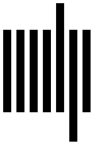
Certified Programming with Dependent Types

A Pragmatic Introduction to the Coq Proof Assistant

By: Adam Chlipala

Citation:

Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant
By: Adam Chlipala
DOI: 10.7551/mitpress/9153.001.0001
ISBN (electronic): 9780262317870
Publisher: The MIT Press
Published: 2022



The MIT Press

11 Generic Programming

Generic programming makes it possible to write functions that operate over different types of data. Parametric polymorphism in ML and Haskell is one of the simplest examples. ML-style module systems [22] and Haskell type classes [45] are more flexible cases. These language features are often not as powerful as we would like. For instance, while Haskell includes a type class classifying those types whose values can be pretty-printed, per-type pretty-printing is usually either implemented manually or implemented via a *deriving clause* [33], which triggers ad hoc code generation. Some clever encoding techniques have been used to achieve better within Haskell and other languages, but we can do *datatype-generic programming* much more cleanly with dependent types. Thanks to the expressive power of the Calculus of Inductive Constructions (CIC), we need no special language support.

Generic programming can often be very useful in Coq developments. In a proof assistant, there is the new possibility of generic proofs about generic programs.

11.1 Reifying Datatype Definitions

The key to generic programming with dependent types is *universe types*. This concept should not be confused with the idea of *universes* from the metatheory of CIC and related languages (see Chapter 12). Rather, the idea of universe types is to define inductive types that provide *syntactic representations* of Coq types. We cannot directly write CIC programs that do case analysis on types, but we can case-analyze on reified syntactic versions of those types.

Thus, to begin, we define a syntactic representation of some class of datatypes. The running example in this chapter involves basic algebraic datatypes of the kind found in ML and Haskell, but without enhancements like type parameters and mutually recursive definitions.

The first step is to define a representation for constructors of the datatypes. We use the `Record` command as a shorthand for defining an inductive type with a single constructor, plus projection functions for pulling out any of the named arguments to that constructor.

```
Record constructor : Type := Con {
  nonrecursive : Type;
  recursive : nat
}.
```

The idea is that a constructor represented as Con T n has n arguments of the type that we are defining. Additionally, all the other, nonrecursive arguments can be encoded in the type T. When there are no nonrecursive arguments, T can be **unit**. When there are two nonrecursive arguments, of types A and B, T can be $A \times B$. We can generalize to any number of arguments via tupling.

With this definition, it is easy to define a datatype representation in terms of lists of constructors. The intended meaning is that the datatype came from an inductive definition including exactly the constructors in the list.

```
Definition datatype := list constructor.
```

Here are a few example encodings for some common types from the Coq standard library. While the syntax type does not support type parameters directly, we can implement them at the metalevel, via functions from types to datatypes.

```
Definition Empty_set_dt : datatype := nil.
Definition unit_dt : datatype := Con unit 0 :: nil.
Definition bool_dt : datatype := Con unit 0 :: Con unit 0 :: nil.
Definition nat_dt : datatype := Con unit 0 :: Con unit 1 :: nil.
Definition list_dt (A : Type) : datatype :=
  Con unit 0 :: Con A 1 :: nil.
```

The type **Empty_set** has no constructors, so its representation is the empty list. The type **unit** has one constructor with no arguments, so its one reified constructor indicates no nonrecursive data and 0 recursive arguments. The representation for **bool** just duplicates this single argumentless constructor. We get from **bool** to **nat** by changing one of the constructors to indicate 1 recursive argument. We get from **nat** to **list** by adding a nonrecursive argument of a parameter type A.

As a further example, we can do the same encoding for a generic binary tree type.

```
Section tree.
```

```
  Variable A : Type.

  Inductive tree : Type :=
  | Leaf : A → tree
  | Node : tree → tree → tree.
End tree.

Definition tree_dt (A : Type) : datatype :=
  Con A 0 :: Con unit 2 :: nil.
```

Each datatype representation stands for a family of inductive types. For a specific real datatype and a reputed representation for it, it is useful to define a type of *evidence* that the datatype is compatible with the encoding.

```
Section denote.
  Variable T : Type.
```

This variable stands for the concrete datatype of interest.

```
  Definition constructorDenote (c : constructor) :=
    nonrecursive c → ilist T (recursive c) → T.
```

We write that a constructor is represented as a function returning a T. Such a function takes two arguments, which pack together the nonrecursive and recursive arguments of the constructor. We represent a tuple of all recursive arguments using the length-indexed list type **ilist** (see Chapter 8).

```
  Definition datatypeDenote := hlist constructorDenote.
```

Finally, the evidence for type T is a heterogeneous list, including a constructor denotation for every constructor encoding in a datatype encoding. Recall that since we are inside a section binding T as a variable, constructorDenote is automatically parameterized by T.

```
End denote.
```

Some example pieces of evidence should help clarify the convention. First, we define a helpful notation for constructor denotations. The ASCII ~> from the notation is rendered as ⇝.

```
Notation "[ v , r ~> x ]" :=
  ((fun v r ⇒ x) : constructorDenote _ (Con _ _)).

Definition Empty_set_den
  : datatypeDenote Empty_set Empty_set_dt := HNil.
Definition unit_den : datatypeDenote unit unit_dt :=
  [_, _ ⇝ tt] ::: HNil.
Definition bool_den : datatypeDenote bool bool_dt :=
  [_, _ ⇝ true] ::: [_, _ ⇝ false] ::: HNil.
Definition nat_den : datatypeDenote nat nat_dt :=
```

```
  [_, _ ⇝ O] ::: [_, r ⇝ S (hd r)] ::: HNil.
Definition list_den (A : Type) : datatypeDenote (list A) (list_dt A) :=
  [_, _ ⇝ nil] ::: [x, r ⇝ x :: hd r] ::: HNil.
Definition tree_den (A : Type)
  : datatypeDenote (tree A) (tree_dt A) :=
  [v, _ ⇝ Leaf v] ::: [_, r ⇝ Node (hd r) (hd (tl r))] ::: HNil.
```

Recall that the hd and tl calls operate on richly typed lists, where type indices indicate the lengths of lists, guaranteeing the safety of operations like hd. The type annotation attached to each definition provides enough information for Coq to infer list lengths at appropriate points.

11.2 Recursive Definitions

We built these encodings of datatypes to help us write datatype-generic recursive functions. To do so, we want a reified representation of a *recursion scheme* for each type, similar to the *T_rect* principle generated automatically for an inductive definition of T. A clever reuse of datatypeDenote yields a short definition.

```
Definition fixDenote (T : Type) (dt : datatype) :=
  ∀ (R : Type), datatypeDenote R dt → (T → R).
```

The idea of a recursion scheme is parameterized by a type and a reputed encoding of it. The principle itself is polymorphic in a type R, which is the return type of the recursive function that we mean to write. The next argument is a heterogeneous list of one case of the recursive function definition for each datatype constructor. The datatypeDenote function has just the right definition to express the type we need; a set of function cases is just like an alternative set of constructors where we replace the original type T with the function result type R. Given such a reified definition, a fixDenote invocation returns a function from T to R, which is just what we wanted.

We are ready to write some example functions using one new function from the DepList library (see book source).

```
Check hmake.

hmake
    : ∀ (A : Type) (B : A → Type),
      (∀ x : A, B x) → ∀ ls : list A, hlist B ls
```

The function hmake is a kind of map alternative that goes from a regular **list** to an **hlist**. We can use it to define a generic size function that counts the number of constructors used to build a value in a datatype.

```
Definition size T dt (fx : fixDenote T dt) : T → nat :=
  fx nat (hmake (B := constructorDenote nat)
    (fun _ _ r ⇒ foldr plus 1 r) dt).
```

The definition is parameterized over a recursion scheme *fx*. We instantiate *fx* by passing it the function result type and a set of function cases, building the latter with hmake. The function argument to hmake takes three arguments: the representation of a constructor, its nonrecursive arguments, and the results of recursive calls on all its recursive arguments. We only need the recursive call results here, so we call them *r* and bind the other two inputs with wildcards. The actual case body is simple: we add together the recursive call results and increment the result by one (to account for the current constructor). This foldr function is an **ilist**-specific version defined in the DepList module.

It is instructive to build fixDenote values for the example types and see what specialized size functions result from them.

```
Definition Empty_set_fix : fixDenote Empty_set Empty_set_dt :=
  fun R _ emp ⇒ match emp with end.
Eval compute in size Empty_set_fix.

    = fun emp : Empty_set ⇒ match emp return nat with
                            end
    : Empty_set → nat
```

CIC's standard computation rules suffice to normalize the generic size function specialization to exactly what we would have written manually.

```
Definition unit_fix : fixDenote unit unit_dt :=
  fun R cases _ ⇒ (hhd cases) tt INil.
Eval compute in size unit_fix.

    = fun _ : unit ⇒ 1
    : unit → nat
```

Again normalization gives us the natural function definition. We see this pattern repeated for the other example types.

```
Definition bool_fix : fixDenote bool bool_dt :=
  fun R cases b ⇒ if b
    then (hhd cases) tt INil
    else (hhd (htl cases)) tt INil.
Eval compute in size bool_fix.
```

```
    = fun b : bool ⇒ if b then 1 else 1
    : bool → nat
```

```
Definition nat_fix : fixDenote nat nat_dt :=
  fun R cases ⇒ fix F (n : nat) : R :=
    match n with
      | O ⇒ (hhd cases) tt INil
      | S n' ⇒ (hhd (htl cases)) tt (ICons (F n') INil)
    end.
```

To look at the size function for **nat**, it is useful to avoid full computation, so that the recursive definition of addition is not expanded inline. We can accomplish this with proper flags for the cbv reduction strategy.

```
Eval cbv beta iota delta -[plus] in size nat_fix.
```

```
    = fix F (n : nat) : nat := match n with
                               | 0 ⇒ 1
                               | S n' ⇒ F n' + 1
                               end
    : nat → nat
```

```
Definition list_fix (A : Type) : fixDenote (list A) (list_dt A) :=
  fun R cases ⇒ fix F (ls : list A) : R :=
    match ls with
      | nil ⇒ (hhd cases) tt INil
      | x :: ls' ⇒ (hhd (htl cases)) x (ICons (F ls') INil)
    end.
Eval cbv beta iota delta -[plus] in fun A ⇒ size (@list_fix A).
```

```
    = fun A : Type ⇒
      fix F (ls : list A) : nat :=
        match ls with
        | nil ⇒ 1
        | _ :: ls' ⇒ F ls' + 1
        end
    : ∀ A : Type, list A → nat
```

```
Definition tree_fix (A : Type) : fixDenote (tree A) (tree_dt A) :=
  fun R cases ⇒ fix F (t : tree A) : R :=
    match t with
      | Leaf x ⇒ (hhd cases) x INil
      | Node t1 t2 ⇒
```

```
      (hhd (htl cases)) tt (ICons (F t1) (ICons (F t2) INil))
  end.
Eval cbv beta iota delta -[plus] in fun A ⇒ size (@tree_fix A).

    = fun A : Type ⇒
      fix F (t : tree A) : nat :=
        match t with
        | Leaf _ ⇒ 1
        | Node t1 t2 ⇒ F t1 + (F t2 + 1)
        end
    : ∀ A : Type, tree A → nat
```

As the examples show, even recursive datatypes are mapped to normal-looking size functions.

11.2.1 Pretty-Printing

It is useful to do generic pretty-printing of datatype values, rendering them as human-readable strings. To do so, we need a bit of metadata for each constructor, specifically the name to print for the constructor and the function to use to render its nonrecursive arguments. Everything else can be done generically.

```
Record print_constructor (c : constructor) : Type := PI {
  printName : string;
  printNonrec : nonrecursive c → string
}.
```

It is useful to define a shorthand for applying the constructor PI. By applying it explicitly to an unknown application of the constructor Con, we help type inference work.

```
Notation "^" := (PI (Con _ _)).
```

As in earlier examples, we define the type of metadata for a datatype to be a heterogeneous list type collecting metadata for each constructor.

```
Definition print_datatype := hlist print_constructor.
```

For some string manipulation, we import the notations associated with strings.

```
Local Open Scope string_scope.
```

Now it is easy to implement the generic printer, using another function from DepList.

```
Check hmap.

  hmap
```

```
    : ∀ (A : Type) (B1 B2 : A → Type),
        (∀ x : A, B1 x → B2 x) →
        ∀ ls : list A, hlist B1 ls → hlist B2 ls

Definition print T dt (pr : print_datatype dt) (fx : fixDenote T dt)
  : T → string :=
  fx string (hmap (B1 := print_constructor)
    (B2 := constructorDenote string)
    (fun _ pc x r ⇒ printName pc ++ "(" ++ printNonrec pc x
      ++ foldr (fun s acc ⇒ ", " ++ s ++ acc) ")" r) pr).
```

Some simple tests establish that print gets the job done.

```
Eval compute in print HNil Empty_set_fix.

    = fun emp : Empty_set ⇒ match emp return string with
                            end
    : Empty_set → string

Eval compute in print (^ "tt" (fun _ ⇒ "") ::: HNil) unit_fix.

    = fun _ : unit ⇒ "tt()"
    : unit → string

Eval compute in print (^ "true" (fun _ ⇒ "")
  ::: ^ "false" (fun _ ⇒ "")
  ::: HNil) bool_fix.

    = fun b : bool ⇒ if b then "true()" else "false()"
    : bool → string

Definition print_nat := print (^ "O" (fun _ ⇒ "")
  ::: ^ "S" (fun _ ⇒ "")
  ::: HNil) nat_fix.
Eval cbv beta iota delta -[append] in print_nat.

    = fix F (n : nat) : string :=
        match n with
        | 0%nat ⇒ "O" ++ "(" ++ "" ++ ")"
        | S n' ⇒ "S" ++ "(" ++ "" ++ ", " ++ F n' ++ ")"
        end
    : nat → string

Eval simpl in print_nat 0.

    = "O()"
    : string
```

```
Eval simpl in print_nat 1.

     = "S(, O())"
     : string

Eval simpl in print_nat 2.

     = "S(, S(, O()))"
     : string

Eval cbv beta iota delta -[append] in fun A (pr : A → string) ⇒
  print (^ "nil" (fun _ ⇒ "")
  ::: ^ "cons" pr
  ::: HNil) (@list_fix A).

     = fun (A : Type) (pr : A → string) ⇒
       fix F (ls : list A) : string :=
         match ls with
         | nil ⇒ "nil" ++ "(" ++ "" ++ ")"
         | x :: ls' ⇒
           "cons" ++ "(" ++ pr x ++ ", " ++ F ls' ++ ")"
         end
     : ∀ A : Type, (A → string) → list A → string

Eval cbv beta iota delta -[append] in fun A (pr : A → string) ⇒
  print (^ "Leaf" pr
  ::: ^ "Node" (fun _ ⇒ "")
  ::: HNil) (@tree_fix A).

     = fun (A : Type) (pr : A → string) ⇒
       fix F (t : tree A) : string :=
         match t with
         | Leaf x ⇒ "Leaf" ++ "(" ++ pr x ++ ")"
         | Node t1 t2 ⇒ "Node" ++ "(" ++ "" ++ ", " ++ F t1
           ++ ", " ++ F t2 ++ ")"
         end
     : ∀ A : Type, (A → string) → tree A → string
```

Some of these simplified terms seem overly complex because we have turned off simplification of calls to append, which is what uses of the ++ operator desugar to. Selective ++ simplification would combine adjacent string literals, yielding more or less the code we would write manually to implement this printing scheme.

11.2.2 Mapping

By this point, we have developed enough machinery to define a generic function similar to the list map function.

```
Definition map T dt (dd : datatypeDenote T dt) (fx : fixDenote T dt)
  (f : T → T) : T → T :=
  fx T (hmap (B1 := constructorDenote T)
    (B2 := constructorDenote T)
    (fun _ c x r ⇒ f (c x r)) dd).
```

```
Eval compute in map Empty_set_den Empty_set_fix.
```

```
= fun (_ : Empty_set → Empty_set) (emp : Empty_set) ⇒
  match emp return Empty_set with
  end
: (Empty_set → Empty_set) → Empty_set → Empty_set
```

```
Eval compute in map unit_den unit_fix.
```

```
= fun (f : unit → unit) (_ : unit) ⇒ f tt
: (unit → unit) → unit → unit
```

```
Eval compute in map bool_den bool_fix.
```

```
= fun (f : bool → bool) (b : bool) ⇒
  if b then f true else f false
: (bool → bool) → bool → bool
```

```
Eval compute in map nat_den nat_fix.
```

```
= fun f : nat → nat ⇒
  fix F (n : nat) : nat :=
    match n with
    | 0%nat ⇒ f 0%nat
    | S n' ⇒ f (S (F n'))
    end
: (nat → nat) → nat → nat
```

```
Eval compute in fun A ⇒ map (list_den A) (@list_fix A).
```

```
= fun (A : Type) (f : list A → list A) ⇒
  fix F (ls : list A) : list A :=
    match ls with
    | nil ⇒ f nil
    | x :: ls' ⇒ f (x :: F ls')
    end
: ∀ A : Type, (list A → list A) → list A → list A
```

```
Eval compute in fun A ⇒ map (tree_den A) (@tree_fix A).

    = fun (A : Type) (f : tree A → tree A) ⇒
      fix F (t : tree A) : tree A :=
        match t with
        | Leaf x ⇒ f (Leaf x)
        | Node t1 t2 ⇒ f (Node (F t1) (F t2))
        end
    : ∀ A : Type, (tree A → tree A) → tree A → tree A
```

These map functions are just as easy to use as those we write by hand. Readers may want to try figuring out the input-output pattern that map_nat S displays in the following examples.

```
Definition map_nat := map nat_den nat_fix.
Eval simpl in map_nat S 0.

    = 1%nat
    : nat

Eval simpl in map_nat S 1.

    = 3%nat
    : nat

Eval simpl in map_nat S 2.

    = 5%nat
    : nat
```

We get map_nat S $n = 2 \times n + 1$, because the mapping process adds an extra S at every level of the inductive tree that defines a natural, including at the last level, the O constructor.

11.3 Proving Theorems about Recursive Definitions

We would like to be able to prove theorems about generic functions. To do so, we need to establish additional well-formedness properties that must hold of pieces of evidence.

```
Section ok.
  Variable T : Type.
  Variable dt : datatype.

  Variable dd : datatypeDenote T dt.
  Variable fx : fixDenote T dt.
```

First, we characterize when a piece of evidence about a datatype is acceptable. The basic idea is that the type T should really be an inductive type with the definition given by *dd*. Semantically, inductive types are characterized by the ability to do induction on them. Therefore, we require that the usual induction principle be true with respect to the constructors given in the encoding *dd*.

```
Definition datatypeDenoteOk :=
  ∀ P : T → Prop,
    (∀ c (m : member c dt) (x : nonrecursive c)
       (r : ilist T (recursive c)),
       (∀ i : fin (recursive c), P (get r i))
       → P ((hget dd m) x r))
    → ∀ v, P v.
```

This definition can take a while to understand. The quantifier over m : **member** c dt is considering each constructor in turn; as in normal induction principles, each constructor has an associated proof case. The expression hget dd m then names the constructor we have selected. After binding m, we quantify over all possible arguments (encoded with x and r) to the constructor that m selects. Within each specific case, we quantify further over i : **fin** (recursive c) to consider all of the induction hypotheses, one for each recursive argument of the current constructor.

We have completed half the process of defining side conditions. The other half comes in characterizing when a recursion scheme fx is valid. The natural condition is that fx behaves appropriately when applied to any constructor application.

```
Definition fixDenoteOk :=
  ∀ (R : Type) (cases : datatypeDenote R dt)
    c (m : member c dt)
    (x : nonrecursive c) (r : ilist T (recursive c)),
    fx cases ((hget dd m) x r)
    = (hget cases m) x (imap (fx cases) r).
```

As for datatypeDenoteOk, we consider all constructors and all possible arguments to them by quantifying over m, x, and r. The left side of the equality that follows shows a call to the recursive function on the specific constructor application that we selected. The right side shows an application of the function case associated with constructor m, applied to the nonrecursive arguments and to appropriate recursive calls on the recursive arguments.

```
End ok.
```

We are now ready to prove that the size function we defined earlier always returns positive results. First, we establish a simple lemma.

```
Lemma foldr_plus : ∀ n (ils : ilist nat n),
  foldr plus 1 ils > 0.
  induction ils; crush.
Qed.

Theorem size_positive : ∀ T dt
  (dd : datatypeDenote T dt) (fx : fixDenote T dt)
  (dok : datatypeDenoteOk dd) (fok : fixDenoteOk dd fx)
  (v : T),
  size fx v > 0.
  unfold size; intros.

  ============================
   fx nat
     (hmake
        (fun (x : constructor) (_ : nonrecursive x)
           (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt) v > 0
```

The goal is an inequality over a particular call to size, with its definition expanded. How can we proceed here? We cannot use induction directly because there is no way for Coq to know that T is an inductive type. Instead, we need to use the induction principle encoded in the hypothesis *dok* of type datatypeDenoteOk *dd*. Let us try applying it directly.

```
  apply dok.
Error: Impossible to unify "datatypeDenoteOk dd" with
 "fx nat
    (hmake
       (fun (x : constructor) (_ : nonrecursive x)
          (r : ilist nat (recursive x)) =>
             foldr plus 1%nat r) dt) v > 0".
```

Matching the type of *dok* with the type of the conclusion requires more than simple first-order unification, so apply is not up to the challenge. We can use the pattern tactic to get the goal into a form that makes it apparent exactly what the induction hypothesis is.

```
  pattern v.

  ============================
   (fun t : T ⇒
    fx nat
```

```
    (hmake
       (fun (x : constructor) (_ : nonrecursive x)
          (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt) t
  > 0) v
```

```
apply dok; crush.
```

```
H : ∀ i : fin (recursive c),
    fx nat
      (hmake
         (fun (x : constructor) (_ : nonrecursive x)
            (r : ilist nat (recursive x)) ⇒ foldr plus 1%nat r) dt)
      (get r i) > 0
============================
 hget
   (hmake
      (fun (x0 : constructor) (_ : nonrecursive x0)
         (r0 : ilist nat (recursive x0)) ⇒
         foldr plus 1%nat r0) dt) m x
   (imap
      (fx nat
         (hmake
            (fun (x0 : constructor) (_ : nonrecursive x0)
               (r0 : ilist nat (recursive x0)) ⇒
             foldr plus 1%nat r0) dt)) r) > 0
```

An induction hypothesis H is generated, but we do not need it for this example. We can simplify the goal using a library theorem about the composition of hget and hmake.

```
rewrite hget_hmake.
```

```
============================
 foldr plus 1%nat
   (imap
      (fx nat
         (hmake
            (fun (x0 : constructor) (_ : nonrecursive x0)
               (r0 : ilist nat (recursive x0)) ⇒
             foldr plus 1%nat r0) dt)) r) > 0
```

The lemma we proved earlier finishes the proof.

```
apply foldr_plus.
```

Using hints, we can redo this proof in automated form.

```
  Restart.

  Hint Rewrite hget_hmake.
  Hint Resolve foldr_plus.

  unfold size; intros; pattern v; apply dok; crush.
Qed.
```

In this example, we only needed to use induction degenerately as case analysis. A more involved theorem about map may only be proved using induction hypotheses. I give its proof only in unautomated form and leave effective automation as an exercise for the reader.

In particular, it ought to be the case that generic map applied to an identity function is itself an identity function.

```
Theorem map_id : ∀ T dt
  (dd : datatypeDenote T dt) (fx : fixDenote T dt)
  (dok : datatypeDenoteOk dd) (fok : fixDenoteOk dd fx)
  (v : T),
  map dd fx (fun x ⇒ x) v = v.
```

Let us begin as we did in the last theorem, after adding another useful library equality as a hint.

```
  Hint Rewrite hget_hmap.

  unfold map; intros; pattern v; apply dok; crush.

  H : ∀ i : fin (recursive c),
      fx T
        (hmap
           (fun (x : constructor) (c : constructorDenote T x)
              (x0 : nonrecursive x) (r : ilist T (recursive x)) ⇒
            c x0 r) dd) (get r i) = get r i
  ============================
   hget dd m x
     (imap
        (fx T
           (hmap
              (fun (x0 : constructor) (c0 : constructorDenote T x0)
                 (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
               c0 x1 r0) dd)) r) = hget dd m x r
```

The goal is an equality whose two sides begin with the same function call and initial arguments. We believe that the remaining arguments are in fact equal as well, and the f_equal tactic applies this reasoning step formally.

```
  f_equal.
```

```
============================
 imap
   (fx T
      (hmap
         (fun (x0 : constructor) (c0 : constructorDenote T x0)
            (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
          c0 x1 r0) dd)) r = r
```

At this point, it is helpful to proceed by an inner induction on the heterogeneous list r of recursive call results. We could arrive at a cleaner proof by breaking this step out into an explicit lemma, but here we do the induction inline to save space.

```
induction r; crush.
```

The base case is discharged automatically. In the following inductive case, H is the outer IH (for induction over T values) and IHr is the inner IH (for induction over the recursive arguments).

```
H : ∀ i : fin (S n),
    fx T
      (hmap
         (fun (x : constructor) (c : constructorDenote T x)
            (x0 : nonrecursive x) (r : ilist T (recursive x)) ⇒
          c x0 r) dd)
      (match i in (fin n') return ((fin (pred n') → T) → T) with
       | First n ⇒ fun _ : fin n → T ⇒ a
       | Next n idx' ⇒ fun get_ls' : fin n → T ⇒ get_ls' idx'
       end (get r)) =
    match i in (fin n') return ((fin (pred n') → T) → T) with
    | First n ⇒ fun _ : fin n → T ⇒ a
    | Next n idx' ⇒ fun get_ls' : fin n → T ⇒ get_ls' idx'
    end (get r)
IHr : (∀ i : fin n,
       fx T
         (hmap
            (fun (x : constructor) (c : constructorDenote T x)
               (x0 : nonrecursive x) (r : ilist T (recursive x)) ⇒
             c x0 r) dd) (get r i) = get r i) →
      imap
        (fx T
           (hmap
              (fun (x : constructor) (c : constructorDenote T x)
```

```
                    (x0 : nonrecursive x) (r : ilist T (recursive x)) ⇒
                  c x0 r) dd)) r = r
============================
 ICons
   (fx T
      (hmap
         (fun (x0 : constructor) (c0 : constructorDenote T x0)
            (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
          c0 x1 r0) dd) a)
   (imap
      (fx T
         (hmap
            (fun (x0 : constructor) (c0 : constructorDenote T x0)
               (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
             c0 x1 r0) dd)) r) = ICons a r
```

We see another opportunity to apply `f_equal`, this time to split the goal into two different equalities over corresponding arguments. After that, the form of the first goal matches the outer induction hypothesis H, when we give type inference some help by specifying the right quantifier instantiation.

```
f_equal.
apply (H First).
```

```
============================
 imap
   (fx T
      (hmap
         (fun (x0 : constructor) (c0 : constructorDenote T x0)
            (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
          c0 x1 r0) dd)) r = r
```

Now the goal matches the inner IH *IHr*.

```
apply IHr; crush.
```

```
i : fin n
============================
 fx T
   (hmap
      (fun (x0 : constructor) (c0 : constructorDenote T x0)
         (x1 : nonrecursive x0) (r0 : ilist T (recursive x0)) ⇒
       c0 x1 r0) dd) (get r i) = get r i
```

We can finish the proof by applying the outer IH again, specialized to a different **fin** value.

```
  apply (H (Next i)).
Qed.
```

The proof involves complex subgoals, but few steps are required and the work may be reused across a variety of datatypes.

This book was set in 10/13 Lucida Bright by the author using LaTeX 2_ε. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Chlipala, Adam, 1981–

Certified programming with dependent types : a pragmatic introduction to the Coq proof assistant / Adam Chlipala.

p. cm

Includes bibliographical references and index.

ISBN 978-0-262-02665-9 (hardcover : alk. paper)

1. Automatic theorem proving–Computer programs. 2. Computer programming. 3. Coq (Electronic resource) I. Title.

QA76.9.A96C45 2013

005.1—dc23

2013012837

10 9 8 7 6 5 4 3 2 1