10 Reasoning about Equality Proofs

In traditional mathematics, the concept of equality is usually taken as a given, but in type theory, equality is a very contentious subject. There are at least three different notions of equality that are important in Coq, and researchers are actively investigating new definitions of what it means for two terms to be equal. Even once we fix a notion of equality, there are inevitably issues that arise in proving properties of programs that manipulate equality proofs explicitly. In this chapter, I introduce the different notions of equality and describe design patterns for manipulating equality proofs in programs.

10.1 The Definitional Equality

In many examples so far, proof goals followed by computation. That is, we applied computational reduction rules to reduce the goal to a normal form, at which point it followed trivially. Exactly when this works and when it does not depends on the details of Coq's *definitional equality*. This is an untyped binary relation appearing in the formal metatheory of the Calculus of Inductive Constructions (CIC), which contains a typing rule allowing the conclusion $E : T$ from the premise $E : T'$ and a proof that T and T' are definitionally equal.

The `cbv` tactic helps us illustrate the rules of Coq's definitional equality. We redefine the natural number predecessor function and construct a manual proof that it returns 0 when applied to 1.

```
Definition pred' (x : nat) :=
  match x with
    | O ⇒ O
    | S n' ⇒ let y := n' in y
  end.

Theorem reduce_me : pred' 1 = 0.
```

CIC follows the traditions of lambda calculus in associating reduction rules with Greek letters. Coq can certainly be said to support the familiar alpha reduction rule, which allows capture-avoiding renaming of bound variables, but we never need to apply alpha explicitly, since Coq uses a de Bruijn representation [11] that encodes terms canonically.

The delta rule is for unfolding global definitions. We can use it here to unfold the definition of pred'. We do this with the `cbv` tactic, which takes a list of reduction rules and makes as many call-by-value reduction steps as possible, using only those rules. There is an analogous tactic `lazy` for call-by-need reduction.

```
cbv delta.
```

```
============================
 (fun x : nat ⇒ match x with
                | 0 ⇒ 0
                | S n' ⇒ let y := n' in y
                end) 1 = 0
```

At this point, we apply the beta reduction of lambda calculus to simplify the application of a known function abstraction.

```
cbv beta.
```

```
============================
 match 1 with
 | 0 ⇒ 0
 | S n' ⇒ let y := n' in y
 end = 0
```

Next is the iota reduction, which simplifies a single `match` term by determining which pattern matches.

```
cbv iota.
```

```
============================
 (fun n' : nat ⇒ let y := n' in y) 0 = 0
```

Now we need another beta reduction.

```
cbv beta.
```

```
============================
 (let y := 0 in y) = 0
```

The final reduction rule is zeta, which replaces a `let` expression by its body with the appropriate term substituted.

```
cbv zeta.
```

```
============================
```

```
    0 = 0

  reflexivity.
Qed.
```

The `beta` reduction rule applies to recursive functions as well, and its behavior may be surprising in some instances. For instance, we can run some simple tests using the reduction strategy `compute`, which applies all applicable rules of the definitional equality.

```
Definition id (n : nat) := n.
Eval compute in fun x ⇒ id x.

     = fun x : nat ⇒ x

Fixpoint id' (n : nat) := n.
Eval compute in fun x ⇒ id' x.

     = fun x : nat ⇒ (fix id' (n : nat) : nat := n) x
```

By running `compute`, we ask Coq to run reduction steps until no more apply, so why do we see an application of a known function where clearly no beta reduction has been performed? The answer has to do with ensuring termination of all Gallina programs. One candidate rule would say that we apply recursive definitions wherever possible. However, this would clearly lead to nonterminating reduction sequences, since the function may appear fully applied within its own definition, and we would naïvely "simplify" such applications immediately. Instead, Coq only applies the beta rule for a recursive function when *the top-level structure of the recursive argument is known.* For `id'`, we have only one argument n, so clearly it is the recursive argument, and the top-level structure of n is known when the function is applied to `O` or to some `S` e term. The variable x is neither, so reduction is blocked.

What are recursive arguments in general? Every recursive function is compiled by Coq to a `fix` expression, for anonymous definition of recursive functions. Further, every `fix` with multiple arguments has one designated as the recursive argument via a `struct` annotation. The recursive argument is the one that must decrease across recursive calls, to appease Coq's termination checker. Coq will generally infer which argument is recursive, though we may also specify it manually, if we want to tweak reduction behavior. For instance, consider this definition of a function to add two lists of **nat**s elementwise:

```
Fixpoint addLists (ls1 ls2 : list nat) : list nat :=
  match ls1, ls2 with
    | n1 :: ls1' , n2 :: ls2' ⇒ n1 + n2 :: addLists ls1' ls2'
```

```
  | _, _ ⇒ nil
end.
```

By default, Coq chooses *ls1* as the recursive argument. We see that *ls2* would have been another valid choice. The choice has a critical effect on reduction behavior, as two examples illustrate.

```
Eval compute in fun ls ⇒ addLists nil ls.

    = fun _ : list nat ⇒ nil

Eval compute in fun ls ⇒ addLists ls nil.

    = fun ls : list nat ⇒
      (fix addLists (ls1 ls2 : list nat) : list nat :=
         match ls1 with
         | nil ⇒ nil
         | n1 :: ls1' ⇒
             match ls2 with
             | nil ⇒ nil
             | n2 :: ls2' ⇒
                 (fix plus (n m : nat) : nat :=
                    match n with
                    | 0 ⇒ m
                    | S p ⇒ S (plus p m)
                    end) n1 n2 :: addLists ls1' ls2'
             end
         end) ls nil
```

The outer application of the `fix` expression for addLists was only simplified in the first case, because in the second case the recursive argument is *ls*, whose top-level structure is not known.

The opposite behavior pertains to a version of addLists with *ls2* marked as recursive.

```
Fixpoint addLists' (ls1 ls2 : list nat) {struct ls2} : list nat :=
  match ls1, ls2 with
    | n1 :: ls1' , n2 :: ls2' ⇒ n1 + n2 :: addLists' ls1' ls2'
    | _, _ ⇒ nil
  end.

Eval compute in fun ls ⇒ addLists' ls nil.

    = fun ls : list nat ⇒ match ls with
                          | nil ⇒ nil
                          | _ :: _ ⇒ nil
                          end
```

We see that all use of recursive functions has been eliminated, though the term has not quite simplified to nil. We could get it to do so by switching the order of the `match` discriminees in the definition of addLists'.

Recall that co-recursive definitions have a dual rule: a co-recursive call only simplifies when it is the discriminee of a `match`. This condition is built into the beta rule for `cofix`, the anonymous form of `CoFixpoint`.

The standard **eq** relation is critically dependent on the definitional equality. The relation **eq** is often called a *propositional equality*, because it reifies definitional equality as a proposition that may or may not hold. Standard axiomatizations of an equality predicate in first-order logic define equality in terms of properties it has, like reflexivity, symmetry, and transitivity. In contrast, for **eq** in Coq, those properties are implicit in the properties of the definitional equality, which are built into CIC's metatheory and the implementation of Gallina. We could add new rules to the definitional equality, and **eq** would keep its definition and methods of use.

This may sound like the choice of **eq**'s definition is unimportant. To the contrary, this chapter gives examples where alternative definitions may simplify proofs. Before that, I introduce proof methods for goals that use proofs of the standard propositional equality as data.

10.2 Heterogeneous Lists Revisited

An example dependent data structure was the heterogeneous list and its associated cursor type (see Section 9.2). The recursive version poses some special challenges related to equality proofs, since it uses such proofs in its definition of fmember types (see Section 9.3 or the code repeated below).

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

  Fixpoint fhlist (ls : list A) : Type :=
    match ls with
      | nil ⇒ unit
      | x :: ls' ⇒ B x × fhlist ls'
    end%type.

  Variable elm : A.

  Fixpoint fmember (ls : list A) : Type :=
```

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

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

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

We can define a map-like function for fhlists.

```
Section fhlist_map.
  Variables A : Type.
  Variables B C : A → Type.
  Variable f : ∀ x, B x → C x.

  Fixpoint fhmap (ls : list A) : fhlist B ls → fhlist C ls :=
    match ls return fhlist B ls → fhlist C ls with
      | nil ⇒ fun _ ⇒ tt
      | _ :: _ ⇒ fun hls ⇒ (f (fst hls), fhmap _ (snd hls))
    end.

  Implicit Arguments fhmap [ls].
```

For the inductive versions of the ilist definitions, we proved a lemma about the interaction of get and imap. It was a strategic choice not to attempt such a proof for the preceding definitions, which brings us to the problems that are the subject of this chapter.

```
  Variable elm : A.

  Theorem fhget_fhmap : ∀ ls (mem : fmember elm ls) (hls : fhlist B ls),
    fhget (fhmap hls) mem = f (fhget hls mem).
    induction ls; crush.
```

In Coq 8.2, one subgoal remains at this point. Coq 8.3 has added some tactic improvements that enable *crush* to complete all of both

inductive cases. To introduce the basics of reasoning about equality, it is useful to review what was necessary in Coq 8.2.

Part of the single remaining subgoal is

```
a0 : a = elm
============================
 match a0 in (_ = a2) return (C a2) with
 | eq_refl ⇒ f a1
 end = f match a0 in (_ = a2) return (B a2) with
         | eq_refl ⇒ a1
         end
```

This seems like a trivial obligation. The equality proof *a0* must be eq_refl, the only constructor of **eq**. Therefore, both the `match`es reduce to the point where the conclusion follows by reflexivity.

```
destruct a0.
```

```
User error: Cannot solve a second-order unification problem
```

This is one of Coq's standard error messages for indicating a failure in its heuristics for attempting an instance of an undecidable problem about dependent typing. We might try to nudge things in the right direction by stating the lemma that we believe makes the conclusion trivial.

```
assert (a0 = eq_refl _).
```

```
The term "eq_refl ?98" has type "?98 = ?98"
while it is expected to have type "a = elm"
```

In retrospect, the problem is not hard to see. Reflexivity proofs only show $x = x$ for particular values of x, whereas here we are thinking in terms of a proof of $a = elm$, where the two sides of the equality are not equal syntactically. Thus, the essential lemma we need does not even type-check.

This chapter gives several useful patterns for proving obligations like this.

For this particular example, the solution is straightforward. The `destruct` tactic has a simpler sibling `case` that should behave identically for any inductive type with one constructor of no arguments.

```
case a0.
```

```
============================
 f a1 = f a1
```

It seems that `destruct` was trying to be too smart.

```
  reflexivity.
Qed.
```

It is helpful to examine the proof terms generated by this sort of strategy. A simpler example illustrates this.

```
Lemma lemma1 : ∀ x (pf : x = elm),
  O = match pf with eq_refl ⇒ O end.
  simple destruct pf; reflexivity.
Qed.
```

The tactic `simple destruct` *pf* is a convenient form for applying `case`. It runs `intro` to bring into scope all quantified variables up to its argument.

```
Print lemma1.
```

```
lemma1 =
fun (x : A) (pf : x = elm) ⇒
match pf as e in (_ = y) return (0 = match e with
                                     | eq_refl ⇒ 0
                                     end) with
| eq_refl ⇒ eq_refl 0
end
     : ∀ (x : A) (pf : x = elm), 0 = match pf with
                                     | eq_refl ⇒ 0
                                     end
```

Using what we know about shorthands for `match` annotations, we can write this proof in shorter form manually.

```
Definition lemma1' (x : A) (pf : x = elm) :=
  match pf return (0 = match pf with
                       | eq_refl ⇒ 0
                       end) with
    | eq_refl ⇒ eq_refl 0
  end.
```

Surprisingly, what seems at first like a simpler lemma is harder to prove.

```
Lemma lemma2 : ∀ (x : A) (pf : x = x),
  O = match pf with eq_refl ⇒ O end.
  simple destruct pf.
```

```
User error: Cannot solve a second-order unification problem
```

```
  Abort.
```

Nonetheless, we can adapt the last manual proof to handle this theorem.

```
  Definition lemma2 :=
    fun (x : A) (pf : x = x) =>
      match pf return (0 = match pf with
                           | eq_refl => 0
                           end) with
        | eq_refl => eq_refl 0
      end.
```

We can try to prove a lemma that would simplify proofs of many facts like lemma2.

```
  Lemma lemma3 : forall (x : A) (pf : x = x), pf = eq_refl x.
    simple destruct pf.

User error: Cannot solve a second-order unification problem
  Abort.
```

This time, even our manual attempt fails.

```
  Definition lemma3' :=
    fun (x : A) (pf : x = x) =>
      match pf as pf' in (_ = x') return (pf' = eq_refl x') with
        | eq_refl => eq_refl _
      end.

The term "eq_refl x'" has type "x' = x'"
while it is expected to have type "x = x'"
```

The type error comes from the `return` annotation. In that annotation, the `as`-bound variable *pf'* has type $x = x'$, referring to the `in`-bound variable *x'*. To do a dependent `match`, we must choose a fresh name for the second argument of **eq** and use the "real" value x for the first argument. Thus, within the `return` clause, the proof we are matching on must equate two nonmatching terms, which makes it impossible to equate that proof with reflexivity.

Nonetheless, it turns out that, with one catch, we can prove this lemma.

```
  Lemma lemma3 : forall (x : A) (pf : x = x), pf = eq_refl x.
    intros; apply UIP_refl.
  Qed.

  Check UIP_refl.
```

```
UIP_refl
     : ∀ (U : Type) (x : U) (p : x = x), p = eq_refl x
```

The theorem UIP_refl comes from the Eqdep module of the standard library. (Its name uses the acronym UIP for "unicity of identity proofs.") Do the Coq authors know some clever technique for building such proofs that we have not seen yet? If they do, they did not use it for this proof. Rather, the proof is based on an *axiom*, the term *eq_rect_eq*.

```
Print eq_rect_eq.
```

```
*** [ eq_rect_eq :
∀ (U : Type) (p : U) (Q : U → Type) (x : Q p) (h : p = p),
x = eq_rect p Q x p h ]
```

The axiom *eq_rect_eq* states a fact that seems like common sense, once the notation is deciphered. The term eq_rect is the automatically generated recursion principle for **eq**. Calling eq_rect is another way of `match`ing on an equality proof. The proof we match on is the argument h, and x is the body of the `match`. The statement of *eq_rect_eq* says that `match`es on proofs of $p = p$, for any p, are superfluous and may be removed. We can see this intuition better in code by asking Coq to simplify the theorem statement with the `compute` reduction strategy.

```
Eval compute in (∀ (U : Type) (p : U) (Q : U → Type)
  (x : Q p) (h : p = p),
  x = eq_rect p Q x p h).
```

```
  = ∀ (U : Type) (p : U) (Q : U → Type) (x : Q p) (h : p = p),
    x = match h in (_ = y) return (Q y) with
        | eq_refl ⇒ x
        end
```

We cannot prove *eq_rect_eq* from within Coq. This proposition is introduced as an axiom; that is, a proposition asserted as true without proof. We cannot assert just any statement without proof. For instance, adding **False** as an axiom would allow us to prove any proposition, defeating the point of using a proof assistant. In general, we need to be sure that we never assert *inconsistent* sets of axioms. A set of axioms is inconsistent if its conjunction implies **False**. For the case of *eq_rect_eq*, consistency has been verified outside of Coq via informal metatheory [43], in a study that also established unprovability of the axiom in CIC.

This axiom is equivalent to another that is more commonly known and mentioned in type theory circles.

```
Check Streicher_K.
```

```
Streicher_K
    : ∀ (U : Type) (x : U) (P : x = x → Prop),
      P eq_refl → ∀ p : x = x, P p
```

This refers to Streicher's axiom K [43], which says that a predicate on properly typed equality proofs holds of all such proofs if it holds of reflexivity.

```
End fhlist_map.
```

It is possible to avoid axioms altogether for equalities on types with decidable equality. The Eqdep_dec module of the standard library contains a parametric proof of UIP_refl for such cases. To simplify presentation, we continue to work with the axiom version in the rest of this chapter.

10.3 Type Casts in Theorem Statements

Sometimes we need to use tricks with equality just to state the theorems that we care about. To illustrate, we start by defining a concatenation function for fhlists.

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

  Fixpoint fhapp (ls1 ls2 : list A)
    : fhlist B ls1 → fhlist B ls2 → fhlist B (ls1 ++ ls2) :=
    match ls1 with
      | nil ⇒ fun _ hls2 ⇒ hls2
      | _ :: _ ⇒ fun hls1 hls2 ⇒ (fst hls1, fhapp _ _ (snd hls1) hls2)
    end.

  Implicit Arguments fhapp [ls1 ls2].
```

We might like to prove that fhapp is associative.

```
  Theorem fhapp_assoc : ∀ ls1 ls2 ls3
    (hls1 : fhlist B ls1) (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) = fhapp (fhapp hls1 hls2) hls3.
```

```
The term
"fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
  (fhapp (ls1:=ls1) (ls2:=ls2) hls1 hls2) hls3"
has type "fhlist B ((ls1 ++ ls2) ++ ls3)"
```

```
while it is expected to have type
"fhlist B (ls1 ++ ls2 ++ ls3)"
```

This first attempt at the theorem statement does not even type-check. We know that the two fhlist types appearing in the error message are always equal, by associativity of normal list append, but this fact is not apparent to the type checker. This stems from the fact that Coq's equality is *intensional*, in the sense that type equality theorems can never be applied after the fact to get a term to type-check. Instead, we need to make use of equality explicitly in the theorem statement.

```
Theorem fhapp_assoc : ∀ ls1 ls2 ls3
  (pf : (ls1 ++ ls2) ++ ls3 = ls1 ++ (ls2 ++ ls3))
  (hls1 : fhlist B ls1) (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
  fhapp hls1 (fhapp hls2 hls3)
  = match pf in (_ = ls) return fhlist _ ls with
      | eq_refl ⇒ fhapp (fhapp hls1 hls2) hls3
    end.
  induction ls1; crush.
```

The first remaining subgoal looks trivial enough.

```
============================
 fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3 =
 match pf in (_ = ls) return (fhlist B ls) with
 | eq_refl ⇒ fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3
 end
```

We can try what worked in previous examples.

```
  case pf.
```

```
User error: Cannot solve a second-order unification problem
```

It seems we have reached another case where it is unclear how to use a dependent `match` to implement case analysis on the proof. We can try the UIP_refl theorem again.

```
  rewrite (UIP_refl _ _ pf).
```

```
============================
 fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3 =
 fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3

  reflexivity.
```

The second subgoal is trickier.

```
pf : a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3
```

```
============================
 (a0,
 fhapp (ls1:=ls1) (ls2:=ls2 ++ ls3) b
   (fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3)) =
 match pf in (_ = ls) return (fhlist B ls) with
 | eq_refl ⇒
     (a0,
     fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
       (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
 end

  rewrite (UIP_refl _ _ pf).
```

```
The term "pf" has type
"a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3"
while it is expected to have type "?556 = ?556"
```

We can only apply UIP_refl on proofs of equality with syntactically equal operands, which is not the case with *pf* here. We need to manipulate the form of this subgoal to get to a point where we may use UIP_refl. A first step is obtaining a proof suitable to use in applying the induction hypothesis. Inversion on the structure of *pf* is sufficient for that.

```
  injection pf; intro pf'.
```

```
pf : a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3
pf' : (ls1 ++ ls2) ++ ls3 = ls1 ++ ls2 ++ ls3
============================
 (a0,
 fhapp (ls1:=ls1) (ls2:=ls2 ++ ls3) b
   (fhapp (ls1:=ls2) (ls2:=ls3) hls2 hls3)) =
 match pf in (_ = ls) return (fhlist B ls) with
 | eq_refl ⇒
     (a0,
     fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
       (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
 end
```

Now we can rewrite using the inductive hypothesis.

```
  rewrite (IHls1 _ _ pf').
```

```
============================
```

```
(a0,
match pf' in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒
    fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
      (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3
end) =
match pf in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒
    (a0,
    fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3)
      (fhapp (ls1:=ls1) (ls2:=ls2) b hls2) hls3)
end
```

We have made progress, as now only a single call to fhapp appears in the conclusion, repeated twice. Trying case analysis on the proofs still will not work, but we can enable it. Not only does just one call to fhapp matter now but it also *does not matter what the result of the call is*. In other words, the subgoal should remain true if we replace this fhapp call with a fresh variable. The `generalize` tactic helps us do exactly that.

```
generalize (fhapp (fhapp b hls2) hls3).
```

```
∀ f : fhlist B ((ls1 ++ ls2) ++ ls3),
(a0,
match pf' in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒ f
end) =
match pf in (_ = ls) return (fhlist B ls) with
| eq_refl ⇒ (a0, f)
end
```

The conclusion has gotten markedly simpler. It seems counterintuitive that proving a more general theorem is easier, but such is the case here and in many other proofs that use dependent types heavily. Speaking informally, the reason is that `match` annotations contain some positions where only variables are allowed. When more elements of a goal are reduced to variables, built-in tactics can have more success building `match` terms.

In this case, it is helpful to generalize over the two proofs as well.

```
generalize pf pf'.
```

```
∀ (pf0 : a :: (ls1 ++ ls2) ++ ls3 = a :: ls1 ++ ls2 ++ ls3)
```

```
    (pf'0 : (ls1 ++ ls2) ++ ls3 = ls1 ++ ls2 ++ ls3)
    (f : fhlist B ((ls1 ++ ls2) ++ ls3)),
  (a0,
  match pf'0 in (_ = ls) return (fhlist B ls) with
  | eq_refl ⇒ f
  end) =
  match pf0 in (_ = ls) return (fhlist B ls) with
  | eq_refl ⇒ (a0, f)
  end
```

To an experienced dependent types hacker, the appearance of this goal term calls for a celebration. The formula has a critical property that indicates that our problems are over. To get the proofs into the right form to apply UIP_refl, we need to use associativity of list append to rewrite their types. We could not do so before because other parts of the goal require the proofs to retain their original types. In particular, the call to fhapp that we generalized must have type $(ls1 \mathbin{+\!\!+} ls2) \mathbin{+\!\!+} ls3$, for some values of the list variables. If we rewrite the type of the proof used to type-cast this value to something like $ls1 \mathbin{+\!\!+} ls2 \mathbin{+\!\!+} ls3 = ls1 \mathbin{+\!\!+} ls2 \mathbin{+\!\!+} ls3$, then the left side of the equality would no longer match the type of the term we are trying to cast.

However, now that we have generalized over the fhapp call, the type of the term being type-cast appears explicitly in the goal and *may be rewritten as well*. In particular, the final masterstroke is rewriting everywhere in the goal using associativity of list append.

```
rewrite app_assoc.
```

```
============================
 ∀ (pf0 : a :: ls1 ++ ls2 ++ ls3 = a :: ls1 ++ ls2 ++ ls3)
    (pf'0 : ls1 ++ ls2 ++ ls3 = ls1 ++ ls2 ++ ls3)
    (f : fhlist B (ls1 ++ ls2 ++ ls3)),
  (a0,
  match pf'0 in (_ = ls) return (fhlist B ls) with
  | eq_refl ⇒ f
  end) =
  match pf0 in (_ = ls) return (fhlist B ls) with
  | eq_refl ⇒ (a0, f)
  end
```

We have achieved the crucial property: the type of each generalized equality proof has syntactically equal operands. This makes it easy to finish the proof with UIP_refl.

```
    intros.
    rewrite (UIP_refl _ _ pf0).
    rewrite (UIP_refl _ _ pf'0).
    reflexivity.
  Qed.
End fhapp.

Implicit Arguments fhapp [A B ls1 ls2].
```

This proof strategy was cumbersome and unorthodox, from the perspective of mainstream mathematics. The next section explores an alternative that leads to simpler developments in some cases.

10.4 Heterogeneous Equality

There is another equality predicate, defined in the JMeq module of the standard library, implementing *heterogeneous equality*.

```
Print JMeq.

Inductive JMeq (A : Type) (x : A) : ∀ B : Type, B → Prop :=
    JMeq_refl : JMeq x x
```

The identifier JMeq stands for *John Major equality*, a name coined by Conor McBride [23] as an inside joke about British politics. The definition JMeq starts out looking a lot like the definition of **eq**. The crucial difference is that we may use JMeq *on arguments of different types*. For instance, a lemma that we failed to establish before is trivial with JMeq. It makes for prettier theorem statements to define some syntactic shorthand first.

```
Infix "==" := JMeq (at level 70, no associativity).

Definition UIP_refl' (A : Type) (x : A) (pf : x = x)
  : pf == eq_refl x :=
  match pf return (pf == eq_refl _) with
    | eq_refl ⇒ JMeq_refl _
  end.
```

There is no quick way to write such a proof by tactics, but the underlying proof term that we want is trivial.

Suppose that we want to use UIP_refl' to establish another lemma of the kind seen earlier.

```
Lemma lemma4 : ∀ (A : Type) (x : A) (pf : x = x),
  O = match pf with eq_refl ⇒ O end.
  intros; rewrite (UIP_refl' pf); reflexivity.
```

```
Qed.
```

This seems straightforward, but the `rewrite` is implemented in terms of an axiom.

```
Check JMeq_eq.
```

```
JMeq_eq
     : ∀ (A : Type) (x y : A), x == y → x = y
```

We cannot prove that heterogeneous equality implies normal equality. The difficulties are based on limitations of `match` annotations. The *JMeq_eq* axiom has been proved consistent, but asserting it may still be considered to complicate the logic, so there is some motivation for avoiding it.

We can redo the fhapp associativity proof based on JMeq.

```
Section fhapp'.
  Variable A : Type.
  Variable B : A → Type.
```

This time, the naïve theorem statement type-checks.

```
  Theorem fhapp_assoc' : ∀ ls1 ls2 ls3 (hls1 : fhlist B ls1)
    (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) == fhapp (fhapp hls1 hls2) hls3.
    induction ls1; crush.
```

Even better, *crush* discharges the first subgoal automatically. The second subgoal is

```
============================
 (a0, fhapp b (fhapp hls2 hls3)) == (a0, fhapp (fhapp b hls2) hls3)
```

It looks like one rewrite with the inductive hypothesis should be enough to make the goal trivial. Here is what happens when we try that in Coq 8.2:

```
    rewrite IHls1.

Error: Impossible to unify
"fhlist B ((ls1 ++ ?1572) ++ ?1573)" with
"fhlist B (ls1 ++ ?1572 ++ ?1573)"
```

Coq 8.4 currently gives an error message about an uncaught exception. Perhaps that will be fixed soon. In any case, it is educational to consider a more explicit approach.

We see that JMeq is not a silver bullet. We can use it to simplify the statements of equality facts, but the Coq type checker uses nontrivial heterogeneous equality facts no more readily than it uses standard

equality facts. Here, the problem is that the form (*e1*, *e2*) is syntactic sugar for an explicit application of a constructor of an inductive type. That application mentions the type of each tuple element explicitly, and `rewrite` tries to change one of those elements without updating the corresponding type argument.

We can get around this problem by another multiple use of `generalize`. We want to bring into the goal the proper instance of the inductive hypothesis and also generalize the two relevant uses of fhapp.

```
generalize (fhapp b (fhapp hls2 hls3))
  (fhapp (fhapp b hls2) hls3)
  (IHls1 _ _ b hls2 hls3).
```

```
============================
 ∀ (f : fhlist B (ls1 ++ ls2 ++ ls3))
   (f0 : fhlist B ((ls1 ++ ls2) ++ ls3)),
   f == f0 → (a0, f) == (a0, f0)
```

Now we can rewrite with append associativity, as before.

```
rewrite app_assoc.
```

```
============================
 ∀ f f0 : fhlist B (ls1 ++ ls2 ++ ls3), f == f0
   → (a0, f) == (a0, f0)
```

From this point, the goal is trivial.

```
intros f f0 H; rewrite H; reflexivity.
Qed.
```

```
End fhapp'.
```

This example illustrates a general pattern: heterogeneous equality often simplifies theorem statements, but we still need to line up some dependent pattern matches that tactics will generate for us.

The proof we have found relies on the *JMeq_eq* axiom, which we can verify with a command `Print Assumptions`.

```
Print Assumptions fhapp_assoc'.
```

```
Axioms:
JMeq_eq : ∀ (A : Type) (x y : A), x == y → x = y
```

It was the `rewrite` H tactic that implicitly appealed to the axiom. By restructuring the proof, we can avoid axiom dependence. A general lemma about pairs provides the key element. (The use of `generalize` can be thought of as reducing the proof to another, more complex and specialized lemma.)

```
Lemma pair_cong : ∀ A1 A2 B1 B2 (x1 : A1) (x2 : A2)
  (y1 : B1) (y2 : B2),
  x1 == x2
  → y1 == y2
  → (x1, y1) == (x2, y2).
  intros until y2; intros Hx Hy; rewrite Hx; rewrite Hy;
    reflexivity.
Qed.

Hint Resolve pair_cong.

Section fhapp''.
  Variable A : Type.
  Variable B : A → Type.

  Theorem fhapp_assoc'' : ∀ ls1 ls2 ls3 (hls1 : fhlist B ls1)
    (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) == fhapp (fhapp hls1 hls2) hls3.
    induction ls1; crush.
  Qed.
End fhapp''.

Print Assumptions fhapp_assoc''.

Closed under the global context
```

One might wonder exactly which elements of a proof involving JMeq imply that *JMeq_eq* must be used. For instance, `rewrite` brought *JMeq_eq* into the proof of fhapp_assoc', yet we also used `rewrite` with JMeq hypotheses while avoiding axioms. One illuminating exercise is comparing the types of the lemmas that `rewrite` uses to implement the rewrites. Here is the normal lemma for **eq** rewriting:

```
Check eq_ind_r.

eq_ind_r
     : ∀ (A : Type) (x : A) (P : A → Prop),
       P x → ∀ y : A, y = x → P y
```

The corresponding lemma used for JMeq in the proof of pair_cong is internal_JMeq_rew_r, which, confusingly, is defined by `rewrite` as needed, so it is not available for checking until after we apply it.

```
Check internal_JMeq_rew_r.

internal_JMeq_rew_r
     : ∀ (A : Type) (x : A) (B : Type) (b : B)
         (P : ∀ B0 : Type, B0 → Type), P B b → x == b → P A x
```

The key difference is that whereas the **eq** lemma is parameterized on a predicate of type $A \rightarrow$ `Prop`, the JMeq lemma is parameterized on a predicate of type more like $\forall\ A : \texttt{Type},\ A \rightarrow \texttt{Prop}$. To apply eq_ind_r with a proof of $x = y$, it is only necessary to rearrange the goal into an application of a `fun` abstraction to y. In contrast, to apply internal_JMeq_rew_r, it is necessary to rearrange the goal to an application of a `fun` abstraction to both y and *its type*. In other words, the predicate must be *polymorphic* in y's type; any type must make sense from a type-checking standpoint. There may be cases where the former rearrangement is easy to do in a type-correct way, but the second rearrangement done naïvely leads to a type error.

When `rewrite` cannot figure out how to apply internal_JMeq_rew_r for $x == y$ where x and y have the same type, the tactic can instead use an alternative theorem, which is easy to prove as a composition of eq_ind_r and *JMeq_eq*.

```
Check JMeq_ind_r.

JMeq_ind_r
     : ∀ (A : Type) (x : A) (P : A → Prop),
       P x → ∀ y : A, y == x → P y
```

Ironically, where in the proof of fhapp_assoc' we used `rewrite` app_assoc to make it clear that a use of JMeq was actually homogeneously typed, we created a situation where `rewrite` applied the axiom-based JMeq_ind_r instead of the axiom-free internal_JMeq_rew_r.

For another simple example, consider this theorem, which applies a heterogeneous equality to prove a congruence fact:

```
Theorem out_of_luck : ∀ n m : nat,
  n == m
  → S n == S m.
  intros n m H.
```

Applying JMeq_ind_r is easy, as the `pattern` tactic transforms the goal into an application of an appropriate `fun` to a term that we want to abstract. (In general, `pattern` abstracts over a term by introducing a new anonymous function taking that term as argument.)

```
  pattern n.

  n : nat
  m : nat
  H : n == m
  ============================
   (fun n0 : nat ⇒ S n0 == S m) n
```

apply JMeq_ind_r with $(x := m)$; auto.

However, we run into trouble trying to get the goal into a form compatible with internal_JMeq_rew_r.

Undo 2.

pattern **nat**, n.

```
Error: The abstracted term
"fun (P : Set) (n0 : P) => S n0 == S m"
is not well typed.
Illegal application (Type Error):
The term "S" of type "nat -> nat"
cannot be applied to the term
 "n0" : "P"
This term has type "P" which should be coercible to
"nat".
```

In other words, the successor function S is insufficiently polymorphic. If we try to generalize over the type of n, we find that S is no longer legal to apply to n.

Abort.

Why did we not run into this problem in the proof of fhapp_assoc''? The reason is that the pair constructor is polymorphic in the types of the pair components, whereas functions like S are not polymorphic at all. Use of such nonpolymorphic functions with JMeq tends to push toward use of axioms. The example with **nat** here is a bit unrealistic; more likely cases would involve functions that have *some* polymorphism but not enough to allow abstractions of the sort we attempted with pattern. For instance, we might have an equality between two lists, where the goal only type-checks when the terms involved really are lists, though everything is polymorphic in the types of list data elements. The Heq[6] library builds up a slightly different foundation to help avoid such problems.

10.5 Equivalence of Equality Axioms

Assuming axioms (like axiom K and *JMeq_eq*) is a hazardous business. The due diligence associated with it is necessarily global in scope, since two axioms may be consistent alone but inconsistent together. It turns

6. http://www.mpi-sws.org/~gil/Heq/

out that all the major axioms proposed for reasoning about equality in Coq are logically equivalent, so we only need to pick one to assert without proof. This section shows how each of the two previous approaches reduces to the other logically.

To show that JMeq and its axiom let us prove UIP_refl, we start from the lemma UIP_refl'. The rest of the proof is trivial.

```
Lemma UIP_refl'' : ∀ (A : Type) (x : A) (pf : x = x), pf = eq_refl x.
  intros; rewrite (UIP_refl' pf); reflexivity.
Qed.
```

The other direction is perhaps more interesting. Assume that we only have the axiom of the Eqdep module available. We can define JMeq in a way that satisfies the same interface as the combination of the JMeq module's inductive definition and axiom.

```
Definition JMeq' (A : Type) (x : A) (B : Type) (y : B) : Prop :=
  ∃ pf : B = A, x = match pf with eq_refl ⇒ y end.

Infix "===" := JMeq' (at level 70, no associativity).
```

By definition, x and y are equal if and only if there exists a proof pf that their types are equal, such that x equals the result of casting y with pf. This statement seems odd from the standpoint of classical math, where proofs are rarely mentioned explicitly with quantifiers in formulas, but it is perfectly legal Coq code.

We can easily prove a theorem with the same type as that of the JMeq_refl constructor of JMeq.

```
Theorem JMeq_refl' : ∀ (A : Type) (x : A), x === x.
  intros; unfold JMeq'; exists eq_refl; reflexivity.
Qed.
```

The proof of an analogue to *JMeq_eq* is a little more interesting, but most of the action is in appealing to UIP_refl.

```
Theorem JMeq_eq' : ∀ (A : Type) (x y : A),
  x === y → x = y.
  unfold JMeq'; intros.


  H : ∃ pf : A = A,
        x = match pf in (_ = T) return T with
            | eq_refl ⇒ y
            end
  ============================
   x = y
```

```
  destruct H.

  x0 : A = A
  H : x = match x0 in (_ = T) return T with
          | eq_refl ⇒ y
          end
  ============================
   x = y

  rewrite H.

  x0 : A = A
  ============================
   match x0 in (_ = T) return T with
   | eq_refl ⇒ y
   end = y

  rewrite (UIP_refl _ _ x0); reflexivity.
Qed.
```

In a very formal sense, we are free to switch back and forth between the two styles of proofs about equality proofs. One style may be more convenient than the other for some proofs, but we can always interconvert between the results. The style that does not use heterogeneous equality may be preferable in cases where many results do not require the techniques explored in this chapter, since then the use of axioms is avoided altogether for the simple cases, and a wider audience will be able to follow those proofs. On the other hand, heterogeneous equality often makes for shorter and more readable theorem statements.

10.6 Equality of Functions

The following seems like a reasonable theorem to want to hold, and it does hold in set theory.

```
Theorem two_funs : (fun n ⇒ n) = (fun n ⇒ n + 0).
```

Unfortunately, this theorem is not provable in CIC without additional axioms. None of the definitional equality rules force function equality to be *extensional*. That is, the fact that two functions return equal results on equal inputs does not imply that the functions are equal. We can assert function extensionality as an axiom, and indeed the standard library already contains that axiom.

```
Require Import FunctionalExtensionality.
About functional_extensionality.

functional_extensionality :
∀ (A B : Type) (f g : A → B), (∀ x : A, f x = g x) → f = g
```

This axiom has been verified metatheoretically to be consistent with CIC and the two equality axioms considered previously. With it, the proof of two_funs is trivial.

```
Theorem two_funs : (fun n ⇒ n) = (fun n ⇒ n + 0).
  apply functional_extensionality; crush.
Qed.
```

The same axiom can help us prove equality of types, where we need to reason under quantifiers.

```
Theorem forall_eq : (∀ x : nat, match x with
                                | O ⇒ True
                                | S _ ⇒ True
                                end)
  = (∀ _ : nat, True).
```

There are no immediate opportunities to apply functional_extensionality, but we can use change to fix that problem.

```
  change ((∀ x : nat, (fun x ⇒ match x with
                               | 0 ⇒ True
                               | S _ ⇒ True
                               end) x) = (nat → True)).
  rewrite (functional_extensionality (fun x ⇒ match x with
                                              | 0 ⇒ True
                                              | S _ ⇒ True
                                              end)
                                     (fun _ ⇒ True)).
```

```
2 subgoals

  ============================
   (nat → True) = (nat → True)

subgoal 2 is:
 ∀ x : nat, match x with
            | 0 ⇒ True
            | S _ ⇒ True
            end = True
```

```
  reflexivity.

  destruct x; constructor.
Qed.
```

Unlike in the case of *eq_rect_eq*, we have no way of deriving this axiom of *functional extensionality* for types with decidable equality. To allow equality reasoning without axioms, it may be worth rewriting a development to replace functions with other representations, such as finite map types for which extensionality is derivable in CIC.

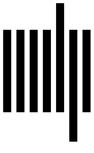
Certified Programming with Dependent Types

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

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