17 Reasoning about Programming Language Syntax

Reasoning about the syntax and semantics of programming languages is a popular application of proof assistants. Before proving the first theorem of this kind, it is necessary to choose a formal encoding of the informal notions of syntax, dealing with such issues as variable-binding conventions. I believe the pragmatic questions in this domain are far from settled and remain important open research problems. However, in this chapter, I demonstrate two underused encoding approaches. Note that I am not recommending either approach as a complete solution for all contexts. For a broader introduction to programming language formalization, using more elementary techniques, see *Software Foundations* by Pierce et al.[7]

This chapter is meant to serve as a case study, bringing together concepts from previous chapters. There is a concrete example of the importance of representation choices. Translating mathematics to Coq is not a deterministic process, and different creative choices can have big impacts. We will also see dependent types and scripted proof automation in action, applied to solve a particular problem as well as possible rather than to demonstrate new Coq concepts.

I make a few remarks intended to relate the material with common ideas in semantics, but readers not familiar with the theory of programming language semantics can safely disregard them.

We study a small programming language and reason about its semantics, expressed as an interpreter into Coq terms, much as in examples throughout the book. It is helpful to build a slight extension of *crush* that tries to apply the functional extensionality axiom, which says that two functions are equal if they map equal inputs to equal outputs (see Section 10.6).

```
Ltac ext := let x := fresh "x" in extensionality x.
```

7. http://www.cis.upenn.edu/~bcpierce/sf/

```
Ltac pl := crush; repeat (ext || f_equal; crush).
```

At this point in the book source, some auxiliary proofs also appear.

Here is a definition of the type system used throughout this chapter. It is for simply typed lambda calculus with natural numbers as the base type.

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

Fixpoint typeDenote (t : type) : Type :=
  match t with
    | Nat ⇒ nat
    | Func t1 t2 ⇒ typeDenote t1 → typeDenote t2
  end.
```

Now we can choose how to represent the syntax of programs. The two sections of the chapter explore two such choices, demonstrating the effect the choice has on proof complexity.

17.1 Dependent de Bruijn Indices

The first encoding is the *dependent de Bruijn index encoding* (see Chapter 9). We represent program syntax terms in a type family parameterized by a list of types, representing the *typing context*, or information on which free variables are in scope and what their types are. Variables are represented in a way isomorphic to the natural numbers, where number 0 represents the first element in the context, number 1 the second element, and so on. Actually, instead of numbers, we use the **member** dependent type family from Chapter 9.

```
Module FirstOrder.
```

Here is the definition of the **term** type, including variables, constants, addition, function abstraction and application, and let binding of local variables:

```
  Inductive term : list type → type → Type :=
  | Var : ∀ G t, member t G → term G t

  | Const : ∀ G, nat → term G Nat
  | Plus : ∀ G, term G Nat → term G Nat → term G Nat

  | Abs : ∀ G dom ran,
```

```
    term (dom :: G) ran → term G (Func dom ran)
  | App : ∀ G dom ran,
    term G (Func dom ran) → term G dom → term G ran

  | Let : ∀ G t1 t2, term G t1 → term (t1 :: G) t2 → term G t2.

  Implicit Arguments Const [G].
```

Here are two example term encodings, the first of addition packaged as a two-argument curried function, and the second of a sample application of addition to constants:

```
Example add : term nil (Func Nat (Func Nat Nat)) :=
  Abs (Abs (Plus (Var (HNext HFirst)) (Var HFirst))).

Example three_the_hard_way : term nil Nat :=
  App (App add (Const 1)) (Const 2).
```

Since dependent typing ensures that any term is well-formed in its context and has a particular type, it is easy to translate syntactic terms into Coq values.

```
Fixpoint termDenote G t (e : term G t)
  : hlist typeDenote G → typeDenote t :=
  match e with
    | Var _ _ x ⇒ fun s ⇒ hget s x

    | Const _ n ⇒ fun _ ⇒ n
    | Plus _ e1 e2 ⇒ fun s ⇒ termDenote e1 s + termDenote e2 s

    | Abs _ _ _ e1 ⇒ fun s ⇒ fun x ⇒ termDenote e1 (x ::: s)
    | App _ _ _ e1 e2 ⇒ fun s ⇒
      (termDenote e1 s) (termDenote e2 s)

    | Let _ _ _ e1 e2 ⇒ fun s ⇒
      termDenote e2 (termDenote e1 s ::: s)
  end.
```

With this term representation, some program transformations are easy to implement and prove correct. Certainly we would worry if this were not the case for the *identity* transformation, which takes a term apart and reassembles it.

```
Fixpoint ident G t (e : term G t) : term G t :=
  match e with
    | Var _ _ x ⇒ Var x
```

```
    | Const _ n ⇒ Const n
    | Plus _ e1 e2 ⇒ Plus (ident e1) (ident e2)

    | Abs _ _ _ e1 ⇒ Abs (ident e1)
    | App _ _ _ e1 e2 ⇒ App (ident e1) (ident e2)

    | Let _ _ _ e1 e2 ⇒ Let (ident e1) (ident e2)
  end.

Theorem identSound : ∀ G t (e : term G t) s,
  termDenote (ident e) s = termDenote e s.
  induction e; pl.
Qed.
```

A slightly more ambitious transformation belongs to the family of *constant folding* optimizations used as examples in other chapters.

```
Fixpoint cfold G t (e : term G t) : term G t :=
  match e with
    | Plus _ e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
      let maybeOpt := match e1' return _ with
                        | Const _ n1 ⇒
                          match e2' return _ with
                            | Const _ n2 ⇒
                              Some (Const (n1 + n2))
                            | _ ⇒ None
                          end
                        | _ ⇒ None
                      end in
      match maybeOpt with
        | None ⇒ Plus e1' e2'
        | Some e' ⇒ e'
      end

    | Abs _ _ _ e1 ⇒ Abs (cfold e1)
    | App _ _ _ e1 e2 ⇒ App (cfold e1) (cfold e2)

    | Let _ _ _ e1 e2 ⇒ Let (cfold e1) (cfold e2)

    | e ⇒ e
  end.
```

The correctness proof is more complex, but only slightly so.

```
Theorem cfoldSound : ∀ G t (e : term G t) s,
  termDenote (cfold e) s = termDenote e s.
  induction e; pl;
    repeat (match goal with
              | [ ⊢ context[match ?E with Var _ _ _ ⇒ _
                                        | _ ⇒ _ end] ] ⇒
                dep_destruct E
            end; pl).
Qed.
```

The transformations so far have been straightforward because they do not have interesting effects on the variable-binding structure of terms. The dependent de Bruijn representation is called *first-order* because it encodes variable identity explicitly; all such representations incur bookkeeping overheads in transformations that rearrange binding structure.

As an example of a more complex transformation, consider one that removes all uses of "`let` *x* = *e1* `in` *e2*" by substituting *e1* for *x* in *e2*. We implement the translation by pairing the compile-time typing environment with a run-time value environment or *substitution*, mapping each variable to a value to be substituted for it. Such a substitute term may be placed within a program in a position with a larger typing environment than applied at the point where the substitute term was chosen. To support such context transplantation, we need *lifting*, a standard de Bruijn indices operation. With dependent typing, lifting corresponds to weakening for typing judgments.

The fundamental goal of lifting is to add a new variable to a typing context, maintaining the validity of a term in the expanded context. To express the operation of adding a type to a context, we use a helper function insertAt.

```
Fixpoint insertAt (t : type) (G : list type) (n : nat) {struct n}
  : list type :=
  match n with
    | O ⇒ t :: G
    | S n' ⇒ match G with
                | nil ⇒ t :: G
                | t' :: G' ⇒ t' :: insertAt t G' n'
              end
  end.
```

Another function lifts bound variable instances, which we represent with **member** values.

```
Fixpoint liftVar t G (x : member t G) t' n
  : member t (insertAt t' G n) :=
  match x with
    | HFirst G' ⇒ match n return member t
                        (insertAt t' (t :: G') n) with
                        | O ⇒ HNext HFirst
                        | _ ⇒ HFirst
                      end
    | HNext t'' G' x' ⇒ match n return member t
                              (insertAt t' (t'' :: G') n) with
                              | O ⇒ HNext (HNext x')
                              | S n' ⇒ HNext (liftVar x' t' n')
                            end
  end.
```

The final helper function for lifting allows us to insert a new variable anywhere in a typing context.

```
Fixpoint lift' G t' n t (e : term G t) : term (insertAt t' G n) t :=
  match e with
    | Var _ _ x ⇒ Var (liftVar x t' n)

    | Const _ n ⇒ Const n
    | Plus _ e1 e2 ⇒ Plus (lift' t' n e1) (lift' t' n e2)

    | Abs _ _ _ e1 ⇒ Abs (lift' t' (S n) e1)
    | App _ _ _ e1 e2 ⇒ App (lift' t' n e1) (lift' t' n e2)

    | Let _ _ _ e1 e2 ⇒ Let (lift' t' n e1) (lift' t' (S n) e2)
  end.
```

In the Let removal transformation, we only need to apply lifting to add a new variable at the *beginning* of a typing context, so we package lifting into this final, simplified form.

```
Definition lift G t' t (e : term G t) : term (t' :: G) t :=
  lift' t' O e.
```

Finally, we can implement Let removal. The argument of type **hlist** (**term** *G'*) *G* represents a substitution mapping each variable from context *G* into a term that is valid in context *G'*. Note how the Abs case (1) extends via lifting the substitution *s* to hold in the broader context of the abstraction body *e1*, and (2) maps the new first variable to itself.

It is only the Let case that maps a variable to any substitute besides itself.

```
Fixpoint unlet G t (e : term G t) G'
  : hlist (term G') G → term G' t :=
  match e with
    | Var _ _ x ⇒ fun s ⇒ hget s x

    | Const _ n ⇒ fun _ ⇒ Const n
    | Plus _ e1 e2 ⇒ fun s ⇒ Plus (unlet e1 s) (unlet e2 s)

    | Abs _ _ _ e1 ⇒ fun s ⇒
      Abs (unlet e1 (Var HFirst ::: hmap (lift _) s))
    | App _ _ _ e1 e2 ⇒ fun s ⇒ App (unlet e1 s) (unlet e2 s)

    | Let _ _ _ e1 e2 ⇒ fun s ⇒ unlet e2 (unlet e1 s ::: s)
  end.
```

We have finished defining the transformation, but the parade of helper functions is not done. To prove correctness, we use one more helper function and a few lemmas. First, we need an operation to insert a new value into a substitution at a particular position.

```
Fixpoint insertAtS (t : type) (x : typeDenote t) (G : list type)
  (n : nat) {struct n}
  : hlist typeDenote G → hlist typeDenote (insertAt t G n) :=
  match n with
    | O ⇒ fun s ⇒ x ::: s
    | S n' ⇒ match G return hlist typeDenote G
                            → hlist typeDenote
                            (insertAt t G (S n')) with
                | nil ⇒ fun s ⇒ x ::: s
                | t' :: G' ⇒ fun s ⇒
                  hhd s ::: insertAtS t x n' (htl s)
              end
  end.

Implicit Arguments insertAtS [t G].
```

Next we prove that liftVar is correct. That is, a lifted variable retains its value with respect to a substitution when we perform an analogue to lifting by inserting a new mapping into the substitution.

```
Lemma liftVarSound : ∀ t' (x : typeDenote t') t G (m : member t G)
  s n,
  hget s m = hget (insertAtS x n s) (liftVar m t' n).
```

```
  induction m; destruct n; dep_destruct s; pl.
Qed.

Hint Resolve liftVarSound.
```

An analogous lemma establishes correctness of lift'.

```
Lemma lift'Sound : ∀ G t' (x : typeDenote t') t (e : term G t) n s,
  termDenote e s = termDenote (lift' t' n e) (insertAtS x n s).
  induction e; pl;
    repeat match goal with
             | [ IH : ∀ n s, _ = termDenote (lift' _ n ?E) _
                 ⊢ context[lift' _ (S ?N) ?E] ] ⇒
               specialize (IH (S N))
           end; pl.
Qed.
```

Correctness of lift itself is an easy corollary.

```
Lemma liftSound : ∀ G t' (x : typeDenote t') t (e : term G t) s,
  termDenote (lift t' e) (x ::: s) = termDenote e s.
  unfold lift; intros; rewrite (lift'Sound _ x e O); trivial.
Qed.

Hint Rewrite hget_hmap hmap_hmap liftSound.
```

Finally, we prove correctness of unlet for terms in arbitrary typing environments.

```
Lemma unletSound' : ∀ G t (e : term G t) G' (s : hlist (term G') G)
  s1,
  termDenote (unlet e s) s1
  = termDenote e (hmap (fun t' (e' : term G' t') ⇒
    termDenote e' s1) s).
  induction e; pl.
Qed.
```

The lemma statement is complex, with all its details of typing contexts and substitutions. It is usually prudent to state a final theorem in as simple a way as possible, to make clear that the statement properly formalizes its informal counterpart. We do that here for the simple case of terms with empty typing contexts.

```
Theorem unletSound : ∀ t (e : term nil t),
  termDenote (unlet e HNil) HNil = termDenote e HNil.
  intros; apply unletSound'.
Qed.

End FirstOrder.
```

The Let removal optimization is a good case study of a simple transformation that may turn out to be much more work than expected, based on representation choices. In the next section, we consider a better choice.

17.2 Parametric Higher-Order Abstract Syntax

In contrast to first-order encodings, *higher-order encodings* avoid explicit modeling of variable identity. Instead, the binding constructs of an object language (the language being formalized) can be represented using the binding constructs of the metalanguage (the language in which the formalization is done). The best known higher-order encoding is called *higher-order abstract syntax* (HOAS) [35]. We start by attempting to apply it directly in Coq.

```
Module HigherOrder.
```

With HOAS, each object language binding construct is represented with a *function* of the metalanguage. Here is the result of applying that idea within an inductive definition of term syntax:

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

| Abs : ∀ dom ran, (term dom → term ran) → term (Func dom ran)
| App : ∀ dom ran, term (Func dom ran) → term dom → term ran

| Let : ∀ t1 t2, term t1 → (term t1 → term t2) → term t2.
```

However, Coq rejects this definition for failing to meet the strict positivity restriction. For instance, the constructor Abs takes an argument that is a function over the same type family **term** that we are defining. Inductive definitions of this kind can be used to write nonterminating Gallina programs, which breaks the consistency of Coq's logic.

An alternative higher-order encoding is *parametric HOAS* (PHOAS), as introduced by Washburn and Weirich [46] for Haskell and tweaked by me [5] for use in Coq. Here the idea is to parameterize the syntax type by a type family standing for a *representation of variables*.

```
Section var.
  Variable var : type → Type.

  Inductive term : type → Type :=
  | Var : ∀ t, var t → term t
```

```
  | Const : nat → term Nat
  | Plus : term Nat → term Nat → term Nat

  | Abs : ∀ dom ran, (var dom → term ran) → term (Func dom ran)
  | App : ∀ dom ran, term (Func dom ran) → term dom → term ran

  | Let : ∀ t1 t2, term t1 → (var t1 → term t2) → term t2.
End var.

Implicit Arguments Var [var t].
Implicit Arguments Const [var].
Implicit Arguments Abs [var dom ran].
```

Coq accepts this definition because the embedded functions now merely take *variables* as arguments instead of arbitrary terms. One might wonder whether there is an easy loophole to exploit here, instantiating the parameter *var* as **term** itself. However, to do that, we would need to choose a variable representation for this nested mention of **term**, and so on, through an infinite descent into **term** arguments.

We write the final type of a closed term using polymorphic quantification over all possible choices of *var* type family.

```
Definition Term t := ∀ var, term var t.
```

Here are the new representations of the example terms from the last section. Note how each is written as a function over a *var* choice, such that the specific choice has no impact on the structure of the term.

```
Example add : Term (Func Nat (Func Nat Nat)) := fun var ⇒
  Abs (fun x ⇒ Abs (fun y ⇒ Plus (Var x) (Var y))).

Example three_the_hard_way : Term Nat := fun var ⇒
  App (App (add var) (Const 1)) (Const 2).
```

The argument *var* does not even appear in the function body for add. How can that be? By giving the terms expressive types, we allow Coq to infer many arguments for us. In fact, we do not even need to name the *var* argument.

```
Example add' : Term (Func Nat (Func Nat Nat)) := fun _ ⇒
  Abs (fun x ⇒ Abs (fun y ⇒ Plus (Var x) (Var y))).

Example three_the_hard_way' : Term Nat := fun _ ⇒
  App (App (add' _) (Const 1)) (Const 2).
```

Even though the *var* formal parameters appear as underscores, they *are* mentioned in the function bodies that type inference calculates.

17.2.1 Functional Programming with PHOAS

It may not be obvious that the PHOAS representation admits the crucial computable operations. The key to effective deconstruction of PHOAS terms is one principle: treat the *var* parameter as an unconstrained choice of *which data should be annotated on each variable*. We begin with a simple example, that of counting how many variable nodes appear in a PHOAS term. This operation requires no data annotated on variables, so we simply annotate variables with **unit** values. Note that when we go under binders in the cases for Abs and Let, we must provide the data value to annotate on the new variable we pass beneath. For the current choice of **unit** data, we always pass tt.

```
Fixpoint countVars t (e : term (fun _ ⇒ unit) t) : nat :=
  match e with
    | Var _ _ ⇒ 1

    | Const _ ⇒ 0
    | Plus e1 e2 ⇒ countVars e1 + countVars e2

    | Abs _ _ e1 ⇒ countVars (e1 tt)
    | App _ _ e1 e2 ⇒ countVars e1 + countVars e2

    | Let _ _ e1 e2 ⇒ countVars e1 + countVars (e2 tt)
  end.
```

This definition may seem a bit peculiar. Why may we represent variables as **unit** values? Recall that the final representation of closed terms is as polymorphic functions. We merely specialize a closed term to exactly the right variable representation for the transformation we wish to perform.

```
Definition CountVars t (E : Term t) :=
  countVars (E (fun _ ⇒ unit)).
```

It is easy to test that CountVars operates properly.

```
Eval compute in CountVars three_the_hard_way.
```

```
= 2
```

In fact, PHOAS can be used anywhere that first-order representations can. I do not go into detail here, but the intuition is that it is possible to interconvert between PHOAS and any reasonable first-order representation. Here is a suggestive example, translating PHOAS terms into strings giving a first-order rendering. To implement this translation, the key insight is to tag variables with strings, giving their names.

The function takes as an additional input a string giving the name to be assigned to the next variable introduced. We evolve this name by adding a prime to its end. To avoid getting bogged down in orthogonal details, we render all constants as the string "N".

```
Require Import String.
Open Scope string_scope.

Fixpoint pretty t (e : term (fun _ ⇒ string) t) (x : string)
  : string :=
  match e with
    | Var _ s ⇒ s

    | Const _ ⇒ "N"
    | Plus e1 e2 ⇒ "(" ++ pretty e1 x ++ " + " ++ pretty e2 x ++ ")"

    | Abs _ _ e1 ⇒
      "(fun " ++ x ++ " => " ++ pretty (e1 x) (x ++ "'") ++ ")"
    | App _ _ e1 e2 ⇒
      "(" ++ pretty e1 x ++ " " ++ pretty e2 x ++ ")"

    | Let _ _ e1 e2 ⇒ "(let " ++ x ++ " = " ++ pretty e1 x ++ " in "
      ++ pretty (e2 x) (x ++ "'") ++ ")"
  end.

Definition Pretty t (E : Term t) := pretty (E (fun _ ⇒ string)) "x".

Eval compute in Pretty three_the_hard_way.

    = "(((fun x => (fun x' => (x + x'))) N) N)"
```

However, it is not necessary to convert to first-order form to support many common operations on terms. For instance, we can implement substitution of terms for variables. The key insight here is to *tag variables with terms*, so that on encountering a variable, we can simply replace it by the term in its tag. We call this function initially on a term with exactly one free variable, tagged with the appropriate substitute. During recursion, new variables are added, but they are only tagged with their own term equivalents. Note that this function squash is parameterized over a specific *var* choice.

```
Fixpoint squash var t (e : term (term var) t) : term var t :=
  match e with
    | Var _ e1 ⇒ e1

    | Const n ⇒ Const n
```

```
  | Plus e1 e2 ⇒ Plus (squash e1) (squash e2)

  | Abs _ _ e1 ⇒ Abs (fun x ⇒ squash (e1 (Var x)))
  | App _ _ e1 e2 ⇒ App (squash e1) (squash e2)

  | Let _ _ e1 e2 ⇒ Let (squash e1) (fun x ⇒ squash (e2 (Var x)))
end.
```

To define the final substitution function over terms with single free variables, we define Term1, an analogue to Term that was defined before for closed terms.

```
Definition Term1 (t1 t2 : type) := ∀ var, var t1 → term var t2.
```

Substitution is defined by (1) instantiating a Term1 to tag variables with terms, and (2) applying the result to a specific term to be substituted. Note how the parameter *var* of squash is instantiated: the body of Subst is itself a polymorphic quantification over *var*, standing for a variable tag choice in the output term; and we use that input to compute a tag choice for the input term.

```
Definition Subst (t1 t2 : type) (E : Term1 t1 t2) (E' : Term t1)
  : Term t2 := fun var ⇒ squash (E (term var) (E' var)).

Eval compute in Subst (fun _ x ⇒ Plus (Var x) (Const 3))
  three_the_hard_way.

  = fun var : type → Type ⇒
    Plus
      (App
         (App
            (Abs
               (fun x : var Nat ⇒
                  Abs (fun y : var Nat ⇒ Plus (Var x) (Var y))))
            (Const 1)) (Const 2)) (Const 3)
```

One further development is that we can also implement a usual term denotation function, when we *tag variables with their denotations*.

```
Fixpoint termDenote t (e : term typeDenote t) : typeDenote t :=
  match e with
    | Var _ v ⇒ v

    | Const n ⇒ n
    | Plus e1 e2 ⇒ termDenote e1 + termDenote e2

    | Abs _ _ e1 ⇒ fun x ⇒ termDenote (e1 x)
```

```
    | App _ _ e1 e2 ⇒ (termDenote e1) (termDenote e2)

    | Let _ _ e1 e2 ⇒ termDenote (e2 (termDenote e1))
  end.

Definition TermDenote t (E : Term t) : typeDenote t :=
  termDenote (E typeDenote).

Eval compute in TermDenote three_the_hard_way.
```

$= 3$

To summarize, the PHOAS representation has all the expressive power of more standard first-order encodings, and a variety of translations are actually much more pleasant to implement than usual, thanks to the novel ability to tag variables with data.

17.2.2 Verifying Program Transformations

Let us now revisit the three example program transformations from Section 17.1. Each is easy to implement with PHOAS, and the last is substantially easier than with first-order representations.

First, we have the recursive identity function, following the same pattern as earlier, with a helper function, polymorphic in a tag choice; and a final function that instantiates the choice appropriately.

```
Fixpoint ident var t (e : term var t) : term var t :=
  match e with
    | Var _ x ⇒ Var x

    | Const n ⇒ Const n
    | Plus e1 e2 ⇒ Plus (ident e1) (ident e2)

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ ident (e1 x))
    | App _ _ e1 e2 ⇒ App (ident e1) (ident e2)

    | Let _ _ e1 e2 ⇒ Let (ident e1) (fun x ⇒ ident (e2 x))
  end.

Definition Ident t (E : Term t) : Term t := fun var ⇒
  ident (E var).
```

Proving correctness is both easier and harder than before, easier because we do not need to manipulate substitutions, and harder because we do the induction in an extra lemma about ident, to establish the correctness theorem for Ident.

```
Lemma identSound : ∀ t (e : term typeDenote t),
  termDenote (ident e) = termDenote e.
  induction e; pl.
Qed.

Theorem IdentSound : ∀ t (E : Term t),
  TermDenote (Ident E) = TermDenote E.
  intros; apply identSound.
Qed.
```

The translation of the constant-folding function and its proof work more or less the same way.

```
Fixpoint cfold var t (e : term var t) : term var t :=
  match e with
    | Plus e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
        match e1', e2' with
          | Const n1, Const n2 ⇒ Const (n1 + n2)
          | _, _ ⇒ Plus e1' e2'
        end

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ cfold (e1 x))
    | App _ _ e1 e2 ⇒ App (cfold e1) (cfold e2)

    | Let _ _ e1 e2 ⇒ Let (cfold e1) (fun x ⇒ cfold (e2 x))

    | e ⇒ e
  end.

Definition Cfold t (E : Term t) : Term t := fun var ⇒
  cfold (E var).

Lemma cfoldSound : ∀ t (e : term typeDenote t),
  termDenote (cfold e) = termDenote e.
  induction e; pl;
    repeat (match goal with
              | [ ⊢ context[match ?E with Var _ _ ⇒ _
                                        | _ ⇒ _ end] ] ⇒
                dep_destruct E
            end; pl).
Qed.

Theorem CfoldSound : ∀ t (E : Term t),
  TermDenote (Cfold E) = TermDenote E.
```

```
  intros; apply cfoldSound.
Qed.
```

Things get more interesting in the Let removal optimization. The recursive helper function adapts the key idea from earlier definitions of squash and Subst: tag variables with terms. We have a straightforward generalization of squash, where only the Let case has changed, to tag the new variable with the term it is bound to rather than just tagging the variable with itself as a term.

```
Fixpoint unlet var t (e : term (term var) t) : term var t :=
  match e with
    | Var _ e1 ⇒ e1

    | Const n ⇒ Const n
    | Plus e1 e2 ⇒ Plus (unlet e1) (unlet e2)

    | Abs _ _ e1 ⇒ Abs (fun x ⇒ unlet (e1 (Var x)))
    | App _ _ e1 e2 ⇒ App (unlet e1) (unlet e2)

    | Let _ _ e1 e2 ⇒ unlet (e2 (unlet e1))
  end.

Definition Unlet t (E : Term t) : Term t := fun var ⇒
  unlet (E (term var)).
```

We can test Unlet first on an uninteresting example, three_the_hard_way, which does not use Let.

```
Eval compute in Unlet three_the_hard_way.

    = fun var : type → Type ⇒
      App
        (App
           (Abs
              (fun x : var Nat ⇒
                Abs (fun x0 : var Nat ⇒ Plus (Var x) (Var x0))))
           (Const 1)) (Const 2)
```

Next, we try a more interesting example, with some extra Lets introduced in three_the_hard_way.

```
Definition three_a_harder_way : Term Nat := fun _ ⇒
  Let (Const 1) (fun x ⇒ Let (Const 2) (fun y ⇒
    App (App (add _) (Var x)) (Var y))).

Eval compute in Unlet three_a_harder_way.
```

```
= fun var : type → Type ⇒
  App
    (App
       (Abs
          (fun x : var Nat ⇒
           Abs (fun x0 : var Nat ⇒ Plus (Var x) (Var x0))))
       (Const 1)) (Const 2)
```

The output is the same as in the previous test, confirming that Unlet operates properly here.

Now we need to state a correctness theorem for Unlet, based on an inductively proved lemma about unlet. It is not obvious how to arrive at a proper induction principle for the lemma. The problem is that we want to relate two instantiations of the same Term, in a way where we know they share the same structure. Note that whereas Unlet is defined to consider all possible *var* choices in the output term, the correctness proof conveniently only depends on the case of *var* := typeDenote. Thus, one parallel instantiation will set *var* := typeDenote to take the denotation of the original term. The other parallel instantiation will set *var* := **term** typeDenote to perform the unlet transformation in the original term.

Here is a relation formalizing the idea that two terms are structurally the same, differing only by replacing the variable data of one with another isomorphic set of variable data in some possibly different type family:

```
Section wf.
  Variables var1 var2 : type → Type.
```

To formalize the tag isomorphism, we use lists of values with the following record type. Each entry has an object language type and an appropriate tag for that type, in each of the two tag families *var1* and *var2*.

```
Record varEntry := {
  Ty : type;
  First : var1 Ty;
  Second : var2 Ty
}.
```

Here is the inductive relation definition. An instance `wf` *G e1 e2* asserts that terms *e1* and *e2* are equivalent up to the variable tag isomorphism *G*. Note how the Var rule looks up an entry in *G*, and the Abs and Let rules include recursive `wf` invocations inside the scopes

of quantifiers to introduce parallel tag values to be considered as isomorphic.

```
Inductive wf : list varEntry → ∀ t, term var1 t → term var2 t
  → Prop :=
| WfVar : ∀ G t x x', In {| Ty := t; First := x; Second := x' |} G
  → wf G (Var x) (Var x')

| WfConst : ∀ G n, wf G (Const n) (Const n)

| WfPlus : ∀ G e1 e2 e1' e2', wf G e1 e1'
  → wf G e2 e2'
  → wf G (Plus e1 e2) (Plus e1' e2')

| WfAbs : ∀ G dom ran (e1 : _ dom → term _ ran) e1',
  (∀ x1 x2,
    wf ({| First := x1; Second := x2 |} :: G) (e1 x1) (e1' x2))
  → wf G (Abs e1) (Abs e1')

| WfApp : ∀ G dom ran (e1 : term _ (Func dom ran))
  (e2 : term _ dom) e1' e2',
  wf G e1 e1'
  → wf G e2 e2'
  → wf G (App e1 e2) (App e1' e2')

| WfLet : ∀ G t1 t2 e1 e1' (e2 : _ t1 → term _ t2) e2',
  wf G e1 e1'
  → (∀ x1 x2,
    wf ({| First := x1; Second := x2 |} :: G) (e2 x1) (e2' x2))
  → wf G (Let e1 e2) (Let e1' e2').
End wf.
```

We can state a well-formedness condition for closed terms: for any two choices of tag type families, the parallel instantiations belong to the `wf` relation, starting from an empty variable isomorphism.

```
Definition Wf t (E : Term t) := ∀ var1 var2,
  wf nil (E var1) (E var2).
```

After digesting the syntactic details of Wf, it is probably not hard to see that reasonable term encodings will satisfy it. For example,

```
Theorem three_the_hard_way_Wf : Wf three_the_hard_way.
  red; intros; repeat match goal with
                      | [ ⊢ wf _ _ _ ] ⇒ constructor; intros
```

```
                    end; intuition.
Qed.
```

Now we are ready to give a simple proof of correctness for unlet. First, we add one hint to apply a small variant of a standard library theorem connecting Forall, a higher-order predicate asserting that every element of a list satisfies some property, and In, the list membership predicate.

```
Hint Extern 1 ⇒ match goal with
                  | [ H1 : Forall _ _, H2 : In _ _ ⊢ _ ] ⇒
                    apply (Forall_In H1 _ H2)
                end.
```

The rest of the proof is about as automated as we could hope for.

```
Lemma unletSound : ∀ G t (e1 : term _ t) e2,
  wf G e1 e2
  → Forall (fun ve ⇒ termDenote (First ve) = Second ve) G
  → termDenote (unlet e1) = termDenote e2.
  induction 1; pl.
Qed.

Theorem UnletSound : ∀ t (E : Term t), Wf E
  → TermDenote (Unlet E) = TermDenote E.
  intros; eapply unletSound; eauto.
Qed.
```

With this example, it is not obvious that the PHOAS encoding is more tractable than dependent de Bruijn. Where the de Bruijn version had lift and its helper functions, here we have Wf and its auxiliary definitions. In practice, Wf is defined once per object language, whereas such operations as lift often need to operate differently for different examples, forcing new implementations for new transformations.

Readers may also have identified another objection: via Curry-Howard, wf proofs may be thought of as first-order encodings of term syntax. For instance, the In hypothesis of rule WfVar is equivalent to a **member** value. There is some merit to this objection. However, as the preceding proofs show, we are able to reason about transformations using first-order representation only for their inputs, not their outputs. Furthermore, explicit numbering of variables remains absent from the proofs.

Have we really avoided first-order reasoning about the output terms of translations? The answer depends on some subtle issues.

17.2.3 Establishing Term Well-Formedness

Can there be values of type Term t that are not well-formed according to Wf? We expect that Gallina satisfies key *parametricity* [38] properties, which indicate how polymorphic types may only be inhabited by specific values. We omit details of parametricity theorems here, but $\forall\ t\ (E :$ Term $t)$, Wf E follows the flavor of such theorems. One option would be to assert that fact as an axiom, "proving" that any output of any of the translations is well-formed. We could even prove the soundness of the theorem on paper metatheoretically, say, by considering some particular model of the Calculus of Inductive Constructions.

To be more cautious, we could prove Wf for every term that interests us, threading such proofs through all transformations. Here is an example exercise of that kind, for Unlet.

First, we prove that `wf` is monotone, in that a given instance continues to hold as we add new variable pairs to the variable isomorphism.

```
Hint Constructors wf.
Hint Extern 1 (In _ _) ⇒ simpl; tauto.
Hint Extern 1 (Forall _ _) ⇒
  eapply Forall_weaken; [ eassumption | simpl ].

Lemma wf_monotone : ∀ var1 var2 G t (e1 : term var1 t)
  (e2 : term var2 t),
  wf G e1 e2
  → ∀ G', Forall (fun x ⇒ In x G') G
    → wf G' e1 e2.
  induction 1; pl; auto 6.
Qed.

Hint Resolve wf_monotone Forall_In'.
```

Now we are ready to prove that unlet preserves any `wf` instance. The key invariant has to do with the parallel execution of unlet on two different *var* instantiations of a particular term. Since unlet uses **term** as the type of variable data, the variable isomorphism context G contains pairs of terms, which allows us to state the invariant that any pair of terms in the context is also related by `wf`.

```
Hint Extern 1 (wf _ _ _) ⇒ progress simpl.

Lemma unletWf : ∀ var1 var2 G t (e1 : term (term var1) t)
  (e2 : term (term var2) t),
  wf G e1 e2
  → ∀ G', Forall (fun ve ⇒ wf G' (First ve) (Second ve)) G
    → wf G' (unlet e1) (unlet e2).
```

```
  induction 1; pl; eauto 9.
Qed.
```

Repackaging unletWf into a theorem about Wf and Unlet is straightforward.

```
Theorem UnletWf : ∀ t (E : Term t), Wf E
  → Wf (Unlet E).
  red; intros; apply unletWf with nil; auto.
Qed.
```

This example demonstrates how we may need to use reasoning reminiscent of that associated with first-order representations, though the bookkeeping details are generally easier to manage, and bookkeeping theorems may generally be proved separately from the independently interesting theorems about program transformations.

17.2.4 A Few Additional Remarks

Higher-order encodings derive their strength from reuse of the metalanguage's binding constructs. As a result, we can write encoded terms so that they look very similar to their informal counterparts, without variable numbering schemes like those for de Bruijn indices. The example encodings demonstrated this fact, but modulo the clunkiness of explicit use of the constructors of **term**. After defining a few new Coq syntax notations, we can work with terms in an even more standard form.

```
Infix "->" := Func (right associativity, at level 52).

Notation "^" := Var.
Notation "#" := Const.
Infix "@" := App (left associativity, at level 50).
Infix "@+" := Plus (left associativity, at level 50).
Notation "\ x : t , e" := (Abs (dom := t) (fun x ⇒ e))
  (no associativity, at level 51, x at level 0).
Notation "[ e ]" := (fun _ ⇒ e).

Example Add : Term (Nat -> Nat -> Nat) :=
  [\x : Nat, \y : Nat, ^x @+ ^y].

Example Three_the_hard_way : Term Nat :=
  [Add _ @ #1 @ #2].

Eval compute in TermDenote Three_the_hard_way.

    = 3
```

End HigherOrder.

The PHOAS approach shines here because we are working with an object language that has an easy embedding into Coq. That is, there is a straightforward recursive function translating object terms into terms of Gallina. All Gallina programs terminate, so clearly we cannot hope to find such embeddings for Turing-complete languages; and non-Turing-complete languages may still require much more involved translations. I have some work [6] on modeling semantics of Turing-complete languages with PHOAS, but my impression is that many more advances are still to be made in this field, possibly with completely new term representations that have not yet been devised.