even (x + y))
    ∨ (~is_even x ∧ is_even y ∧ ¬is_even (x + y)).
Proof.
  revert y; induction x as [|x IHx];
    intro y;
    [ | destruct (is_even_or_odd_x x) as [H|H]; specialize (IHx (S y));
        rewrite ← !plus_n_Sm in IHx ];
    destruct (is_even_or_odd_x y) as [Hy|Hy];
    intuition.
  { left; repeat constructor; assumption. }
  { right; right; left; repeat constructor; assumption. }
Qed.

Definition is_even_or_odd_mul_helper
           x y
  : (is_even x ∧ is_even y ∧ is_even (x × y))
    ∨ (~is_even x ∧ is_even y ∧ is_even (x × y))
    ∨ (is_even x ∧ ¬is_even y ∧ is_even (x × y))
    ∨ (~is_even x ∧ ¬is_even y ∧ ¬is_even (x × y)).
Proof.
  induction x as [|x IHx]; simpl.
  { destruct (is_even_or_odd_x y); intuition.
    { left; repeat constructor; assumption. }
    { right; right; left; repeat constructor; assumption. } }
  { pose proof (is_even_or_odd_sum y (x × y)).
    pose proof (is_even_or_odd_x x).
    intuition. }
Qed.

Definition is_even_or_odd_mul x y
           (Hx : is_even_or_odd x) (Hy : is_even_or_odd y)
  : is_even_or_odd (x × y).
Proof.
  destruct Hx, Hy; [ left | left | left | right ];
    abstract (pose proof (is_even_or_odd_mul_helper x y); intuition).
Defined.

Lemma check_is_even_expr_correct (e : expr)
```

```
    : check_is_even_expr e = true ↔ is_even (denote e).
  Proof.
    induction e; simpl in *.
    { repeat constructor. }
    { rewrite negb_true_iff, ← not_true_iff_false, IHe.
      edestruct (is_even_or_odd_x (denote e)); intuition. }
    { rewrite orb_true_iff, IHe1, IHe2.
      edestruct is_even_or_odd_mul_helper;
        intuition solve [ intuition eauto ]. }
  Qed.

  Theorem check_is_even_expr_sound (e : expr)
    : check_is_even_expr e = true → is_even (denote e).
  Proof. intro; apply check_is_even_expr_correct; assumption. Qed.

  Lemma cut_is_even_eq (x y : nat) (H : x = y) (Hx : is_even x)
    : is_even y.
  Proof. subst; assumption. Qed.
End EvennessChecking.
```

F.1.2 Reification by parametricity

```
Ltac Reify x :=
  let rx := lazymatch (eval pattern nat, O, S, Nat.mul in x) with
            | ?rx _ _ _ _ ⇒ rx end in
  constr:(rx expr NatO NatS NatMul).

Ltac Reify_rhs _ :=
  lazymatch goal with
  | [ ⊢ _ = ?RHS ]
    ⇒ let rv := Reify RHS in
       transitivity (denote rv);
       [ | lazy [denote]; reflexivity ]
  end.
```

F.1.3 Using reification to check evenness

```
Goal is_even (let x0 := 100 × 100 × 100 × 100 in
              let x1 := x0 × x0 × x0 × x0 in
              let x2 := x1 × x1 × x1 × x1 in
              x2).
Proof.
  eapply cut_is_even_eq.
  { Reify_rhs ().
    reflexivity. }
  apply check_is_even_expr_sound; vm_compute; reflexivity.
Qed.
```