This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

9 Dependent Data Structures

The red-black tree example in the last chapter illustrated how dependent types enable static enforcement of data structure invariants. To find interesting uses of dependent data structures, however, we need not look to the favorite examples of data structures and algorithms textbooks. More basic examples like length-indexed and heterogeneous lists come up again and again as the building blocks of dependent programs. There is a surprisingly large design space for this class of data structure, and this chapter explores it.

9.1 More Length-Indexed Lists

We begin with a deeper look at length-indexed lists.

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).
```

We might like to have a certified function for selecting an element of an **ilist** by position. We could do this using subset types and explicit manipulation of proofs, but dependent types let us do it more directly. It is helpful to define a type family **fin**, where **fin** n is isomorphic to $\{m : \mathbf{nat} \mid m < n\}$. The type family name stands for *finite*.

```
  Inductive fin : nat → Set :=
  | First : ∀ n, fin (S n)
  | Next : ∀ n, fin n → fin (S n).
```

An instance of **fin** is essentially a more richly typed copy of a prefix of the natural numbers. Every element is a First iterated through applying Next a number of times that indicates which number is being selected.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

For instance, the three values of type **fin** 3 are First 2, Next (First 1), and Next (Next (First 0)).

Now it is easy to pick a Prop-free type for a selection function. As usual, the first implementation attempt will not satisfy the type checker, and we will attack the deficiencies one at a time.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒ ?
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx with
        | First _ ⇒ x
        | Next _ idx' ⇒ get ls' idx'
      end
  end.
```

We apply the usual wisdom of delaying arguments in Fixpoints so that they may be included in return clauses. This still leaves us with a quandary in each of the match cases. First, we need to figure out how to take advantage of the contradiction in the Nil case. Every **fin** has a type of the form S n, which cannot unify with the O value that we learn for n in the Nil case. The solution we adopt is another case of match-within-return, with the return clause chosen carefully so that it returns the proper type A in case the **fin** index is O, which we know is true here, and so that it returns an easy-to-inhabit type **unit** in the remaining, impossible cases, which nonetheless appear explicitly in the body of the match.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                     | O ⇒ A
                                     | S _ ⇒ unit
                                   end) with
        | First _ ⇒ tt
        | Next _ _ ⇒ tt
      end
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx with
        | First _ ⇒ x
        | Next _ idx' ⇒ get ls' idx'
      end
  end.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

Now the first match case type-checks, and we see that the problem with the Cons case is that the pattern-bound variable *idx'* does not have an apparent type compatible with *ls'*. In fact, the error message Coq gives for this exact code can be confusing, thanks to an overenthusiastic type inference heuristic. We are told that the Nil case body has type match X with | O ⇒ A | S _ ⇒ **unit** end for a unification variable X, while it is expected to have type A. We can see that setting X to O resolves the conflict, but Coq is not yet smart enough to do this unification automatically. Repeating the function's type in a return annotation, used with an in annotation, leads us to a more informative error message saying that *idx'* has type **fin** *n1*, while it is expected to have type **fin** *n0*, where *n0* is bound by the Cons pattern and *n1* by the Next pattern. As the code is written, nothing forces these two natural numbers to be equal, though we know intuitively that they must be.

We need to use match annotations to make the relation explicit. Unfortunately, the usual trick of postponing argument binding will not help here. We need to match on both *ls* and *idx*; one or the other must be matched first. To get around this, we apply the convoy pattern (see Chapter 8). This application is a little more clever than those we saw before; we use the natural number predecessor function pred to express the relation between the types of these variables.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ _ ⇒ tt
      end
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx in fin n' return ilist (pred n') → A with
        | First _ ⇒ fun _ ⇒ x
        | Next _ idx' ⇒ fun ls' ⇒ get ls' idx'
      end ls'
  end.
```

There is just one problem left with this implementation. Though we know that the local *ls'* in the Next case is equal to the original *ls'*, the type checker is not satisfied that the recursive call to get does not

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

introduce nontermination. We solve the problem by convoy-binding the partial application of get to *ls'* rather than *ls'* by itself.

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ _ ⇒ tt
      end
    | Cons _ x ls' ⇒ fun idx ⇒
      match idx in fin n' return (fin (pred n') → A) → A with
        | First _ ⇒ fun _ ⇒ x
        | Next _ idx' ⇒ fun get_ls' ⇒ get_ls' idx'
      end (get ls')
  end.
End ilist.

Implicit Arguments Nil [A].
Implicit Arguments First [n].
```

A few examples show how to make use of these definitions.

```
Check Cons 0 (Cons 1 (Cons 2 Nil)).

  Cons 0 (Cons 1 (Cons 2 Nil))
     : ilist nat 3

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.

     = 0
     : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).

     = 1
     : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).

     = 2
     : nat
```

The get function is also quite easy to reason about. We look at a short example about an analogue to the list map function.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
Section ilist_map.
  Variables A B : Set.
  Variable f : A → B.

  Fixpoint imap n (ls : ilist A n) : ilist B n :=
    match ls with
      | Nil ⇒ Nil
      | Cons _ x ls' ⇒ Cons (f x) (imap ls')
    end.
```

It is easy to prove that get distributes over imap calls.

```
  Theorem get_imap : ∀ n (idx : fin n) (ls : ilist A n),
    get (imap ls) idx = f (get ls idx).
    induction ls; dep_destruct idx; crush.
  Qed.
End ilist_map.
```

The only tricky bit is remembering to use the *dep_destruct* tactic in place of plain `destruct` when faced with a baffling tactic error message.

9.2 Heterogeneous Lists

Programmers who move to statically typed functional languages from scripting languages often complain about the requirement that every element of a list have the same type. With more elaborate type systems, we can partially lift this requirement. We can index a list type with a type-level list that explains what type each element of the list should have. This has been done in a variety of ways in Haskell using type classes, and it can be done much more cleanly and directly in Coq.

```
Section hlist.
  Variable A : Type.
  Variable B : A → Type.
```

We parameterize the heterogeneous lists by a type A and an A-indexed type B.

```
  Inductive hlist : list A → Type :=
  | HNil : hlist nil
  | HCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).
```

We can implement a variant of the get function for **hlist**s. To get the dependent typing to work out, we need to index the element selectors (in type family **member**) by the types of data that they point to.

```
  Variable elm : A.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
Inductive member : list A → Type :=
| HFirst : ∀ ls, member (elm :: ls)
| HNext : ∀ x ls, member ls → member (x :: ls).
```

Because the element *elm* that we are searching for in a list does not change across the constructors of **member**, we simplify the definitions by making *elm* a local variable. In the definition of **member**, we say that *elm* is found in any list that begins with *elm*, and if removing the first element of a list leaves *elm* present, then *elm* is present in the original list, too. The form looks much like a predicate for list membership, but we purposely define **member** in Type so that we may decompose its values to guide computations.

We can use **member** to adapt the definition of get to **hlist**s. The same basic match techniques apply. In the HCons case, we form a two-element convoy, passing both the data element *x* and the recursor for the sublist *mls'* to the result of the inner match. We did not need to do that in get's definition because the types of list elements were not dependent there.

```
Fixpoint hget ls (mls : hlist ls) : member ls → B elm :=
  match mls with
    | HNil ⇒ fun mem ⇒
      match mem in member ls'
        return (match ls' with
                  | nil ⇒ B elm
                  | _ :: _ ⇒ unit
                end) with
        | HFirst _ ⇒ tt
        | HNext _ _ _ ⇒ tt
      end
    | HCons _ _ x mls' ⇒ fun mem ⇒
      match mem in member ls'
        return (match ls' with
                  | nil ⇒ Empty_set
                  | x' :: ls'' ⇒
                    B x' → (member ls'' → B elm) → B elm
                end) with
        | HFirst _ ⇒ fun x _ ⇒ x
        | HNext _ _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
      end x (hget mls')
  end.
End hlist.

Implicit Arguments HNil [A B].
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
Implicit Arguments HCons [A B x ls].

Implicit Arguments HFirst [A elm ls].
Implicit Arguments HNext [A elm x ls].
```

By putting the parameters A and B in Type, we enable more kinds of polymorphism than in mainstream functional languages. For instance, one use of **hlist** is for the simple heterogeneous lists referred to earlier.

```
Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set ⇒ T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

    = 5
    : (fun T : Set ⇒ T) nat

Eval simpl in hget someValues (HNext HFirst).

    = true
    : (fun T : Set ⇒ T) bool
```

We can also build indexed lists of pairs in this way.

```
Example somePairs : hlist (fun T : Set ⇒ T × T)%type someTypes :=
  HCons (1, 2) (HCons (true, false) HNil).
```

There are many other useful applications of heterogeneous lists, based on different choices of the first argument to **hlist**.

9.2.1 A Lambda Calculus Interpreter

Heterogeneous lists are very useful in implementing interpreters for functional programming languages. Using the types and operations already defined, it is trivial to write an interpreter for simply typed lambda calculus. Those familiar with the terminology of semantics may find it helpful to think of the interpreter as a denotational semantics.

We start with an algebraic datatype for types.

```
Inductive type : Set :=
| Unit : type
| Arrow : type → type → type.
```

Now we can define a type family for expressions. An **exp** *ts* *t* stands for an expression that has type t and whose free variables have types in the list *ts*. We effectively use the de Bruijn index variable representation [11]. Variables are represented as **member** values; that is, a

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

variable is more or less a constructive proof that a particular type is found in the type environment.

```
Inductive exp : list type → type → Set :=
| Const : ∀ ts, exp ts Unit

| Var : ∀ ts t, member t ts → exp ts t
| App : ∀ ts dom ran,
  exp ts (Arrow dom ran) → exp ts dom → exp ts ran
| Abs : ∀ ts dom ran, exp (dom :: ts) ran → exp ts (Arrow dom ran).
Implicit Arguments Const [ts].
```

We write a simple recursive function to translate types into Sets.

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Unit ⇒ unit
    | Arrow t1 t2 ⇒ typeDenote t1 → typeDenote t2
  end.
```

Now it is straightforward to write an expression interpreter. The type of the function, expDenote, tells us that we translate expressions into functions from properly typed environments to final values. An environment for a free variable list *ts* is simply an **hlist** typeDenote *ts*. That is, for each free variable, the heterogeneous list that is the environment must have a value of the variable's associated type. We use hget to implement the Var case, and we use HCons to extend the environment in the Abs case.

```
Fixpoint expDenote ts t (e : exp ts t)
  : hlist typeDenote ts → typeDenote t :=
  match e with
    | Const _ ⇒ fun _ ⇒ tt

    | Var _ _ mem ⇒ fun s ⇒ hget s mem
    | App _ _ _ e1 e2 ⇒ fun s ⇒ (expDenote e1 s) (expDenote e2 s)
    | Abs _ _ _ e' ⇒ fun s ⇒ fun x ⇒ expDenote e' (HCons x s)
  end.
```

As in previous examples, the interpreter is easy to run with simpl.

```
Eval simpl in expDenote Const HNil.

    = tt
     : typeDenote Unit

Eval simpl in expDenote (Abs (dom := Unit) (Var HFirst)) HNil.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
    = fun x : unit ⇒ x
    : typeDenote (Arrow Unit Unit)

Eval simpl in expDenote (Abs (dom := Unit)
  (Abs (dom := Unit) (Var (HNext HFirst)))) HNil.

    = fun x _ : unit ⇒ x
    : typeDenote (Arrow Unit (Arrow Unit Unit))

Eval simpl in expDenote (Abs (dom := Unit) (Abs (dom := Unit)
  (Var HFirst))) HNil.

    = fun _ x0 : unit ⇒ x0
    : typeDenote (Arrow Unit (Arrow Unit Unit))

Eval simpl in expDenote (App (Abs (Var HFirst)) Const) HNil.

    = tt
    : typeDenote Unit
```

We are starting to develop the tools underlying dependent typing's amazing advantage over alternative approaches in several important areas. Here, we have implemented complete syntax, typing rules, and evaluation semantics for simply typed lambda calculus without even needing to define a syntactic substitution operation. We did it all without a single line of proof, and the implementation is manifestly executable. Other, more common approaches to language formalization often state and prove explicit theorems about type safety of languages. In the preceding example, we established type safety, termination, and other metatheorems by reduction to the Calculus of Inductive Constructions, which we know has those properties.

9.3 Recursive Type Definitions

There is another style of datatype definition that leads to much simpler definitions of the get and hget functions. Because Coq supports type-level computation, we can redo the inductive definitions as *recursive* definitions. Here we preface type names with the letter *f* to indicate that they are based on explicit recursive *function* definitions.

```
Section filist.
  Variable A : Set.

  Fixpoint filist (n : nat) : Set :=
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
    match n with
      | O ⇒ unit
      | S n' ⇒ A × filist n'
    end%type.
```

We say that a list of length 0 has no contents, and a list of length S n' is a pair of a data value and a list of length n'.

```
  Fixpoint ffin (n : nat) : Set :=
    match n with
      | O ⇒ Empty_set
      | S n' ⇒ option (ffin n')
    end.
```

We express that there are no index values when n = O, by defining such indices as type **Empty_set**; and we express that at n = S n' there is a choice between picking the first element of the list (represented as None) or choosing a later element (represented by Some *idx*, where *idx* is an index into the list tail). For instance, the three values of type ffin 3 are None, Some None, and Some (Some None).

```
  Fixpoint fget (n : nat) : filist n → ffin n → A :=
    match n with
      | O ⇒ fun _ idx ⇒ match idx with end
      | S n' ⇒ fun ls idx ⇒
        match idx with
          | None ⇒ fst ls
          | Some idx' ⇒ fget n' (snd ls) idx'
        end
    end.
```

The new get implementation needs only one dependent match, and its annotation is inferred for us. Our choices of data structure implementations lead to just the right typing behavior for this new definition to work out.

```
End filist.
```

Heterogeneous lists are a little trickier to define with recursion, but we then reap similar benefits in simplicity of use.

```
Section fhlist.
  Variable A : Type.
  Variable B : A → Type.

  Fixpoint fhlist (ls : list A) : Type :=
    match ls with
      | nil ⇒ unit
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
    | x :: ls' ⇒ B x × fhlist ls'
  end%type.
```

The definition of fhlist follows the definition of filist, where some data elements now have more dependent types.

```
Variable elm : A.

Fixpoint fmember (ls : list A) : Type :=
  match ls with
    | nil ⇒ Empty_set
    | x :: ls' ⇒ (x = elm) + fmember ls'
  end%type.
```

The definition of fmember follows the definition of ffin. Empty lists have no members, and member types for nonempty lists are built by adding one new option to the type of members of the list tail. While for ffin we needed no new information associated with the option that we add, here we need to know that the head of the list equals the element we are searching for. We express that idea with a sum type whose left branch is the appropriate equality proposition. Since we define fmember to live in Type, we can insert Prop types as needed, because Prop is a subtype of Type.

Here is a first attempt to write a get function for fhlists.

```
Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
  match ls with
    | nil ⇒ fun _ idx ⇒ match idx with end
    | _ :: ls' ⇒ fun mls idx ⇒
      match idx with
        | inl _ ⇒ fst mls
        | inr idx' ⇒ fhget ls' (snd mls) idx'
      end
  end.
```

Only one problem remains. The expression fst *mls* is not known to have the proper type. To demonstrate that it does, we need to use the proof available in the inl case of the inner match.

```
Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
  match ls with
    | nil ⇒ fun _ idx ⇒ match idx with end
    | _ :: ls' ⇒ fun mls idx ⇒
      match idx with
        | inl pf ⇒ match pf with
                      | eq_refl ⇒ fst mls
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
                    end
              | inr idx' ⇒ fhget ls' (snd mls) idx'
          end
      end.
```

By pattern matching on the equality proof *pf*, we make that equality known to the type checker. Exactly why this works can be seen by studying the definition of equality.

```
Print eq.
```

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

In a proposition $x = y$, we see that x is a parameter and y is a regular argument. The type of the constructor eq_refl shows that y can only ever be instantiated to x. Thus, within a pattern match with eq_refl, occurrences of y can be replaced with occurrences of x for typing purposes.

```
End fhlist.

Implicit Arguments fhget [A B elm ls].
```

How does one choose between the two data structure encoding strategies presented so far? Before deciding, we study one further approach.

9.4 Data Structures as Index Functions

Indexed lists can be useful in defining other inductive types with constructors that take variable numbers of arguments. In this section, we consider parameterized trees with arbitrary branching factor.

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, ilist tree n → tree.
End tree.
```

Every Node of a **tree** has a natural number argument, which gives the number of child trees in the second argument, typed with **ilist**. We can define two operations on trees of naturals: summing their elements and incrementing their elements. It is useful to define a generic fold function on **ilist**s first.

```
Section ifoldr.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
  Variables A B : Set.
  Variable f : A → B → B.
  Variable i : B.

  Fixpoint ifoldr n (ls : ilist A n) : B :=
    match ls with
      | Nil ⇒ i
      | Cons _ x ls' ⇒ f x (ifoldr ls')
    end.
End ifoldr.

Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ ls ⇒ ifoldr (fun t' n ⇒ sum t' + n) O ls
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ ls ⇒ Node (imap inc ls)
  end.
```

Now we might like to prove that inc does not decrease a tree's sum.

```
Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  induction t; crush.
```

```
  n : nat
  i : ilist (tree nat) n
  ============================
   ifoldr (fun (t' : tree nat) (n0 : nat) ⇒ sum t' + n0) 0 (imap inc i)
   ≥ ifoldr (fun (t' : tree nat) (n0 : nat) ⇒ sum t' + n0) 0 i
```

We are left with a single subgoal that does not seem provable directly. This is the same problem we encountered in Chapter 3 with other nested inductive types.

```
  Check tree_ind.

  tree_ind
     : ∀ (A : Set) (P : tree A → Prop),
       (∀ a : A, P (Leaf a)) →
       (∀ (n : nat) (i : ilist (tree A) n), P (Node i)) →
       ∀ t : tree A, P t
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

The automatically generated induction principle is too weak. For the Node case, it gives no inductive hypothesis. We could write an alternative induction principle, as in Chapter 3, but there is an easier way, if we alter the definition of **tree**.

```
Abort.

Reset tree.
```

First, let us try using the recursive definition of **ilist**s instead of the inductive version.

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, filist tree n → tree.
```

```
Error: Non strictly positive occurrence of "tree" in
 "forall n : nat, filist tree n -> tree"
```

The special-case rule for nested datatypes only works with nested uses of other inductive types, which could be replaced with uses of new mutually inductive types. We defined filist recursively, so it may not be used in nested inductive definitions.

The final solution uses yet another of the inductive definition techniques introduced in Chapter 3, reflexive types. Instead of merely using **fin** to get elements out of **ilist**, we can *define* **ilist** in terms of **fin**. For the reasons outlined in Section 9.3, it is easier to work with ffin in place of **fin**.

```
  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, (ffin n → tree) → tree.
```

A Node is indexed by a natural number n, and the node's n children are represented as a function from ffin n to trees, which is isomorphic to the **ilist**-based representation used earlier.

```
End tree.

Implicit Arguments Node [A n].
```

We can redefine sum and inc for the new **tree** type. Again, it is useful to define a generic fold function first. This time, it takes in a function whose domain is some ffin type, and it folds another function over the results of calling the first function at every possible ffin value.

```
Section rifoldr.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
  Variables A B : Set.
  Variable f : A → B → B.
  Variable i : B.

  Fixpoint rifoldr (n : nat) : (ffin n → A) → B :=
    match n with
      | O ⇒ fun _ ⇒ i
      | S n' ⇒ fun get ⇒ f (get None)
        (rifoldr n' (fun idx ⇒ get (Some idx)))
    end.
End rifoldr.

Implicit Arguments rifoldr [A B n].

Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ f ⇒ rifoldr plus O (fun idx ⇒ sum (f idx))
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ f ⇒ Node (fun idx ⇒ inc (f idx))
  end.
```

Now we are ready to prove the theorem. We do not need to define any new induction principle, but it will be helpful to prove some lemmas.

```
Lemma plus_ge : ∀ x1 y1 x2 y2,
  x1 ≥ x2
  → y1 ≥ y2
  → x1 + y1 ≥ x2 + y2.
  crush.
Qed.

Lemma sum_inc' : ∀ n (f1 f2 : ffin n → nat),
  (∀ idx, f1 idx ≥ f2 idx)
  → rifoldr plus O f1 ≥ rifoldr plus O f2.
  Hint Resolve plus_ge.

  induction n; crush.
Qed.

Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  Hint Resolve sum_inc'.

  induction t; crush.
Qed.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

Even if Coq generated complete induction principles automatically for nested inductive definitions like the one we started with, there would still be advantages to using this style of reflexive encoding. We see one of those advantages in the definition of inc, where we did not need to use any kind of auxiliary function. In general, reflexive encodings often admit direct implementations of operations that would require recursion if performed with more traditional inductive data structures.

9.4.1 Another Interpreter Example

We develop another example of variable arity constructors, in the form of optimization of a small expression language with a construct like Scheme's `cond`. Each conditional expression takes a list of pairs of Boolean tests and bodies. The value of the conditional comes from the body of the first test in the list to evaluate to true. To simplify the interpreter, we force each conditional to include a final, default case.

```
Inductive type' : Type := Nat | Bool.

Inductive exp' : type' → Type :=
| NConst : nat → exp' Nat
| Plus : exp' Nat → exp' Nat → exp' Nat
| Eq : exp' Nat → exp' Nat → exp' Bool

| BConst : bool → exp' Bool

| Cond : ∀ n t, (ffin n → exp' Bool)
  → (ffin n → exp' t) → exp' t → exp' t.
```

A Cond is parameterized by a natural n, which tells us how many cases this conditional has. The test expressions are represented with a function of type ffin $n \rightarrow$ **exp'** Bool, and the bodies are represented with a function of type ffin $n \rightarrow$ **exp'** t, where t is the overall type. The final **exp'** t argument is the default case. For example, here is an expression that successively checks whether $2 + 2 = 5$ (returning 0 if so) or if $1 + 1 = 2$ (returning 1 if so), returning 2 otherwise:

```
Example ex1 := Cond 2
  (fun f ⇒ match f with
              | None ⇒ Eq (Plus (NConst 2) (NConst 2)) (NConst 5)
              | Some None ⇒ Eq (Plus (NConst 1) (NConst 1))
                 (NConst 2)
              | Some (Some v) ⇒ match v with end
            end)
  (fun f ⇒ match f with
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
              | None ⇒ NConst 0
              | Some None ⇒ NConst 1
              | Some (Some v) ⇒ match v with end
            end)
  (NConst 2).
```

We start implementing the interpreter with a standard type denotation function.

```
Definition type'Denote (t : type') : Set :=
  match t with
    | Nat ⇒ nat
    | Bool ⇒ bool
  end.
```

To implement the expression interpreter, it is useful to have the following function that implements the functionality of Cond without involving any syntax.

```
Section cond.
  Variable A : Set.
  Variable default : A.

  Fixpoint cond (n : nat) : (ffin n → bool) → (ffin n → A) → A :=
    match n with
      | O ⇒ fun _ _ ⇒ default
      | S n' ⇒ fun tests bodies ⇒
        if tests None
          then bodies None
          else cond n'
            (fun idx ⇒ tests (Some idx))
            (fun idx ⇒ bodies (Some idx))
    end.
End cond.

Implicit Arguments cond [A n].
```

Now the expression interpreter is straightforward to write.

```
Fixpoint exp'Denote t (e : exp' t) : type'Denote t :=
  match e with
    | NConst n ⇒ n
    | Plus e1 e2 ⇒ exp'Denote e1 + exp'Denote e2
    | Eq e1 e2 ⇒
      if eq_nat_dec (exp'Denote e1) (exp'Denote e2) then true
        else false
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
    | BConst b ⇒ b
    | Cond _ _ tests bodies default ⇒
      cond
      (exp'Denote default)
      (fun idx ⇒ exp'Denote (tests idx))
      (fun idx ⇒ exp'Denote (bodies idx))
  end.
```

We implement a constant-folding function that optimizes conditionals, removing cases with known false tests and cases that come after known true tests. A function cfoldCond implements the heart of this logic. The convoy pattern is used again near the end of the implementation.

```
Section cfoldCond.
  Variable t : type'.
  Variable default : exp' t.

  Fixpoint cfoldCond (n : nat)
    : (ffin n → exp' Bool) → (ffin n → exp' t) → exp' t :=
    match n with
      | O ⇒ fun _ _ ⇒ default
      | S n' ⇒ fun tests bodies ⇒
        match tests None return _ with
          | BConst true ⇒ bodies None
          | BConst false ⇒ cfoldCond n'
            (fun idx ⇒ tests (Some idx))
            (fun idx ⇒ bodies (Some idx))
          | _ ⇒
            let e := cfoldCond n'
              (fun idx ⇒ tests (Some idx))
              (fun idx ⇒ bodies (Some idx)) in
            match e in exp' t return exp' t → exp' t with
              | Cond n _ tests' bodies' default' ⇒ fun body ⇒
                Cond
                (S n)
                (fun idx ⇒ match idx with
                              | None ⇒ tests None
                              | Some idx ⇒ tests' idx
                            end)
                (fun idx ⇒ match idx with
                              | None ⇒ body
                              | Some idx ⇒ bodies' idx
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001


```
                              end)
                     default'
                   | e ⇒ fun body ⇒
                     Cond
                     1
                     (fun _ ⇒ tests None)
                     (fun _ ⇒ body)
                     e
                 end (bodies None)
             end
         end.
End cfoldCond.
```

```
Implicit Arguments cfoldCond [t n].
```

As in the interpreters, most of the action was in this helper function, and cfold itself is easy to write.

```
Fixpoint cfold t (e : exp' t) : exp' t :=
  match e with
    | NConst n ⇒ NConst n
    | Plus e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
      match e1', e2' return exp' Nat with
        | NConst n1, NConst n2 ⇒ NConst (n1 + n2)
        | _, _ ⇒ Plus e1' e2'
      end
    | Eq e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
      match e1', e2' return exp' Bool with
        | NConst n1, NConst n2 ⇒
          BConst (if eq_nat_dec n1 n2 then true else false)
        | _, _ ⇒ Eq e1' e2'
      end

    | BConst b ⇒ BConst b
    | Cond _ _ tests bodies default ⇒
      cfoldCond
      (cfold default)
      (fun idx ⇒ cfold (tests idx))
      (fun idx ⇒ cfold (bodies idx))
  end.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

To prove the final correctness theorem, it is useful to know that cfoldCond preserves expression meanings. The following lemma formalizes that property. The proof is a standard, mostly automated one, along with a guided instantiation of the quantifiers in the induction hypothesis.

```
Lemma cfoldCond_correct : ∀ t (default : exp' t)
  n (tests : ffin n → exp' Bool) (bodies : ffin n → exp' t),
  exp'Denote (cfoldCond default tests bodies)
  = exp'Denote (Cond n tests bodies default).
  induction n; crush;
    match goal with
      | [ IHn : ∀ tests bodies, _, tests : _ → _, bodies : _ → _ ⊢ _ ] ⇒
        specialize (IHn (fun idx ⇒ tests (Some idx))
          (fun idx ⇒ bodies (Some idx)))
    end;
    repeat (match goal with
              | [ ⊢ context[match ?E with NConst _ ⇒ _
                                | _ ⇒ _ end] ] ⇒
                dep_destruct E
              | [ ⊢ context[if ?B then _ else _] ] ⇒ destruct B
            end; crush).
Qed.
```

It is also useful to know that the result of a call to cond is not changed by substituting new tests and bodies functions, so long as the new functions have the same input-output behavior as the old. It turns out that, in Coq, it is not possible to prove in general that functions related in this way are equal (see Section 10.6). For now, it suffices to prove that the particular function cond is *extensional*; that is, it is unaffected by substitution of functions with input-output equivalents.

```
Lemma cond_ext : ∀ (A : Set) (default : A) n
  (tests tests' : ffin n → bool) (bodies bodies' : ffin n → A),
  (∀ idx, tests idx = tests' idx)
  → (∀ idx, bodies idx = bodies' idx)
  → cond default tests bodies
  = cond default tests' bodies'.
  induction n; crush;
    match goal with
      | [ ⊢ context[if ?E then _ else _] ] ⇒ destruct E
    end; crush.
Qed.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

Now the final theorem is easy to prove.

```
Theorem cfold_correct : ∀ t (e : exp' t),
  exp'Denote (cfold e) = exp'Denote e.
  Hint Rewrite cfoldCond_correct.
  Hint Resolve cond_ext.

  induction e; crush;
    repeat (match goal with
              | [ ⊢ context[cfold ?E] ] ⇒ dep_destruct (cfold E)
            end; crush).
Qed.
```

We add the two lemmas as hints and perform standard automation with pattern matching of subterms to destruct.

9.5 Choosing between Representations

It is not always clear which of these representation techniques to apply in a particular situation, but I summarize here the pros and cons of each.

Inductive types are often the most pleasant to work with, after someone has spent the time implementing some basic library functions for them using `match` annotations. Many aspects of Coq's logic and tactic support are specialized to deal with inductive types, and one may miss out with other encodings.

Recursive types usually involve much less initial effort, but they can be less convenient to use with proof automation. For instance, the `simpl` tactic (which is among the ingredients in *crush*) is sometimes overzealous in simplifying uses of functions over recursive types. Consider a call get l f, where variable l has type filist A (S n). The type of l would be simplified to an explicit pair type. In a proof involving many recursive types, this kind of unhelpful "simplification" can lead to rapid bloat in the sizes of subgoals. Even worse, it can prevent syntactic pattern matching, as in cases where filist is expected but a pair type is found in the "simplified" version. The same problem applies to applications of recursive functions to values in recursive types: the recursive function call may "simplify" when the top-level structure of the type index but not the recursive value is known, because such functions are generally defined by recursion on the index, not the value.

Another disadvantage of recursive types is that they only apply to type families whose indices determine their skeletons. This is not true for all data structures; a good counterexample comes from the richly

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a portion of the eBook at doi:10.7551/mitpress/9153.001.0001

typed programming language syntax types we have used several times so far. The fact that a piece of syntax has type Nat says nothing about the tree structure of that syntax.

It is also true that parameterized recursive types are hard to use within the definitions of nested inductive types, as we saw in Section 9.4. Coq's well-formedness check on inductive definitions will accept many definitions that use inductive or function-based nested types but reject similar definitions based on recursive types.

Finally, Coq type inference can be more helpful in constructing values in inductive types. Application of a particular constructor of that type tells Coq what to expect from the arguments, whereas, for instance, forming a generic pair does not make clear an intention to interpret the value as belonging to a particular recursive type. This downside can be mitigated to an extent by writing constructor functions for a recursive type, mirroring the definition of the corresponding inductive type.

Reflexive encodings of datatypes are seen relatively rarely. As the earlier examples demonstrated, manipulating index values manually can lead to hard-to-read code. A normal inductive type is generally easier to work with, once someone has gone through the trouble of implementing an induction principle manually with the techniques studied in Chapter 3. For small developments, avoiding that kind of coding can justify the use of reflexive data structures. There are also some useful instances of co-inductive definitions with nested data structures (e.g., lists of values in the co-inductive type) that can only be deconstructed effectively with reflexive encoding of the nested structures.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

This is a section of doi:10.7551/mitpress/9153.001.0001

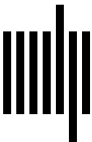
Certified Programming with Dependent Types

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

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024

MIT Press Direct

© 2013 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in 10/13 Lucida Bright by the author using $\LaTeX\, 2_\varepsilon$. 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

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2274826/9780262317870_cai.pdf by guest on 07 December 2024