12 Universes and Axioms

Many traditional theorems can be proved in Coq without special knowledge of the Calculus of Inductive Constructions (CIC), the logic behind the prover. A development just seems to be using a particular ASCII notation for standard formulas based on set theory. Nonetheless, as noted in Chapter 4, CIC differs from set theory in starting from fewer orthogonal primitives. It is possible to define the usual logical connectives as derived notions. The foundation is a dependently typed functional programming language, based on dependent function types and inductive type families. By using the facilities of this language directly, we can accomplish some things much more easily than in mainstream math.

Gallina, which adds features to the more theoretical CIC [31], is the logic implemented in Coq. It has a relatively simple foundation that can be defined rigorously in a page or two of formal proof rules. Still, there are some important subtleties with practical ramifications. This chapter focuses on those subtleties, avoiding formal metatheory in favor of example code.

12.1 The `Type` Hierarchy

Every object in Gallina has a type.

```
Check 0.
```

```
0
    : nat
```

It is natural enough that zero be considered a natural number.

```
Check nat.
```

```
nat
    : Set
```

As in set theory, we consider the natural numbers as a set.

```
Check Set.
```

```
Set
    : Type
```

The type `Set` may be considered to be the set of all sets, a concept that set theory handles in terms of *classes*. In Coq, this more general notion is `Type`.

```
Check Type.
```

```
Type
    : Type
```

Strangely enough, `Type` appears to be its own type. It is known that polymorphic languages with this property are inconsistent, via Girard's paradox [8]. That is, using such a language to encode proofs is unwise, because it is possible to prove any proposition. What is really going on here?

Let us repeat some of our queries after toggling a flag related to Coq's printing behavior.

```
Set Printing Universes.
```

```
Check nat.
```

```
nat
    : Set
```

```
Check Set.
```

```
Set
    : Type (* (0)+1 *)
```

```
Check Type.
```

```
Type (* Top.3 *)
    : Type (* (Top.3)+1 *)
```

Occurrences of `Type` are annotated with some additional information inside comments. These annotations have to do with the fact that `Type` stands for an infinite hierarchy of types. The type of `Set` is `Type`(0), the type of `Type`(0) is `Type`(1), the type of `Type`(1) is `Type`(2), and so on. This is how we avoid the `Type` : `Type` paradox. As a convenience, the universe hierarchy drives Coq's one variety of subtyping. Any term whose type is `Type` at level i is automatically also described by `Type` at level j when $j > i$.

In the output of the first Check query, we see that the type level of Set's type is (0)+1. Here 0 stands for the level of Set, and we increment it to arrive at the level that *classifies* Set.

In the third query's output, we see that the occurrence of Type that we check is assigned a fresh *universe variable* *Top*.3. The output type increments *Top*.3 to move up a level in the universe hierarchy. As we write code that uses definitions whose types mention universe variables, unification may refine the values of those variables. Luckily, the user rarely has to worry about the details.

Another crucial concept in CIC is *predicativity*. Consider the following queries.

```
Check ∀ T : nat, fin T.

  ∀ T : nat, fin T
    : Set

Check ∀ T : Set, T.

  ∀ T : Set, T
    : Type (* max(0, (0)+1) *)

Check ∀ T : Type, T.

  ∀ T : Type (* Top.9 *) , T
    : Type (* max(Top.9, (Top.9)+1) *)
```

These outputs demonstrate the rule for determining which universe a ∀ type lives in. In particular, for a type ∀ *x* : *T1*, *T2*, we take the maximum of the universes of *T1* and *T2*. In the first example query, both *T1* (**nat**) and *T2* (**fin** *T*) are in Set, so the ∀ type is in Set, too. In the second query, *T1* is Set, which is at level (0)+1; and *T2* is *T*, which is at level 0. Thus, the ∀ exists at the maximum of these two levels. The third example illustrates the same outcome, where we replace Set with an occurrence of Type that is assigned universe variable *Top*.9. This universe variable appears in the places where 0 appeared in the previous query.

The behind-the-scenes manipulation of universe variables gives us predicativity. Consider a simple definition of a polymorphic identity function, where the first argument *T* will automatically be marked as implicit, since it can be inferred from the type of the second argument *x*.

```
Definition id (T : Set) (x : T) : T := x.

Check id 0.
```

```
id 0
    : nat
```

```
Check id Set.
```

```
Error: Illegal application (Type Error):
...
The 1st term has type "Type (* (Top.15)+1 *)"
which should be coercible to "Set".
```

The parameter T of id must be instantiated with a Set. The type **nat** is a Set, but Set is not. We can try fixing the problem by generalizing the definition of id.

```
Reset id.
Definition id (T : Type) (x : T) : T := x.
Check id 0.
```

```
id 0
    : nat
```

```
Check id Set.
```

```
id Set
    : Type (* Top.17 *)
```

```
Check id Type.
```

```
id Type (* Top.18 *)
    : Type (* Top.19 *)
```

So far, so good. As we apply id to different T values, the inferred index for T's Type occurrence automatically moves higher up the type hierarchy.

```
Check id id.
```

```
Error: Universe inconsistency
(cannot enforce Top.16 < Top.16).
```

This error message reminds us that the universe variable for T still exists, even though it is usually hidden. To apply id to itself, that variable would need to be less than itself in the type hierarchy. Universe inconsistency error messages announce cases like this, where a term could only type-check by violating an implied constraint over universe variables. Such errors demonstrate that Type is *predicative*; this word has a CIC meaning closely related to its usual mathematical meaning. A predicative system enforces the constraint that when an object is

defined using some sort of quantifier, none of the quantifiers may ever be instantiated with the object itself. Impredicativity is associated with popular paradoxes in set theory, involving inconsistent constructions like "the set of all sets that do not contain themselves" (Russell's paradox). Similar paradoxes would result from uncontrolled impredicativity in Coq.

12.1.1 Inductive Definitions

Predicativity restrictions also apply to inductive definitions. As an example, let us consider a type of expression trees that allows injection of any native Coq value. The idea is that an **exp** T stands for an encoded expression of type T.

```
Inductive exp : Set → Set :=
| Const : ∀ T : Set, T → exp T
| Pair : ∀ T1 T2, exp T1 → exp T2 → exp (T1 × T2)
| Eq : ∀ T, exp T → exp T → exp bool.
```

```
Error: Large non-propositional inductive types must be in
Type.
```

This definition is *large* in the sense that at least one of its constructors takes an argument whose type has type `Type`. Coq would be inconsistent if we allowed definitions like this in their full generality. Instead, we must change **exp** to live in `Type` and move **exp**'s index to `Type` as well.

```
Inductive exp : Type → Type :=
| Const : ∀ T, T → exp T
| Pair : ∀ T1 T2, exp T1 → exp T2 → exp (T1 × T2)
| Eq : ∀ T, exp T → exp T → exp bool.
```

Note that we originally had to include an annotation : `Set` for the variable T in Const's type, but we need no annotation now. When the type of a variable is not known, and when that variable is used in a context where only types are allowed, Coq infers that the variable is of type `Type`, the right behavior here, though it was wrong for the `Set` version of **exp**.

The new definition is accepted. We can build some sample expressions.

```
Check Const 0.
```

```
Const 0
    : exp nat
```

```
Check Pair (Const 0) (Const tt).
```

```
  Pair (Const 0) (Const tt)
    : exp (nat × unit)
```

```
Check Eq (Const Set) (Const Type).
```

```
  Eq (Const Set) (Const Type (* Top.59 *) )
    : exp bool
```

We can check many expressions, including complex expressions that include types. However, it is not hard to hit a type-checking wall.

```
Check Const (Const O).
```

```
Error: Universe inconsistency
(cannot enforce Top.42 < Top.42).
```

We are unable to instantiate the parameter T of Const with an **exp** type. To see why, it is helpful to print the annotated version of **exp**'s inductive definition.

```
Print exp.
```

```
Inductive exp
                : Type (* Top.8 *) →
                  Type
                  (* max(0, (Top.11)+1, (Top.14)+1, (Top.15)+1,
                        (Top.19)+1) *) :=
  Const : ∀ T : Type (* Top.11 *) , T → exp T
| Pair : ∀ (T1 : Type (* Top.14 *) ) (T2 : Type (* Top.15 *) ),
         exp T1 → exp T2 → exp (T1 × T2)
| Eq : ∀ T : Type (* Top.19 *) , exp T → exp T → exp bool
```

We see that the index type of **exp** has been assigned to universe level *Top*.8. In addition, each of the four occurrences of Type in the types of the constructors gets its own universe variable. Each of these variables appears explicitly in the type of **exp**. In particular, any type **exp** T lives at a universe level found by incrementing by one the maximum of the four argument variables. Therefore, **exp** *must* live at a higher universe level than any type which may be passed to one of its constructors. This consequence led to the universe inconsistency.

Strangely, the universe variable *Top*.8 only appears in one place. Is there no restriction imposed on which types are valid arguments to **exp**? In fact, there is a restriction, but it only appears in a global set of universe constraints that are maintained but do not appear explicitly in types. We can print the current database.

```
Print Universes.
```

$Top.19 < Top.9 \leq Top.8$
$Top.15 < Top.9 \leq Top.8 \leq$ Coq.Init.Datatypes.38
$Top.14 < Top.9 \leq Top.8 \leq$ Coq.Init.Datatypes.37
$Top.11 < Top.9 \leq Top.8$

The command outputs many more constraints, but we have collected only those that mention *Top* variables. We see one constraint for each universe variable associated with a constructor argument from **exp**'s definition. Universe variable *Top*.19 is the type argument to Eq. The constraint for *Top*.19 effectively says that *Top*.19 must be less than *Top*.8, the universe of **exp**'s indices; an intermediate variable *Top*.9 appears as an artifact of the way the constraint was generated.

The next constraint, for *Top*.15, is more complicated. This is the universe of the second argument to the Pair constructor. Not only must *Top*.15 be less than *Top*.8 but it is also evident that *Top*.8 must be no greater than Coq.Init.Datatypes.38. What is this new universe variable? It is from the definition of the **prod** inductive family, to which types of the form $A \times B$ are desugared.

```
Print prod.
```

```
Inductive prod (A : Type (* Coq.Init.Datatypes.37 *) )
          (B : Type (* Coq.Init.Datatypes.38 *) )
            : Type (* max(Coq.Init.Datatypes.37,
                          Coq.Init.Datatypes.38) *) :=
    pair : A → B → A × B
```

We see that the constraint is enforcing that indices to **exp** must not live in a higher universe level than B indices to **prod**. The next constraint establishes a symmetric condition for A.

Thus, it is apparent that Coq maintains an elaborate set of universe variable inequalities behind the scenes. It may look like some functions are polymorphic in the universe levels of their arguments, but what is really happening is imperative updating of a system of constraints, such that all uses of a function are consistent with a global set of universe levels. When the constraint system may not be evolved soundly, we get a universe inconsistency error.

The annotated definition of **prod** reveals something interesting. A type **prod** A B lives at a universe that is the maximum of the universes of A and B, though we might expect that **prod**'s universe would in fact need to be *one higher* than the maximum. The critical difference is that in the definition of **prod**, A and B are defined as *parameters*; that is,

they appear named to the left of the main colon rather than (possibly unnamed) to the right.

Parameters are not as flexible as normal inductive type arguments. The range types of all constructors of a parameterized type must share the same parameters. Nonetheless, when it is possible to define a polymorphic type in this way, we gain the ability to use the new type family in more ways, without triggering universe inconsistencies. For instance, nested pairs of types are perfectly legal.

```
Check (nat, (Type, Set)).
```

```
(nat, (Type (* Top.44 *) , Set))
    : Set × (Type (* Top.45 *) × Type (* Top.46 *) )
```

The same cannot be done with a counterpart to **prod** that does not use parameters.

```
Inductive prod' : Type → Type → Type :=
| pair' : ∀ A B : Type, A → B → prod' A B.

Check (pair' nat (pair' Type Set)).
```

```
Error: Universe inconsistency
(cannot enforce Top.51 < Top.51).
```

The key benefit that parameters bring is the ability to avoid quantifying over types in the types of constructors. Such quantification induces less-than constraints, whereas parameters only introduce less-than-or-equal-to constraints.

Coq includes one more (potentially confusing) feature related to parameters. While Gallina does not support real universe polymorphism, there is a convenience facility that mimics universe polymorphism in some cases. We can illustrate what this means with a simple example.

```
Inductive foo (A : Type) : Type :=
| Foo : A → foo A.

Check foo nat.
```

```
foo nat
    : Set
```

```
Check foo Set.
```

```
foo Set
    : Type
```

```
Check foo True.
```

```
foo True
    : Prop
```

The basic pattern here is that Coq is willing to automatically build a copied-and-pasted version of an inductive definition, where some occurrences of `Type` have been replaced by `Set` or `Prop`. In each context, the type checker tries to find the valid replacements that are lowest in the type hierarchy. Automatic cloning of definitions can be much more convenient than manual cloning. We have already taken advantage of the fact that we may reuse the same families of tuple and list types to form values in `Set` and `Type`.

Imitation polymorphism can be confusing in some contexts. For instance, it is responsible for this odd behavior.

```
Inductive bar : Type := Bar : bar.

Check bar.

bar
    : Prop
```

The type that Coq comes up with may be used in strictly more contexts than the type one might have expected.

12.1.2 Deciphering Baffling Messages about Inability to Unify

One of the most confusing sorts of Coq error messages arises from an interplay between universes, syntax notations, and implicit arguments. Consider the following innocuous lemma, which is symmetry of equality for the special case of types.

```
Theorem symmetry : ∀ A B : Type,
  A = B
  → B = A.
  intros ? ? H; rewrite H; reflexivity.
Qed.
```

Let us attempt an admittedly silly proof of the following theorem.

```
Theorem illustrative_but_silly_detour : unit = unit.
  apply symmetry.

Error: Impossible to unify "?35 = ?34" with "unit = unit".
```

Coq tells us that we cannot apply the lemma `symmetry` here, but the error message seems defective. In particular, one might think that

`apply` should unify ?35 and ?34 with **unit** to ensure that the unification goes through. In fact, the issue is in a part of the unification problem that is *not* shown in this error message.

The following command is the secret to getting better error messages in such cases.

```
Set Printing All.

 apply symmetry.
```

```
Error: Impossible to unify "@eq Type ?46 ?45" with
"@eq Set unit unit".
```

Now we can see the problem: it is the first, *implicit* argument to the underlying equality function **eq** that disagrees across the two terms. The universe `Set` may be both an element and a subtype of `Type`, but the two are not definitionally equal.

```
Abort.
```

A variety of changes to the theorem statement would lead to use of `Type` as the implicit argument of **eq**. Here is one such change:

```
Theorem illustrative_but_silly_detour : (unit : Type) = unit.
  apply symmetry; reflexivity.
Qed.
```

Many related issues can come up with error messages, where one or both of notations and implicit arguments hide important details. The `Set Printing All` command turns off all such features and exposes underlying CIC terms.

For completeness, I mention one other class of confusing error messages about inability to unify two terms that look obviously unifiable. Each unification variable has a scope; a unification variable instantiation may not mention variables that were not already defined within that scope, at the point in proof search where the unification variable was introduced. Consider this illustrative example:

```
Unset Printing All.

Theorem ex_symmetry : (∃ x, x = 0) → (∃ x, 0 = x).
  eexists.

  H : ∃ x : nat, x = 0
  ============================
   0 = ?98

  destruct H.
```

```
  x : nat
  H : x = 0
  ============================
   0 = ?99

  symmetry; exact H.

Error: In environment
x : nat
H : x = 0
The term "H" has type "x = 0" while it is expected to have
type "?99 = 0".
```

The problem here is that variable x was introduced by `destruct` *after* we introduced ?99 with `eexists`, so the instantiation of ?99 may not mention x. A simple reordering of the proof solves the problem.

```
  Restart.
  destruct 1 as [x]; apply ex_intro with x; symmetry; assumption.
Qed.
```

This restriction for unification variables may seem counterintuitive, but it follows from the fact that CIC contains no concept of unification variable. Rather, to construct the final proof term, at the point in a proof where the unification variable is introduced, we replace it with the instantiation we eventually find for it. It is simply syntactically illegal to refer there to variables that are not in scope. Without such a restriction, we could trivially prove such nontheorems as $\exists\ n : \mathsf{nat}, \forall\ m : \mathsf{nat}, n = m$ by `econstructor; intro; reflexivity`.

12.2 The Prop Universe

Chapter 3 showed parallel versions of useful datatypes for programs and proofs. The convention was that programs live in `Set`, and proofs live in `Prop`. Little explanation was given for why it is useful to maintain this distinction. There is certainly documentation value from separating programs from proofs; in practice, different concerns apply to building the two types of objects. It turns out, however, that these concerns motivate formal differences between the two universes in Coq.

Recall the types **sig** and **ex**, which are the program and proof versions of existential quantification. Their definitions differ only in one place, where **sig** uses `Type` and **ex** uses `Prop`.

```
Print sig.
```

```
  Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ x : A, P x → sig P

Print ex.

  Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P
```

It is natural to want a function to extract the first components of data structures like these. Doing so is easy enough for **sig**.

```
Definition projS A (P : A → Prop) (x : sig P) : A :=
  match x with
    | exist v _ ⇒ v
  end.
```

We run into trouble with a version that has been changed to work with **ex**.

```
Definition projE A (P : A → Prop) (x : ex P) : A :=
  match x with
    | ex_intro v _ ⇒ v
  end.

Error:
Incorrect elimination of "x" in the inductive type "ex":
the return type has sort "Type" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Type
because proofs can be eliminated only to build proofs.
```

In formal Coq parlance, *elimination* means pattern matching. The typing rules of Gallina forbid pattern matching on a discriminee whose type belongs to `Prop`, whenever the result type of the `match` has a type besides `Prop`. This is a sort of information flow policy, where the type system ensures that the details of proofs can never have any effect on parts of a development that are not also marked as proofs.

This restriction matches informal practice. We think of programs and proofs as clearly separated, and outside of constructive logic, the idea of computing with proofs is ill-formed. The distinction also has practical importance in Coq, where it affects the behavior of extraction.

Recall that extraction is Coq's facility for translating Coq developments into programs in general-purpose programming languages like OCaml. Extraction *erases* proofs and leaves programs intact. A simple example with **sig** and **ex** demonstrates the distinction.

```
Definition sym_sig (x : sig (fun n ⇒ n = 0)) : sig (fun n ⇒ 0 = n) :=
```

```
  match x with
    | exist n pf ⇒ exist _ n (sym_eq pf)
  end.

Extraction sym_sig.

(** val sym_sig : nat -> nat **)

let sym_sig x = x
```

Since extraction erases proofs, the second components of **sig** values are elided, making **sig** a simple identity type family. The sym_sig operation is thus an identity function.

```
Definition sym_ex (x : ex (fun n ⇒ n = 0)) : ex (fun n ⇒ 0 = n) :=
  match x with
    | ex_intro n pf ⇒ ex_intro _ n (sym_eq pf)
  end.

Extraction sym_ex.

(** val sym_ex : __ **)

let sym_ex = __
```

In this example, the **ex** type itself is in Prop, so whole **ex** packages are erased. Coq extracts every proposition as the (Coq-specific) type __, whose single constructor is __. Not only are proofs replaced by __ but proof arguments to functions are also removed completely, as we see here.

Extraction is very helpful as an optimization over programs that contain proofs. In languages like Haskell, advanced features make it possible to program with proofs, as a way of convincing the type checker to accept particular definitions. Unfortunately, when proofs are encoded as values in GADTs [50], these proofs exist at run-time and consume resources. In contrast, with Coq, as long as all proofs are kept within Prop, extraction is guaranteed to erase them.

Many users of the Curry-Howard correspondence support the idea of *extracting programs from proofs*, but few users of Coq and related tools do this. Instead, extraction is better thought of as an optimization that reduces the run-time costs of expressive typing.

We have seen two differences between proofs and programs: proofs are subject to an elimination restriction and are elided by extraction. The remaining difference is that Prop is *impredicative*, as the following example shows.

```
Check ∀ P Q : Prop, P ∨ Q → Q ∨ P.
```

```
∀ P Q : Prop, P ∨ Q → Q ∨ P
    : Prop
```

We see that it is possible to define a `Prop` that quantifies over other `Prop`s. This is fortunate, as we want that ability even for such basic purposes as stating propositional tautologies. The next section of this chapter explains why unrestricted impredicativity is undesirable. The impredicativity of `Prop` interacts crucially with the elimination restriction to avoid those pitfalls.

Impredicativity also allows us to implement a version of our earlier **exp** type that does not suffer from the previous weakness.

```
Inductive expP : Type → Prop :=
| ConstP : ∀ T, T → expP T
| PairP : ∀ T1 T2, expP T1 → expP T2 → expP (T1 × T2)
| EqP : ∀ T, expP T → expP T → expP bool.

Check ConstP 0.
```

```
ConstP 0
    : expP nat
```

```
Check PairP (ConstP 0) (ConstP tt).
```

```
PairP (ConstP 0) (ConstP tt)
    : expP (nat × unit)
```

```
Check EqP (ConstP Set) (ConstP Type).
```

```
EqP (ConstP Set) (ConstP Type)
    : expP bool
```

```
Check ConstP (ConstP O).
```

```
ConstP (ConstP 0)
    : expP (expP nat)
```

The new definition is not very helpful in this case. Because we have marked **expP** as a family of proofs, we cannot deconstruct expressions in the usual programmatic ways, which makes them almost useless for the usual purposes. Impredicative quantification is much more useful in defining inductive families that we really think of as judgments. For instance, the following code defines a notion of equality that is strictly more permissive than the base equality =.

```
Inductive eqPlus : ∀ T, T → T → Prop :=
| Base : ∀ T (x : T), eqPlus x x
```

```
| Func : ∀ dom ran (f1 f2 : dom → ran),
  (∀ x : dom, eqPlus (f1 x) (f2 x))
  → eqPlus f1 f2.

Check (Base 0).

  Base 0
     : eqPlus 0 0

Check (Func (fun n ⇒ n) (fun n ⇒ 0 + n) (fun n ⇒ Base n)).

  Func (fun n : nat ⇒ n) (fun n : nat ⇒ 0 + n)
  (fun n : nat ⇒ Base n)
     : eqPlus (fun n : nat ⇒ n) (fun n : nat ⇒ 0 + n)

Check (Base (Base 1)).

  Base (Base 1)
     : eqPlus (Base 1) (Base 1)
```

For a sense of why a term like the preceding one is useful to write, recall that we have already seen in Chapter 10 the utility of stating equality facts about proofs.

12.3 Axioms

While the specific logic Gallina is hardcoded into Coq's implementation, it is possible to add certain logical rules in a controlled way. In other words, Coq may be used to reason about many different refinements of Gallina where strictly more theorems are provable. We achieve this by asserting *axioms* without proof.

I tour through some standard axioms, as enumerated in Coq's online FAQ, and I add additional commentary as appropriate.

12.3.1 The Basics

One simple example of a useful axiom is the law of the excluded middle.

```
Require Import Classical_Prop.
Print classic.

*** [ classic : ∀ P : Prop, P ∨ ¬ P ]
```

In the implementation of module Classical_Prop, this axiom was defined with the command

```
Axiom classic : ∀ P : Prop, P ∨ ¬ P.
```

An `Axiom` may be declared with any type, in any universe. There is a synonym `Parameter` for `Axiom`, and that synonym is often clearer for assertions not of type `Prop`. For instance, we can assert the existence of objects with certain properties.

```
Parameter num : nat.
Axiom positive : num > 0.
Reset num.
```

This kind of axiomatic presentation of a theory is very common outside of higher-order logic. However, in Coq, it is almost always preferable to stick to defining objects, functions, and predicates via inductive definitions and functional programming.

In general, there is a significant burden associated with any use of axioms. It is easy to assert a set of axioms that together is *inconsistent*. That is, a set of axioms may imply **False**, which allows any theorem to be proved, which defeats the purpose of a proof assistant. For example, we could assert the following axiom, which is consistent by itself but inconsistent when combined with *classic*.

```
Axiom not_classic : ¬ ∀ P : Prop, P ∨ ¬ P.

Theorem uhoh : False.
  generalize classic not_classic; tauto.
Qed.

Theorem uhoh_again : 1 + 1 = 3.
  destruct uhoh.
Qed.

Reset not_classic.
```

On the subject of the law of the excluded middle itself, this axiom is usually quite harmless, and many practical Coq developments assume it. It has been proved metatheoretically to be consistent with CIC. Here, *proved metatheoretically* means that someone proved on paper that excluded middle holds in a *model* of CIC in set theory [48]. All the other axioms surveyed in this section hold in the same model, so they are all consistent together.

Recall that Coq implements *constructive logic* by default, where the law of the excluded middle is not provable. Proofs in constructive logic can be thought of as programs. A ∀ quantifier denotes a dependent function type, and a disjunction denotes a variant type. In such a setting, excluded middle could be interpreted as a decision procedure for arbitrary propositions, which computability theory tells us cannot exist. Thus, constructive logic with excluded middle can no longer be associated with the usual notion of programming.

Given all this, why can one assert excluded middle as an axiom? The intuitive justification is that the elimination restriction for `Prop` prevents us from treating proofs as programs. An excluded middle axiom that quantified over `Set` instead of `Prop` would be problematic. If a development used that axiom, we would not be able to extract the code to OCaml (soundly) without implementing a genuine universal decision procedure. In contrast, values whose types belong to `Prop` are always erased by extraction, so we sidestep the axiom's algorithmic consequences.

Because the proper use of axioms is so precarious, there are helpful commands for determining which axioms a theorem relies on.

```
Theorem t1 : ∀ P : Prop, P → ¬ ¬ P.
  tauto.
Qed.

Print Assumptions t1.

  Closed under the global context

Theorem t2 : ∀ P : Prop, ¬ ¬ P → P.
  tauto.

Error: tauto failed.

  intro P; destruct (classic P); tauto.
Qed.

Print Assumptions t2.

  Axioms:
  classic : ∀ P : Prop, P ∨ ¬ P
```

It is possible to avoid this dependence in some specific cases, where excluded middle is provable, for decidable families of propositions.

```
Theorem nat_eq_dec : ∀ n m : nat, n = m ∨ n ≠ m.
  induction n; destruct m; intuition; generalize (IHn m);
    intuition.
Qed.

Theorem t2' : ∀ n m : nat, ¬ ¬ (n = m) → n = m.
  intros n m; destruct (nat_eq_dec n m); tauto.
Qed.

Print Assumptions t2'.

Closed under the global context
```

Mainstream mathematical practice assumes excluded middle, so it can be useful to have it available in Coq developments, though it is also nice to know that a theorem is proved in a simpler formal system than classical logic. The same is true for *proof irrelevance*, which simplifies proof issues that would not even arise in mainstream math.

```
Require Import ProofIrrelevance.
Print proof_irrelevance.
```

```
*** [ proof_irrelevance : ∀ (P : Prop) (p1 p2 : P), p1 = p2 ]
```

This axiom asserts that any two proofs of the same proposition are equal. Recall this example function from Chapter 6.

```
Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

We might want to prove that different proofs of $n > 0$ do not lead to different results from the richly typed predecessor function.

```
Theorem pred_strong1_irrel : ∀ n (pf1 pf2 : n > 0),
  pred_strong1 pf1 = pred_strong1 pf2.
  destruct n; crush.
Qed.
```

The proof script is simple, but it involves peeking into the definition of pred_strong1. For more complicated function definitions, one might like to avoid this burden. The Prop elimination restriction makes it impossible to write any function that discriminates on details of proof arguments, but this is only true metatheoretically, unless we assert an axiom like *proof_irrelevance*. With that axiom, we can prove the theorem without consulting the definition of pred_strong1.

```
Theorem pred_strong1_irrel' : ∀ n (pf1 pf2 : n > 0),
  pred_strong1 pf1 = pred_strong1 pf2.
  intros; f_equal; apply proof_irrelevance.
Qed.
```

Chapter 10 discussed some axioms related to proof irrelevance. In particular, Coq's standard library includes this axiom:

```
Require Import Eqdep.
Import 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 ]
```

This axiom says that it is permissible to simplify pattern matches over proofs of equalities like $e = e$. The axiom is logically equivalent to some simpler corollaries. In the theorem names, UIP stands for "unicity of identity proofs," where *identity* is a synonym for *equality*.

```
Corollary UIP_refl : ∀ A (x : A) (pf : x = x), pf = eq_refl x.
  intros; replace pf with (eq_rect x (eq x) (eq_refl x) x pf); [
    symmetry; apply eq_rect_eq
    | exact (match pf as pf' return match pf' in _ = y
                                          return x = y with
                                          | eq_refl ⇒ eq_refl x
                                        end = pf' with
               | eq_refl ⇒ eq_refl _
             end) ].
Qed.

Corollary UIP : ∀ A (x y : A) (pf1 pf2 : x = y), pf1 = pf2.
  intros; generalize pf1 pf2; subst; intros;
    match goal with
      | [ ⊢ ?pf1 = ?pf2 ] ⇒ rewrite (UIP_refl pf1);
        rewrite (UIP_refl pf2); reflexivity
    end.
Qed.
```

These corollaries are special cases of proof irrelevance. In developments that only need proof irrelevance for equality, there is no need to assert full irrelevance.

Another facet of proof irrelevance is that, like excluded middle, it is often provable for specific propositions. For instance, UIP is provable whenever the type A has a decidable equality operation. The module Eqdep_dec of the standard library contains a proof. A similar phenomenon applies to other notable cases, including less-than proofs. Thus, it is often possible to use proof irrelevance without asserting axioms.

There are two more basic axioms that are often assumed, to avoid complications that do not arise in set theory.

```
Require Import FunctionalExtensionality.
Print functional_extensionality_dep.
```

```
*** [ functional_extensionality_dep :
∀ (A : Type) (B : A → Type) (f g : ∀ x : A, B x),
(∀ x : A, f x = g x) → f = g ]
```

This axiom says that two functions are equal if they map equal inputs to equal outputs. Such facts are not provable in general in CIC, but it is consistent to assume that they are.

A simple corollary shows that the same property applies to predicates.

```
Corollary predicate_extensionality : ∀ (A : Type) (B : A → Prop)
  (f g : ∀ x : A, B x),
  (∀ x : A, f x = g x) → f = g.
  intros; apply functional_extensionality_dep; assumption.
Qed.
```

In some cases, one might prefer to assert this corollary as the axiom, to restrict the consequences to proofs and not programs.

12.3.2 Axioms of Choice

Some Coq axioms are also points of contention in mainstream math. The most prominent example is the axiom of choice. In fact, there are multiple versions, and considered in isolation, none of these versions means quite what it means in classical set theory.

First, it is possible to implement a choice operator *without* axioms in some cases.

```
Require Import ConstructiveEpsilon.
Check constructive_definite_description.

  constructive_definite_description
     : ∀ (A : Set) (f : A → nat) (g : nat → A),
       (∀ x : A, g (f x) = x) →
       ∀ P : A → Prop,
       (∀ x : A, {P x} + { ¬ P x}) →
       (∃! x : A, P x) → {x : A | P x}

Print Assumptions constructive_definite_description.

Closed under the global context
```

This function transforms a decidable predicate P into a function that produces an element satisfying P from a proof that such an element

exists. The functions f and g, in conjunction with an associated injectivity property, are used to express the idea that the set A is countable. Under these conditions, a simple brute force algorithm gets the job done: we just enumerate all elements of A, stopping when we find one satisfying P. The existence proof, specified in terms of *unique* existence $\exists!$, guarantees termination. The definition of this operator in Coq uses some interesting techniques, as seen in the implementation of the ConstructiveEpsilon module.

Countable choice is provable in set theory without appealing to the general axiom of choice. To support the more general principle in Coq, we must also add an axiom. Here is a functional version of the axiom of unique choice:

```
Require Import ClassicalUniqueChoice.
Check dependent_unique_choice.
```

```
dependent_unique_choice
     : ∀ (A : Type) (B : A → Type) (R : ∀ x : A, B x → Prop),
       (∀ x : A, ∃! y : B x, R x y) →
       ∃ f : ∀ x : A, B x,
         ∀ x : A, R x (f x)
```

This axiom lets us convert a relational specification R into a function implementing that specification. We need only prove that R is truly a function. A stronger formulation applies to cases where R maps each input to one or more outputs. We also simplify the statement of the theorem by considering only nondependent function types.

```
Require Import ClassicalChoice.
Check choice.
```

```
choice
     : ∀ (A B : Type) (R : A → B → Prop),
       (∀ x : A, ∃ y : B, R x y) →
       ∃ f : A → B, ∀ x : A, R x (f x)
```

This principle is proved as a theorem, based on the unique choice axiom and an additional axiom of relational choice from the RelationalChoice module.

In set theory, the axiom of choice is a fundamental philosophical commitment one makes about the universe of sets. In Coq, the choice axioms say something weaker. For instance, consider the simple restatement of the choice axiom where we replace existential quantification by its Curry-Howard analogue, subset types.

```
Definition choice_Set (A B : Type) (R : A → B → Prop)
```

```
(H : ∀ x : A, {y : B | R x y})
: {f : A → B | ∀ x : A, R x (f x)} :=
exist (fun f ⇒ ∀ x : A, R x (f x))
(fun x ⇒ proj1_sig (H x)) (fun x ⇒ proj2_sig (H x)).
```

Via the Curry-Howard correspondence, this axiom can be taken to have the same meaning as the original. It is implemented trivially as a transformation not much deeper than uncurrying. Thus, the utility of the axioms mentioned earlier comes in their usage to build programs from proofs. Normal set theory has no explicit proofs, so the meaning of the usual axiom of choice is subtly different. In Gallina, the axioms implement a controlled relaxation of the restrictions on information flow from proofs to programs.

However, when we combine an axiom of choice with the law of the excluded middle, the idea of choice becomes more interesting. Excluded middle gives us a highly noncomputational way of constructing proofs, but it does not change the computational nature of programs. Thus, the axiom of choice can still be used to translate between two different sorts of programs, but the input programs (which are proofs) may be written in a rich language that goes beyond normal computability. This combination truly is more than repackaging a function with a different type.

An even stronger and more direct bridge between the two worlds comes from Hilbert's epsilon operator:

```
Require Import ClassicalEpsilon.
Check constructive_indefinite_description.

constructive_indefinite_description
     : ∀ (A : Type) (P : A → Prop),
       (∃ x : A, P x) → {x : A | P x}
```

This operator lets us translate directly from the `Prop` version of existential quantification into the `Set` version. With such an axiom plus others like excluded middle, computation in all of Coq's universes goes beyond the usual bounds of computability, so one should appeal to the axiom with care. Program extraction will not be feasible for functions that use this operator, but it may be useful in formalizing standard mathematical constructs programmatically, for instance, to materialize the limit of an infinite series that can be proved to converge.

The Coq tools support a command line flag `-impredicative-set`, which modifies Gallina in a more fundamental way by making `Set` impredicative. A term like ∀ T : `Set`, T has type `Set`, and inductive

definitions in `Set` may have constructors that quantify over arguments of any types. To maintain consistency, an elimination restriction must be imposed, similarly to the restriction for `Prop`. The restriction only applies to large inductive types, where some constructor quantifies over a type of type `Type`. In such cases, a value in this inductive type may only be pattern-matched over to yield a result type whose type is `Set` or `Prop`. This rule contrasts with the rule for `Prop`, where the restriction applies even to nonlarge inductive types, and where the result type may only have type `Prop`.

In old versions of Coq, `Set` was impredicative by default. Later versions make `Set` predicative to avoid inconsistency with some classical axioms. In particular, one should watch out when using impredicative `Set` with axioms of choice. In combination with excluded middle or predicate extensionality, inconsistency can result. Impredicative `Set` can be useful for modeling inherently impredicative mathematical concepts, but almost all Coq developments do fine without it.

12.3.3 Axioms and Computation

One additional axiom-related concern arises from an aspect of Gallina that is very different from set theory: a notion of *computational equivalence* is central to the definition of the formal system. Axioms tend not to play well with computation. Consider this example. We start by implementing a function that uses a type equality proof to perform a safe type cast.

```
Definition cast (x y : Set) (pf : x = y) (v : x) : y :=
  match pf with
    | eq_refl ⇒ v
  end.
```

Computation over programs that use cast can proceed smoothly.

```
Eval compute in (cast (eq_refl (nat → nat)) (fun n ⇒ S n)) 12.

    = 13
    : nat
```

Things do not go as smoothly when we use cast with proofs that rely on axioms.

```
Theorem t3 : (∀ n : nat, fin (S n)) = (∀ n : nat, fin (n + 1)).
  change ((∀ n : nat, (fun n ⇒ fin (S n)) n)
    = (∀ n : nat, (fun n ⇒ fin (n + 1)) n));
    rewrite (functional_extensionality (fun n ⇒ fin (n + 1))
      (fun n ⇒ fin (S n))); crush.
```

```
Qed.

Eval compute in (cast t3 (fun _ ⇒ First)) 12.

    = match t3 in (_ = P) return P with
      | eq_refl ⇒ fun n : nat ⇒ First
      end 12
    : fin (12 + 1)
```

Computation gets stuck in a pattern match on the proof t3. The structure of t3 is not known, so the match cannot proceed. It turns out a more basic problem leads to this particular situation. We ended the proof of t3 with Qed, so the definition of t3 is not available to computation. That mistake is easily fixed.

```
Reset t3.

Theorem t3 : (∀ n : nat, fin (S n)) = (∀ n : nat, fin (n + 1)).
  change ((∀ n : nat, (fun n ⇒ fin (S n)) n)
    = (∀ n : nat, (fun n ⇒ fin (n + 1)) n));
    rewrite (functional_extensionality (fun n ⇒ fin (n + 1))
      (fun n ⇒ fin (S n))); crush.
Defined.

Eval compute in (cast t3 (fun _ ⇒ First)) 12.

    = match
        match
          match
            functional_extensionality
    ....
```

Most of the details are elided. A very unwieldy tree of nested matches on equality proofs appears. This time evaluation really *is* stuck on a use of an axiom.

If we are careful in using tactics to prove an equality, we can still compute with casts over the proof.

```
Lemma plus1 : ∀ n, S n = n + 1.
  induction n; simpl; intuition.
Defined.

Theorem t4 : ∀ n, fin (S n) = fin (n + 1).
  intro; f_equal; apply plus1.
Defined.

Eval compute in cast (t4 13) First.

    = First
```

: **fin** (13 + 1)

This simple computational reduction hides the use of a recursive function to produce a suitable eq_refl proof term. The recursion originates in the use of `induction` in t4's proof.

12.3.4 Methods for Avoiding Axioms

The previous section demonstrated one reason to avoid axioms: they interfere with computational behavior of terms. A further reason is to reduce the philosophical commitment of a theorem. The more axioms one assumes, the harder it becomes to convince oneself that the formal system corresponds appropriately to one's intuitions. A refinement of this last point, in applications like proof-carrying code [27] in computer security, has to do with minimizing the size of a *trusted code base*. To convince ourselves that a theorem is true, we must convince ourselves of the correctness of the program that checks the theorem. Axioms effectively become new source code for the checking program, increasing the effort required to perform a correctness audit.

Section 12.3 gave one example of avoiding an axiom. We proved that pred_strong1 is agnostic to details of the proofs passed to it as arguments, by unfolding the definition of the function. A different proof keeps the function definition opaque and instead applies a proof irrelevance axiom. By accepting a more complex proof, we reduce the philosophical commitment and trusted base. (By the way, the less-than relation that the proofs here prove admits proof irrelevance as a theorem provable within normal Gallina.)

We used the *dep_destruct* tactic several times in a way that silently generates proof terms referring to an axiom. Consider this simple case analysis principle for **fin** values:

```
Theorem fin_cases : ∀ n (f : fin (S n)), f = First ∨ ∃ f', f = Next f'.
  intros; dep_destruct f; eauto.
Qed.

Print Assumptions fin_cases.

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

The proof depends on the *JMeq_eq* axiom (see Chapter 10). However, a smarter tactic could have avoided an axiom dependence. Here is a different proof via a slightly strange-looking lemma:

```
Lemma fin_cases_again' : ∀ n (f : fin n),
  match n return fin n → Prop with
```

```
    | O ⇒ fun _ ⇒ False
    | S n' ⇒ fun f ⇒ f = First ∨ ∃ f', f = Next f'
  end f.
  destruct f; eauto.
Qed.
```

We apply a variant of the convoy pattern, which we are used to seeing in function implementations. Here, the pattern helps us state a lemma in a form where the argument to **fin** is a variable. Recall that, thanks to basic typing rules for pattern matching, `destruct` will only work effectively on types whose nonparameter arguments are variables. The `exact` tactic, which takes as argument a literal proof term, now gives us an easy way of proving the original theorem.

```
Theorem fin_cases_again : ∀ n (f : fin (S n)),
  f = First ∨ ∃ f', f = Next f'.
  intros; exact (fin_cases_again' f).
Qed.

Print Assumptions fin_cases_again.
Closed under the global context
```

As the Curry-Howard correspondence might lead us to expect, the same pattern may be applied in programming as in proving. Axioms are relevant in programming, too, because, while Coq includes useful extensions like `Program` that make dependently typed programming more straightforward, in general these extensions generate code that relies on axioms about equality. We can use clever pattern matching to write code axiom-free.

As an example, consider a `Set` version of fin_cases. We use `Set` types instead of `Prop` types, so that return values have computational content and may be used to guide the behavior of algorithms. Besides that, we are essentially writing the same "proof" in a more explicit way.

```
Definition finOut n (f : fin n)
  : match n return fin n → Type with
      | O ⇒ fun _ ⇒ Empty_set
      | _ ⇒ fun f ⇒ {f' : _ | f = Next f'} + {f = First}
    end f :=
  match f with
    | First _ ⇒ inright _ eq_refl
    | Next _ f' ⇒ inleft _ (exist _ f' eq_refl)
  end.
```

As another example, consider the following type of formulas in first-order logic. The intent of the type definition is not important in what follows, but we give a quick intuition. The formulas may include $\forall$ quantification over arbitrary `Type`s, and we index formulas by environments telling which variables are in scope and what their types are; such an environment is a **list** `Type`. A constructor Inject lets us include any Coq `Prop` as a formula, and VarEq and Lift can be used for variable references, in what is essentially the de Bruijn index convention.

```
Inductive formula : list Type → Type :=
| Inject : ∀ Ts, Prop → formula Ts
| VarEq : ∀ T Ts, T → formula (T :: Ts)
| Lift : ∀ T Ts, formula Ts → formula (T :: Ts)
| Forall : ∀ T Ts, formula (T :: Ts) → formula Ts
| And : ∀ Ts, formula Ts → formula Ts → formula Ts.
```

This example is based on my own experiences implementing variants of a program logic called XCAP [28], which also includes an inductive predicate for characterizing which formulas are provable. Here I include a pared-down version of such a predicate, with only two constructors, which is sufficient to illustrate certain issues.

```
Inductive proof : formula nil → Prop :=
| PInject : ∀ (P : Prop), P → proof (Inject nil P)
| PAnd : ∀ p q, proof p → proof q → proof (And p q).
```

Let us prove a lemma showing that a "$P \wedge Q \rightarrow P$" rule is derivable within the rules of **proof**.

```
Theorem proj1 : ∀ p q, proof (And p q) → proof p.
  destruct 1.

  p : formula nil
  q : formula nil
  P : Prop
  H : P
  ============================
   proof p
```

We are reminded that `induction` and `destruct` do not work effectively on types with nonvariable arguments. The first subgoal is clearly unprovable. (Consider the case where p = Inject nil **False**.)

An application of the `dependent destruction` tactic (the basis for *dep_destruct*) solves the problem handily. We use a shorthand with the `intros` tactic that lets us use question marks for variable names that do not matter.

```
  Restart.
  Require Import Program.
  intros ? ? H; dependent destruction H; auto.
Qed.

Print Assumptions proj1.

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

Unfortunately, that built-in tactic appeals to an axiom. It is still possible to avoid axioms by giving the proof via another lemma. Here is a first attempt that fails at remaining axiom-free, using a common equality-based trick for supporting induction on nonvariable arguments to type families. The trick works fine without axioms for datatypes more traditional than **formula**, but we run into trouble with the current type.

```
Lemma proj1_again' : ∀ r, proof r
  → ∀ p q, r = And p q → proof p.
  destruct 1; crush.

  H0 : Inject [] P = And p q
  ============================
   proof p
```

The first goal looks reasonable. Hypothesis *H0* is clearly contradictory, as `discriminate` can show.

```
  discriminate.

  H : proof p
  H1 : And p q = And p0 q0
  ============================
   proof p0
```

It looks like we are almost done. Hypothesis *H1* gives $p = p0$ by injectivity of constructors, and then H finishes the case.

```
  injection H1; intros.
```

Unfortunately, the equality that we expected between p and $p0$ comes in a strange form.

```
  H3 : existT (fun Ts : list Type ⇒ formula Ts) []%list p =
       existT (fun Ts : list Type ⇒ formula Ts) []%list p0
  ============================
   proof p0
```

Reviewing the discussion in Chapter 3 of writing injection principles manually, we see that an existT type is the most direct way to express the output of `injection` on a dependently typed constructor. The constructor And is dependently typed, since it takes a parameter Ts upon which the types of p and q depend. Let us not dwell further here on why this goal appears; readers may like to attempt the (impossible) exercise of building a better injection lemma for And without using axioms.

How exactly does an axiom come into the picture here? Let us ask *crush* to finish the proof.

```
  crush.
Qed.

Print Assumptions proj1_again'.

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

It turns out that this familiar axiom about equality (or some other axiom) is required to deduce $p = p0$ from the hypothesis $H3$ above. The soundness of that proof step is neither provable nor disprovable in Gallina.

We can produce an even stranger-looking lemma, which gives us the theorem without axioms. As always when we want to do case analysis on a term with a tricky dependent type, the key is to refactor the theorem statement so that every term we `match` on has *variables* as its type indices; so instead of talking about proofs of And p q, we talk about proofs of an arbitrary r, but we only conclude anything interesting when r is an And.

```
Lemma proj1_again'' : ∀ r, proof r
  → match r with
      | And Ps p _ ⇒ match Ps return formula Ps → Prop with
                       | nil ⇒ fun p ⇒ proof p
                       | _ ⇒ fun _ ⇒ True
                     end p
      | _ ⇒ True
    end.
  destruct 1; auto.
Qed.

Theorem proj1_again : ∀ p q, proof (And p q) → proof p.
  intros ? ? H; exact (proj1_again'' H).
Qed.

Print Assumptions proj1_again.
```

```
Closed under the global context
```

This example illustrates again how some of the design patterns for dependently typed programming can be used fruitfully in theorem statements.

Consider one final way to avoid dependence on axioms. Often this task is equivalent to writing definitions such that they *compute*. That is, we want Coq's normal reduction to be able to run certain programs to completion. Here is a simple example where such computation can get stuck. In proving properties of such functions, we would need to apply axioms like K manually to make progress.

Imagine we are working with deeply embedded syntax of some programming language, where each term is considered to be in the scope of a number of free variables that hold normal Coq values. To enforce proper typing, we need to model a Coq typing environment somehow. One natural choice is as a list of types, where variable number i is treated as a reference to the ith element of the list.

```
Section withTypes.
  Variable types : list Set.
```

To give the semantics of terms, we need to represent value environments, which assign each variable a term of the proper type.

```
  Variable values : hlist (fun x : Set ⇒ x) types.
```

Now imagine that we are writing some procedure that operates on a distinguished variable of type **nat**. A hypothesis formalizes this assumption, using the standard library function nth_error for looking up list elements by position.

```
  Variable natIndex : nat.
  Variable natIndex_ok : nth_error types natIndex = Some nat.
```

It is not hard to use this hypothesis to write a function for extracting the **nat** value in position *natIndex* of *values*, starting with two helpful lemmas, each of which we finish with `Defined` to mark the lemma as transparent, so that its definition may be expanded during evaluation.

```
  Lemma nth_error_nil : ∀ A n x,
    nth_error (@nil A) n = Some x
    → False.
    destruct n; simpl; unfold error; congruence.
  Defined.

  Implicit Arguments nth_error_nil [A n x].

  Lemma Some_inj : ∀ A (x y : A),
```

```
    Some x = Some y
    → x = y.
    congruence.
  Defined.

  Fixpoint getNat (types' : list Set)
    (values' : hlist (fun x : Set ⇒ x) types')
    (natIndex : nat)
    : (nth_error types' natIndex = Some nat) → nat :=
    match values' with
      | HNil ⇒ fun pf ⇒ match nth_error_nil pf with end
      | HCons t ts x values'' ⇒
        match natIndex return nth_error (t :: ts) natIndex
          = Some nat → nat with
          | O ⇒ fun pf ⇒
            match Some_inj pf in _ = T return T with
              | eq_refl ⇒ x
            end
          | S natIndex' ⇒ getNat values'' natIndex'
        end
    end.
End withTypes.
```

The problem becomes apparent when we experiment with running getNat on a concrete *types* list.

```
Definition myTypes := unit :: nat :: bool :: nil.
Definition myValues : hlist (fun x : Set ⇒ x) myTypes :=
  tt ::: 3 ::: false ::: HNil.

Definition myNatIndex := 1.

Theorem myNatIndex_ok : nth_error myTypes myNatIndex = Some nat.
  reflexivity.
Defined.

Eval compute in getNat myValues myNatIndex myNatIndex_ok.
```

```
    = 3
```

We have not hit the problem yet, since we proceeded with a concrete equality proof for myNatIndex_ok. However, consider a case where we want to reason about the behavior of getNat *independently* of a specific proof.

```
Theorem getNat_is_reasonable : ∀ pf,
  getNat myValues myNatIndex pf = 3.
  intro; compute.
```

```
1 subgoal
  pf : nth_error myTypes myNatIndex = Some nat
  ============================
   match
     match
       pf in (_ = y)
       return (nat = match y with
                     | Some H ⇒ H
                     | None ⇒ nat
                     end)
     with
     | eq_refl ⇒ eq_refl
     end in (_ = T) return T
   with
   | eq_refl ⇒ 3
   end = 3
```

Since the details of the equality proof *pf* are not known, computation can proceed no further. A rewrite with axiom K would allow us to make progress, but we can rethink the definitions a bit to avoid depending on axioms.

```
Abort.
```

Here is a definition of a function that turns out to be useful. A call update *ls n x* overwrites the *n*th position of the list *ls* with the value *x*, padding the end of the list with extra *x* values as needed to ensure sufficient length.

```
Fixpoint copies A (x : A) (n : nat) : list A :=
  match n with
    | O ⇒ nil
    | S n' ⇒ x :: copies x n'
  end.

Fixpoint update A (ls : list A) (n : nat) (x : A) : list A :=
  match ls with
    | nil ⇒ copies x n ++ x :: nil
    | y :: ls' ⇒ match n with
                   | O ⇒ x :: ls'
                   | S n' ⇒ y :: update ls' n' x
                 end
  end.
```

Now let us revisit the definition of getNat.

```
Section withTypes'.
```

```
Variable types : list Set.
Variable natIndex : nat.
```

Instead of asserting properties about the list *types*, we build a new list that is *guaranteed by construction* to have those properties.

```
Definition types' := update types natIndex nat.

Variable values : hlist (fun x : Set ⇒ x) types'.
```

Now a bit of dependent pattern matching helps us rewrite getNat in a way that avoids any use of equality proofs.

```
Fixpoint skipCopies (n : nat)
  : hlist (fun x : Set ⇒ x) (copies nat n ++ nat :: nil) → nat :=
  match n with
    | O ⇒ fun vs ⇒ hhd vs
    | S n' ⇒ fun vs ⇒ skipCopies n' (htl vs)
  end.

Fixpoint getNat' (types'' : list Set) (natIndex : nat)
  : hlist (fun x : Set ⇒ x) (update types'' natIndex nat) → nat :=
  match types'' with
    | nil ⇒ skipCopies natIndex
    | t :: types0 ⇒
      match natIndex return hlist (fun x : Set ⇒ x)
        (update (t :: types0) natIndex nat) → nat with
        | O ⇒ fun vs ⇒ hhd vs
        | S natIndex' ⇒ fun vs ⇒ getNat' types0 natIndex' (htl vs)
      end
  end.
End withTypes'.
```

The surprise comes in how easy it is to use getNat'. While typing works by modification of a types list, we can choose parameters so that the modification has no effect.

```
Theorem getNat_is_reasonable
  : getNat' myTypes myNatIndex myValues = 3.
  reflexivity.
Qed.
```

The same parameters as before work without alteration, and we avoid use of axioms.