15 Proof by Reflection

The last chapter highlighted a heuristic approach to proving. In this chapter, we study an alternative technique, *proof by reflection* [2]. We write, in Gallina, decision procedures with proofs of correctness and appeal to these procedures in writing very short proofs. Such a proof is checked by running the decision procedure. The term *reflection* applies because we need to translate Gallina propositions into values of inductive types representing syntax, so that Gallina programs may analyze them, and translating such a term back to the original form is called *reflecting* it.

15.1 Proving Evenness

Proving that particular natural number constants are even is certainly something we would rather have happen automatically. The Ltac programming techniques explained in Chapter 14 make it easy to implement such a procedure.

```
Inductive isEven : nat → Prop :=
| Even_O : isEven O
| Even_SS : ∀ n, isEven n → isEven (S (S n)).

Ltac prove_even := repeat constructor.

Theorem even_256 : isEven 256.
  prove_even.
Qed.

Print even_256.

even_256 =
Even_SS
  (Even_SS
     (Even_SS
        (Even_SS
```

and so on. This procedure always works (at least on machines with infinite resources), but it has a serious drawback, which we see when we print the proof it generates that 256 is even. The final proof term has length superlinear in the input value. Coq's implicit arguments mechanism is hiding the values given for parameter n of Even_SS, which is why the proof term only appears linear here. Also, proof terms are represented internally as syntax trees, with the opportunity for sharing node representations, but in this chapter we measure proof term size as simple textual length or as the number of nodes in the term's syntax tree, two measures that are approximately equivalent. Sometimes apparently large proof terms have enough internal sharing that they take up less memory than we expect, but one avoids having to reason about such sharing by ensuring that the size of a sharing-free version of a term is low enough.

Superlinear evenness proof terms seem excessively large, since we could write a trivial and trustworthy program to verify evenness of constants. The proof checker could simply call the program where needed. There are also no static typing guarantees that the tactic always behaves appropriately. Other invocations of similar tactics might fail with dynamic type errors, and we would not know about the bugs behind these errors until we attempted to prove complex goals.

The techniques of proof by reflection address both complaints. We can write proofs like the preceding example with constant size overhead beyond the size of the input, and do it with verified decision procedures written in Gallina.

We begin by using a type from the MoreSpecif module (see the book source) to write a certified evenness checker.

```
Print partial.
```

```
Inductive partial (P : Prop) : Set := Proved : P → [P]
  | Uncertain : [P]
```

A **partial** P value is an optional proof of P. The notation $[P]$ stands for **partial** P.

```
Local Open Scope partial_scope.
```

We bring into scope some notations for the **partial** type. These overlap with some of the notations seen previously for specification types, so they were placed in a separate scope that needs separate opening.

```
Definition check_even : ∀ n : nat, [isEven n].
  Hint Constructors isEven.

  refine (fix F (n : nat) : [isEven n] :=
```

```
    match n with
      | 0 ⇒ Yes
      | 1 ⇒ No
      | S (S n') ⇒ Reduce (F n')
    end); auto.
Defined.
```

The function check_even may be viewed as a *verified decision procedure*, because its type guarantees that it never returns `Yes` for inputs that are not even.

Now we can use dependent pattern matching to write a function that performs a surprising feat. When given a **partial** P, this function partialOut returns a proof of P if the **partial** value contains a proof, and it returns a (useless) proof of **True** otherwise. From the standpoint of ML and Haskell programming, it seems impossible to write such a type, but it is trivial with a `return` annotation.

```
Definition partialOut (P : Prop) (x : [P]) :=
  match x return (match x with
                    | Proved _ ⇒ P
                    | Uncertain ⇒ True
                  end) with
    | Proved pf ⇒ pf
    | Uncertain ⇒ I
  end.
```

A function like this is useful in writing a reflective version of the earlier *prove_even* tactic.

```
Ltac prove_even_reflective :=
  match goal with
    | [ ⊢ isEven ?N] ⇒ exact (partialOut (check_even N))
  end.
```

We identify which natural number we are considering and prove its evenness by pulling the proof out of the appropriate check_even call. Recall that the `exact` tactic proves a proposition P when given a proof term of precisely type P.

```
Theorem even_256' : isEven 256.
  prove_even_reflective.
Qed.

Print even_256'.

even_256' = partialOut (check_even 256)
      : isEven 256
```

We can see a constant wrapper around the object of the proof. For any even number, this form of proof will suffice. The size of the proof term is now linear in the number being checked, containing two repetitions of the unary form of that number, one of which is hidden within the implicit argument to partialOut.

What happens if we try the tactic with an odd number?

```
Theorem even_255 : isEven 255.
  prove_even_reflective.
```

```
User error: No matching clauses for match goal
```

The tactic fails. To see more precisely what goes wrong, we can run manually the body of the `match`.

```
  exact (partialOut (check_even 255)).
```

```
Error: The term "partialOut (check_even 255)" has type
"match check_even 255 with
 | Yes => isEven 255
 | No => True
 end" while it is expected to have type "isEven 255"
```

As usual, the type checker performs no reductions to simplify error messages. If we reduced the first term, we would see that check_even 255 reduces to a `No`, so that the first term is equivalent to **True**, which certainly does not unify with **isEven** 255.

```
Abort.
```

The tactic *prove_even_reflective* is reflective because it performs a proof search process (a trivial one, in this case) wholly within Gallina, where the only use of Ltac is to translate a goal into an appropriate use of check_even.

15.2 Reifying the Syntax of a Trivial Tautology Language

We might also like to have reflective proofs of trivial tautologies like this one:

```
Theorem true_galore : (True ∧ True)
  → (True ∨ (True ∧ (True → True))).
  tauto.
Qed.
```

```
Print true_galore.
```

```
true_galore =
fun H : True ∧ True ⇒
and_ind (fun _ _ : True ⇒ or_introl (True ∧ (True → True)) I) H
    : True ∧ True → True ∨ True ∧ (True → True)
```

As we might expect, the proof that `tauto` builds contains explicit applications of natural deduction rules. For large formulas, this can add a superlinear amount of proof size overhead, beyond the size of the input.

To write a reflective procedure for this class of goals, we need to get into the actual "reflection" part of "proof by reflection." It is impossible to case-analyze a `Prop` in any way in Gallina. We must *reify* `Prop` into some type that we can analyze. This inductive type is a good candidate:

```
Inductive taut : Set :=
| TautTrue : taut
| TautAnd : taut → taut → taut
| TautOr : taut → taut → taut
| TautImp : taut → taut → taut.
```

We write a recursive function to *reflect* this syntax back to `Prop`. Such functions are also called *interpretation functions*; they were used in previous examples to give semantics to small programming languages.

```
Fixpoint tautDenote (t : taut) : Prop :=
  match t with
    | TautTrue ⇒ True
    | TautAnd t1 t2 ⇒ tautDenote t1 ∧ tautDenote t2
    | TautOr t1 t2 ⇒ tautDenote t1 ∨ tautDenote t2
    | TautImp t1 t2 ⇒ tautDenote t1 → tautDenote t2
  end.
```

It is easy to prove that every formula in the range of tautDenote is true.

```
Theorem tautTrue : ∀ t, tautDenote t.
  induction t; crush.
Qed.
```

To use tautTrue to prove particular formulas, we need to implement the syntax reification process. A recursive Ltac function does the job.

```
Ltac tautReify P :=
  match P with
    | True ⇒ TautTrue
    | ?P1 ∧ ?P2 ⇒
```

```
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautAnd t1 t2)
    | ?P1 ∨ ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautOr t1 t2)
    | ?P1 → ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautImp t1 t2)
  end.
```

With *tautReify* available, it is easy to finish the reflective tactic. We look at the goal formula, reify it, and apply tautTrue to the reified formula.

```
Ltac obvious :=
  match goal with
    | [ ⊢ ?P ] ⇒
      let t := tautReify P in
        exact (tautTrue t)
  end.
```

We can verify that *obvious* solves the original example, with a proof term that does not mention details of the proof.

```
Theorem true_galore' : (True ∧ True)
  → (True ∨ (True ∧ (True → True))).
  obvious.
Qed.

Print true_galore'.

true_galore' =
tautTrue
  (TautImp (TautAnd TautTrue TautTrue)
     (TautOr TautTrue (TautAnd TautTrue
     (TautImp TautTrue TautTrue))))
     : True ∧ True → True ∨ True ∧ (True → True)
```

It is worth considering how the reflective tactic improves on a pure Ltac implementation. The formula reification process is just as ad hoc as before, so we gain little there. In general, proofs are more complicated than formula translation, and the generic proof rule applied here is on much better formal footing than a recursive Ltac function. The

dependent type of the proof guarantees that it works on any input formula. This benefit is in addition to the proof size improvement that we have already seen.

It may also be worth pointing out that the previous example of evenness testing used a function partialOut for sound handling of input goals that the verified decision procedure fails to prove. Here, we prove that the procedure tautTrue (recall that an inductive proof may be viewed as a recursive procedure) is able to prove any goal representable in **taut**, so no extra step is necessary.

15.3 A Monoid Expression Simplifier

Proof by reflection does not require encoding of all the syntax in a goal. We can insert variables into syntax types to allow injection of arbitrary pieces, even if we cannot apply specialized reasoning to them. In this section, we explore that possibility by writing a tactic for normalizing monoid equations.

```
Section monoid.
  Variable A : Set.
  Variable e : A.
  Variable f : A → A → A.

  Infix "+" := f.

  Hypothesis assoc : ∀ a b c, (a + b) + c = a + (b + c).
  Hypothesis identl : ∀ a, e + a = a.
  Hypothesis identr : ∀ a, a + e = a.
```

We add variables and hypotheses characterizing an arbitrary instance of the algebraic structure of monoids. We have an associative binary operator and an identity element for it.

It is easy to define an expression tree type for monoid expressions. A Var constructor is a catch-all case for subexpressions that we cannot model. These subexpressions could be actual Gallina variables, or they could just use functions that the tactic is unable to understand.

```
  Inductive mexp : Set :=
  | Ident : mexp
  | Var : A → mexp
  | Op : mexp → mexp → mexp.
```

Next, we write an interpretation function.

```
  Fixpoint mdenote (me : mexp) : A :=
    match me with
```

```
    | Ident ⇒ e
    | Var v ⇒ v
    | Op me1 me2 ⇒ mdenote me1 + mdenote me2
  end.
```

We normalize expressions by flattening them into lists, via associativity, so it is helpful to have a denotation function for lists of monoid values.

```
Fixpoint mldenote (ls : list A) : A :=
  match ls with
    | nil ⇒ e
    | x :: ls' ⇒ x + mldenote ls'
  end.
```

The flattening function itself is easy to implement.

```
Fixpoint flatten (me : mexp) : list A :=
  match me with
    | Ident ⇒ nil
    | Var x ⇒ x :: nil
    | Op me1 me2 ⇒ flatten me1 ++ flatten me2
  end.
```

This function has a straightforward correctness proof in terms of the *denote* functions.

```
Lemma flatten_correct' : ∀ ml2 ml1,
  mldenote ml1 + mldenote ml2 = mldenote (ml1 ++ ml2).
  induction ml1; crush.
Qed.

Theorem flatten_correct : ∀ me, mdenote me = mldenote (flatten me).
  Hint Resolve flatten_correct'.

  induction me; crush.
Qed.
```

Now it is easy to prove a theorem that will be the main tool behind the simplification tactic.

```
Theorem monoid_reflect : ∀ me1 me2,
  mldenote (flatten me1) = mldenote (flatten me2)
  → mdenote me1 = mdenote me2.
  intros; repeat rewrite flatten_correct; assumption.
Qed.
```

We implement reification into the **mexp** type.

```
Ltac reify me :=
```

```
    match me with
      | e ⇒ Ident
      | ?me1 + ?me2 ⇒
        let r1 := reify me1 in
        let r2 := reify me2 in
          constr:(Op r1 r2)
      | _ ⇒ constr:(Var me)
    end.
```

The final monoid tactic works on goals that equate two monoid terms. We reify each and change the goal to refer to the reified versions, finishing off by applying monoid_reflect and simplifying uses of mldenote. Recall that the change tactic replaces a conclusion formula with another that is definitionally equal to it.

```
  Ltac monoid :=
    match goal with
      | [ ⊢ ?me1 = ?me2 ] ⇒
        let r1 := reify me1 in
        let r2 := reify me2 in
          change (mdenote r1 = mdenote r2);
            apply monoid_reflect; simpl
    end.
```

We can make short work of theorems like this one:

```
  Theorem t1 : ∀ a b c d, a + b + c + d = a + (b + c) + d.
    intros; monoid.
```

```
============================
 a + (b + (c + (d + e))) = a + (b + (c + (d + e)))
```

The tactic has canonicalized both sides of the equality, such that we can finish the proof by reflexivity.

```
    reflexivity.
  Qed.
```

It is interesting to look at the form of the proof.

```
  Print t1.
```

```
t1 =
fun a b c d : A ⇒
monoid_reflect (Op (Op (Op (Var a) (Var b)) (Var c)) (Var d))
  (Op (Op (Var a) (Op (Var b) (Var c))) (Var d)) eq_refl
```

$$: \forall\ a\ b\ c\ d : A,\ a + b + c + d = a + (b + c) + d$$

The proof term contains only restatements of the equality operands in reified form, followed by a use of reflexivity on the shared canonical form.

```
End monoid.
```

Extensions of this basic approach are used in the implementations of the `ring` and `field` tactics that come packaged with Coq.

15.4 A Smarter Tautology Solver

Now we are ready to revisit the tautology solver example. We want to broaden the scope of the tactic to include formulas whose truth is not syntactically apparent. We want to allow injection of arbitrary formulas, just as arbitrary monoid expressions were allowed in the last example. Since we are working in a richer theory, it is important to be able to use equalities between different injected formulas. For instance, we cannot prove $P \to P$ by translating the formula into a value like Imp (Var P) (Var P), because a Gallina function has no way of comparing the two Ps for equality.

To arrive at an implementation satisfying these criteria, we introduce the `quote` tactic and its associated library.

```
Require Import Quote.

Inductive formula : Set :=
| Atomic : index → formula
| Truth : formula
| Falsehood : formula
| And : formula → formula → formula
| Or : formula → formula → formula
| Imp : formula → formula → formula.
```

The type **index** comes from the Quote library and represents a countable variable type. The rest of **formula**'s definition should be familiar by now.

The `quote` tactic implements injection from `Prop` into **formula**, but it is not quite as smart as we might like. In particular, it wants to treat function types specially, so it gets confused if function types are part of the structure we want to encode syntactically. To trick `quote` into not noticing our uses of function types to express logical implication, we need to declare a wrapper definition for implication (see related code in Section 14.4).

```
Definition imp (P1 P2 : Prop) := P1 → P2.
Infix "->" := imp (no associativity, at level 95).
```

Now we can define the denotation function.

```
Definition asgn := varmap Prop.

Fixpoint formulaDenote (atomics : asgn) (f : formula) : Prop :=
  match f with
    | Atomic v ⇒ varmap_find False v atomics
    | Truth ⇒ True
    | Falsehood ⇒ False
    | And f1 f2 ⇒
      formulaDenote atomics f1 ∧ formulaDenote atomics f2
    | Or f1 f2 ⇒
      formulaDenote atomics f1 ∨ formulaDenote atomics f2
    | Imp f1 f2 ⇒
      formulaDenote atomics f1 -> formulaDenote atomics f2
  end.
```

The **varmap** type family implements maps from **index** values. In this case, we define an assignment as a map from variables to `Prop`s. The interpretation function formulaDenote works with an assignment, and we use the varmap_find function to consult the assignment in the Atomic case. The first argument to varmap_find is a default value, in case the variable is not found.

```
Section my_tauto.
  Variable atomics : asgn.

  Definition holds (v : index) := varmap_find False v atomics.
```

We define some shorthand for a particular variable's being true, and now we are ready to define some helpful functions based on the ListSet module of the standard library, which presents a view of lists as sets.

```
  Require Import ListSet.

  Definition index_eq : ∀ x y : index, {x = y} + {x ≠ y}.
    decide equality.
  Defined.

  Definition add (s : set index) (v : index) := set_add index_eq v s.

  Definition In_dec : ∀ v (s : set index), {In v s} + {¬ In v s}.
    Local Open Scope specif_scope.

    intro; refine (fix F (s : set index) : {In v s} + {¬ In v s} :=
      match s with
        | nil ⇒ No
```

```
      | v' :: s' ⇒ index_eq v' v || F s'
    end); crush.
Defined.
```

We define what it means for all members of an index set to represent true propositions, and we prove some lemmas about this notion.

```
Fixpoint allTrue (s : set index) : Prop :=
  match s with
    | nil ⇒ True
    | v :: s' ⇒ holds v ∧ allTrue s'
  end.

Theorem allTrue_add : ∀ v s,
  allTrue s
  → holds v
  → allTrue (add s v).
  induction s; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
    end; crush.
Qed.

Theorem allTrue_In : ∀ v s,
  allTrue s
  → set_In v s
  → varmap_find False v atomics.
  induction s; crush.
Qed.

Hint Resolve allTrue_add allTrue_In.

Local Open Scope partial_scope.
```

Now we can write a function forward that implements deconstruction of hypotheses, expanding a compound formula into a set of sets of atomic formulas covering all possible cases introduced with use of Or. To handle consideration of multiple cases, the function takes in a continuation argument, which will be called once for each case.

The forward function has a dependent type, in the style of Chapter 6, guaranteeing correctness. The arguments to forward are a goal formula *f*, a set *known* of atomic formulas that we may assume are true, a hypothesis formula *hyp*, and a success continuation *cont* that we call when we have extended *known* to hold new truths implied by *hyp*.

```
Definition forward : ∀ (f : formula) (known : set index)
  (hyp : formula)
```

```
    (cont : ∀ known', [allTrue known' → formulaDenote atomics f]),
    [allTrue known → formulaDenote atomics hyp
      → formulaDenote atomics f].
    refine (fix F (f : formula) (known : set index) (hyp : formula)
      (cont : ∀ known', [allTrue known' → formulaDenote atomics f])
      : [allTrue known → formulaDenote atomics hyp
        → formulaDenote atomics f] :=
      match hyp with
        | Atomic v ⇒ Reduce (cont (add known v))
        | Truth ⇒ Reduce (cont known)
        | Falsehood ⇒ Yes
        | And h1 h2 ⇒
          Reduce (F (Imp h2 f) known h1 (fun known' ⇒
            Reduce (F f known' h2 cont)))
        | Or h1 h2 ⇒ F f known h1 cont && F f known h2 cont
        | Imp _ _ ⇒ Reduce (cont known)
      end); crush.
  Defined.
```

A backward function implements analysis of the final goal. It calls forward to handle implications.

```
  Definition backward : ∀ (known : set index) (f : formula),
    [allTrue known → formulaDenote atomics f].
    refine (fix F (known : set index) (f : formula)
      : [allTrue known → formulaDenote atomics f] :=
      match f with
        | Atomic v ⇒ Reduce (In_dec v known)
        | Truth ⇒ Yes
        | Falsehood ⇒ No
        | And f1 f2 ⇒ F known f1 && F known f2
        | Or f1 f2 ⇒ F known f1 || F known f2
        | Imp f1 f2 ⇒
          forward f2 known f1 (fun known' ⇒ F known' f2)
      end); crush; eauto.
  Defined.
```

A simple wrapper around backward gives us the usual type of a partial decision procedure.

```
  Definition my_tauto : ∀ f : formula, [formulaDenote atomics f].
    intro; refine (Reduce (backward nil f)); crush.
  Defined.
End my_tauto.
```

The final tactic implementation is now fairly straightforward. First, we `intro` all quantifiers that do not bind `Prop`s. Then we call the `quote` tactic, which implements the reification. Finally, we are able to construct an exact proof via partialOut and the my_tauto Gallina function.

```
Ltac my_tauto :=
  repeat match goal with
           | [ ⊢ ∀ x : ?P, _ ] ⇒
             match type of P with
               | Prop ⇒ fail 1
               | _ ⇒ intro
             end
         end;
  quote formulaDenote;
  match goal with
    | [ ⊢ formulaDenote ?m ?f ] ⇒ exact (partialOut (my_tauto m f))
  end.
```

A few examples demonstrate how the tactic works.

```
Theorem mt1 : True.
  my_tauto.
Qed.

Print mt1.

mt1 = partialOut (my_tauto (Empty_vm Prop) Truth)
      : True
```

We see my_tauto applied with an empty **varmap**, since every subformula is handled by formulaDenote.

```
Theorem mt2 : ∀ x y : nat, x = y -> x = y.
  my_tauto.
Qed.

Print mt2.

mt2 =
fun x y : nat ⇒
partialOut
  (my_tauto (Node_vm (x = y) (Empty_vm Prop) (Empty_vm Prop))
     (Imp (Atomic End_idx) (Atomic End_idx)))
     : ∀ x y : nat, x = y -> x = y
```

Crucially, both instances of $x = y$ are represented with the same index, End_idx. The value of this index only needs to appear once in

the **varmap**, whose form reveals that **varmap**s are represented as binary trees, where **index** values denote paths from tree roots to leaves.

```
Theorem mt3 : ∀ x y z,
  (x < y ∧ y > z) ∨ (y > z ∧ x < S y)
  -> y > z ∧ (x < y ∨ x < S y).
  my_tauto.
Qed.

Print mt3.

fun x y z : nat ⇒
partialOut
  (my_tauto
    (Node_vm (x < S y) (Node_vm (x < y) (Empty_vm Prop)
      (Empty_vm Prop))
      (Node_vm (y > z) (Empty_vm Prop) (Empty_vm Prop)))
    (Imp
      (Or (And (Atomic (Left_idx End_idx))
          (Atomic (Right_idx End_idx)))
          (And (Atomic (Right_idx End_idx)) (Atomic End_idx)))
      (And (Atomic (Right_idx End_idx))
          (Or (Atomic (Left_idx End_idx)) (Atomic End_idx)))))
    : ∀ x y z : nat,
      x < y ∧ y > z ∨ y > z ∧ x < S y
      -> y > z ∧ (x < y ∨ x < S y)
```

The goal contained three distinct atomic formulas, and we see that a three-element **varmap** is generated.

It can be interesting to observe differences between the level of repetition in proof terms generated by my_tauto and tauto for especially trivial theorems.

```
Theorem mt4 :
  True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False -> False.
  my_tauto.
Qed.

Print mt4.

mt4 =
partialOut
  (my_tauto (Empty_vm Prop)
    (Imp
      (And Truth
          (And Truth
```

```
                (And Truth (And Truth (And Truth
                   (And Truth Falsehood))))))
          Falsehood))
     : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False -> False

Theorem mt4' :
  True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False → False.
  tauto.
Qed.

Print mt4'.

mt4' =
fun H : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False ⇒
and_ind
  (fun (_ : True)
   (H1 : True ∧ True ∧ True ∧ True ∧ True ∧ False) ⇒
   and_ind
     (fun (_ : True) (H3 : True ∧ True ∧ True ∧ True ∧ False) ⇒
      and_ind
        (fun (_ : True) (H5 : True ∧ True ∧ True ∧ False) ⇒
         and_ind
           (fun (_ : True) (H7 : True ∧ True ∧ False) ⇒
            and_ind
              (fun (_ : True) (H9 : True ∧ False) ⇒
               and_ind (fun (_ : True) (H11 : False) ⇒
                  False_ind False H11)
                  H9) H7) H5) H3) H1) H
     : True ∧ True ∧ True ∧ True ∧ True ∧ True ∧ False → False
```

The traditional `tauto` tactic introduces a quadratic blow-up in the size of the proof term, whereas proofs produced by `my_tauto` always have linear size.

15.4.1 Manual Reification of Terms with Variables

The action of the `quote` tactic may seem like magic. Somehow it performs equality comparison between subterms of arbitrary types, so that these subterms may be represented with the same reified variable. While `quote` is implemented in OCaml, we can code the reification process completely in Ltac as well. To make the job simpler, we represent variables as **nat**s, indexing into a simple list of variable values that may be referenced.

Step one of the process is to crawl over a term, building a duplicate-free list of all values that appear in positions encoded as variables. A useful helper function adds an element to a list, preventing duplicates. Note how we use Ltac pattern matching to implement an equality test on Gallina terms; this is simple syntactic equality, not even the richer definitional equality. We also represent lists as nested tuples, to allow different list elements to have different Gallina types.

```
Ltac inList x xs :=
  match xs with
    | tt ⇒ false
    | (x, _) ⇒ true
    | (_, ?xs') ⇒ inList x xs'
  end.

Ltac addToList x xs :=
  let b := inList x xs in
    match b with
      | true ⇒ xs
      | false ⇒ constr:(x, xs)
    end.
```

Now we can write the recursive function to calculate the list of variable values we want to use to represent a term.

```
Ltac allVars xs e :=
  match e with
    | True ⇒ xs
    | False ⇒ xs
    | ?e1 ∧ ?e2 ⇒
      let xs := allVars xs e1 in
        allVars xs e2
    | ?e1 ∨ ?e2 ⇒
      let xs := allVars xs e1 in
        allVars xs e2
    | ?e1 → ?e2 ⇒
      let xs := allVars xs e1 in
        allVars xs e2
    | _ ⇒ addToList e xs
  end.
```

We will also need a way to map a value to its position in a list.

```
Ltac lookup x xs :=
  match xs with
    | (x, _) ⇒ O
```

```
    | (_, ?xs') ⇒
      let n := lookup x xs' in
        constr:(S n)
  end.
```

The next building block is a procedure for reifying a term, given a list of all allowed variable values. We are free to make this procedure partial, where tactic failure may be triggered upon attempting to reify a term containing subterms not included in the list of variables. The type of the output term is a copy of **formula** where **index** is replaced by **nat**, in the type of the constructor for atomic formulas.

```
Inductive formula' : Set :=
| Atomic' : nat → formula'
| Truth' : formula'
| Falsehood' : formula'
| And' : formula' → formula' → formula'
| Or' : formula' → formula' → formula'
| Imp' : formula' → formula' → formula'.
```

Note that when we write the Ltac procedure, we can work directly with the normal → operator rather than needing to introduce a wrapper for it.

```
Ltac reifyTerm xs e :=
  match e with
    | True ⇒ constr:Truth'
    | False ⇒ constr:Falsehood'
    | ?e1 ∧ ?e2 ⇒
      let p1 := reifyTerm xs e1 in
      let p2 := reifyTerm xs e2 in
        constr:(And' p1 p2)
    | ?e1 ∨ ?e2 ⇒
      let p1 := reifyTerm xs e1 in
      let p2 := reifyTerm xs e2 in
        constr:(Or' p1 p2)
    | ?e1 → ?e2 ⇒
      let p1 := reifyTerm xs e1 in
      let p2 := reifyTerm xs e2 in
        constr:(Imp' p1 p2)
    | _ ⇒
      let n := lookup e xs in
        constr:(Atomic' n)
  end.
```

Finally, we bring all the pieces together.

```
Ltac reify :=
  match goal with
    | [ ⊢ ?G ] ⇒ let xs := allVars tt G in
      let p := reifyTerm xs G in
        pose p
  end.
```

A quick test verifies that we are doing reification correctly.

```
Theorem mt3' : ∀ x y z,
  (x < y ∧ y > z) ∨ (y > z ∧ x < S y)
  → y > z ∧ (x < y ∨ x < S y).
  do 3 intro; reify.
```

A simple tactic adds the translated term as a new variable.

```
f := Imp'
       (Or' (And' (Atomic' 2) (Atomic' 1))
         (And' (Atomic' 1) (Atomic' 0)))
       (And' (Atomic' 1) (Or' (Atomic' 2) (Atomic' 0))) : formula'

Abort.
```

More work would be needed to complete the reflective tactic, as we must connect the new syntax type with the real meanings of formulas, but the details are the same as in the prior implementation with `quote`.

15.5 Building a Reification Tactic That Recurses under Binders

All examples so far have stayed away from reifying the syntax of terms that use such features as quantifiers and `fun` function abstractions. Such cases are complicated by the fact that different subterms may be allowed to reference different sets of free variables. Some cleverness is needed to clear this hurdle, but a few simple patterns will suffice. Consider this example of a simple dependently typed term language, where a function abstraction body is represented conveniently with a Coq function:

```
Inductive type : Type :=
| Nat : type
| NatFunc : type → type.

Inductive term : type → Type :=
| Const : nat → term Nat
| Plus : term Nat → term Nat → term Nat
```

```
| Abs : ∀ t, (nat → term t) → term (NatFunc t).

Fixpoint typeDenote (t : type) : Type :=
  match t with
    | Nat ⇒ nat
    | NatFunc t ⇒ nat → typeDenote t
  end.

Fixpoint termDenote t (e : term t) : typeDenote t :=
  match e with
    | Const n ⇒ n
    | Plus e1 e2 ⇒ termDenote e1 + termDenote e2
    | Abs _ e1 ⇒ fun x ⇒ termDenote (e1 x)
  end.
```

Here is a naïve first attempt at a reification tactic:

```
Ltac refl' e :=
  match e with
    | ?E1 + ?E2 ⇒
      let r1 := refl' E1 in
      let r2 := refl' E2 in
        constr:(Plus r1 r2)

    | fun x : nat ⇒ ?E1 ⇒
      let r1 := refl' E1 in
        constr:(Abs (fun x ⇒ r1 x))

    | _ ⇒ constr:(Const e)
  end.
```

Recall that a regular Ltac pattern variable $?X$ only matches terms that *do not mention new variables introduced within the pattern.* In the naïve implementation, the case for matching function abstractions matches the function body in a way that prevents it from mentioning the function argument. The code structures the function body in a way that leads to independent problems, but we could change it so that it handles function abstractions that ignore their arguments.

To handle functions in general, we use the pattern variable form $@?X$, which allows X to mention newly introduced variables that are declared explicitly. A use of $@?X$ must be followed by a list of the local variables that may be mentioned. The variable X then comes to stand for a Gallina function over the values of those variables. For instance,

```
Reset refl'.
Ltac refl' e :=
```

```
  match e with
    | ?E1 + ?E2 ⇒
      let r1 := refl' E1 in
      let r2 := refl' E2 in
        constr:(Plus r1 r2)

    | fun x : nat ⇒ @?E1 x ⇒
      let r1 := refl' E1 in
        constr:(Abs r1)

    | _ ⇒ constr:(Const e)
  end.
```

Now, in the abstraction case, we bind *E1* as a function from an x value to the value of the abstraction body. Unfortunately, the recursive call there is not destined for success. It will match the same abstraction pattern and trigger another recursive call, and so on, through infinite recursion. One last refactoring yields a working procedure. The key idea is to consider every input to *refl'* as *a function over the values of variables introduced during recursion.*

```
Reset refl'.
Ltac refl' e :=
  match eval simpl in e with
    | fun x : ?T ⇒ @?E1 x + @?E2 x ⇒
      let r1 := refl' E1 in
      let r2 := refl' E2 in
        constr:(fun x ⇒ Plus (r1 x) (r2 x))

    | fun (x : ?T) (y : nat) ⇒ @?E1 x y ⇒
      let r1 := refl' (fun p : T × nat ⇒ E1 (fst p) (snd p)) in
        constr:(fun x ⇒ Abs (fun y ⇒ r1 (x, y)))

    | _ ⇒ constr:(fun x ⇒ Const (e x))
  end.
```

Note how even the addition case now works in terms of functions, with $@?X$ patterns. The abstraction case introduces a new variable by extending the type used to represent the free variables. In particular, the argument to *refl'* uses type T to represent all free variables. We extend the type to $T \times \mathbf{nat}$ for the type representing free variable values within the abstraction body. A bit of bookkeeping with pairs and their projections produces an appropriate version of the abstraction body to pass in a recursive call. To ensure that this repackaging of terms does

not interfere with pattern matching, we add an extra `simpl` reduction on the function argument, in the first line of the body of *refl'*.

Now one more tactic provides an example of how to apply reification. Let us consider goals that are equalities between terms that can be reified. We want to change such goals into equalities between appropriate calls to termDenote.

```
Ltac refl :=
  match goal with
    | [ ⊢ ?E1 = ?E2 ] ⇒
      let E1' := refl' (fun _ : unit ⇒ E1) in
      let E2' := refl' (fun _ : unit ⇒ E2) in
        change (termDenote (E1' tt) = termDenote (E2' tt));
          cbv beta iota delta [fst snd]
  end.

Goal (fun (x y : nat) ⇒ x + y + 13) = (fun (_ z : nat) ⇒ z).
  refl.

  ============================
   termDenote
     (Abs
        (fun y : nat ⇒
         Abs (fun y0 : nat ⇒ Plus (Plus (Const y) (Const y0))
         (Const 13)))) =
   termDenote (Abs (fun _ : nat ⇒ Abs (fun y0 : nat ⇒ Const y0)))

Abort.
```

The encoding here uses Coq functions to represent binding within the terms we reify, which makes it difficult to implement certain functions over reified terms. An alternative would be to represent variables with numbers. This can be done by writing a slightly smarter reification function that identifies variable references by detecting when term arguments are just compositions of fst and snd; from the order of the compositions we may read off the variable number. The details are left as an exercise (though not a trivial one) for the reader.