6 Subset Types and Variations

So far, we have seen many examples of what might be called classical program verification. We write programs, write their specifications, and then prove that the programs satisfy their specifications. The programs that we have written in Coq have been normal functional programs that we could just as well have written in Haskell or ML. In this chapter, we start investigating uses of dependent types to integrate programming, specification, and proving into a single phase. These techniques make it possible to reduce the cost of program verification dramatically.

6.1 Introducing Subset Types

Let us consider several ways of implementing the natural number predecessor function. We start by displaying the definition from the standard library.

```
Print pred.
```

```
pred = fun n : nat ⇒ match n with
                     | 0 ⇒ 0
                     | S u ⇒ u
                     end
     : nat → nat
```

We can use a new command, `Extraction`, to produce an OCaml version of this function.

```
Extraction pred.
```

```
(** val pred : nat -> nat **)

let pred = function
  | O -> O
```

```
  | S u -> u
```

Returning 0 as the predecessor of 0 can come across as somewhat of a hack. In some situations, we might like to be sure that we never try to take the predecessor of 0. We can enforce this by giving pred a stronger, dependent type.

```
Lemma zgtz : 0 > 0 → False.
  crush.
Qed.

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 expand the type of pred to include a proof that its argument n is greater than 0. When n is 0, we use the proof to derive a contradiction, which we can use to build a value of any type via a vacuous pattern match. When n is a successor, we have no need for the proof and just return the answer. The proof argument can be said to have a dependent type because its type *depends* on the value of the argument n.

Coq's `Eval` command can execute particular invocations of pred_strong1 just as easily as it can execute more traditional functional programs. Note that Coq has decided that argument n of pred_strong1 can be made *implicit*, since it can be deduced from the type of the second argument, so we need not write n in function calls.

```
Theorem two_gt0 : 2 > 0.
  crush.
Qed.

Eval compute in pred_strong1 two_gt0.
```

```
     = 1
     : nat
```

One aspect in particular of the definition of pred_strong1 may be surprising. We took advantage of `Definition`'s syntactic sugar for defining function arguments in the case of n, but we bound the proofs later with explicit `fun` expressions. Let us see what happens if we write this function in the way that at first seems most natural.

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

```
    | S n' ⇒ n'
  end.

Error: In environment
n : nat
pf : n > 0
The term "pf" has type "n > 0" while it is expected to have
type "0 > 0"
```

The term zgtz *pf* fails to type-check. Somehow the type checker has failed to take into account information that follows from which `match` branch that term appears in. The problem is that, by default, `match` does not let us use such implied information. To get refined typing, we must always rely on `match` annotations, either written explicitly or inferred.

In this case, we must use a `return` annotation to declare the relation between the *value* of the `match` discriminee and the *type* of the result. There is no annotation that lets us declare a relation between the discriminee and the type of a variable that is already in scope; hence, we delay the binding of *pf* so that we can use the `return` annotation to express the needed relation.

Coq's heuristics infer the `return` clause (specifically, `return` $n > 0 \rightarrow$ **nat**) in the definition of pred_strong1, leading to the following elaborated code.

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

By making explicit the functional relation between value n and the result type of the `match`, we guide Coq toward proper type checking. The clause for this example follows by simple copying of the original annotation on the definition. In general, however, the `match` annotation inference problem is undecidable. The known undecidable problem of *higher-order unification* [15] reduces to the `match` type inference problem. Over time, Coq is enhanced with more and more heuristics to get around this problem, but there must always exist `match`es whose types Coq cannot infer without annotations.

Let us now take a look at the OCaml code Coq generates for pred_strong1.

```
Extraction pred_strong1.
```

```
(** val pred_strong1 : nat -> nat **)

let pred_strong1 = function
  | O -> assert false (* absurd case *)
  | S n' -> n'
```

The proof argument has disappeared. We get exactly the OCaml code we would have written manually. This is the first demonstration of the main technically interesting feature of Coq program extraction: proofs are erased systematically.

We can reimplement the dependently typed pred based on *subset types*, defined in the standard library with the type family **sig**.

```
Print sig.
```

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

The family **sig** is a Curry-Howard twin of **ex**, except that **sig** is in Type, whereas **ex** is in Prop. That means that **sig** values can survive extraction, but **ex** proofs will always be erased. The actual details of extraction of **sig**s are more subtle.

We rewrite pred_strong1, using some syntactic sugar for subset types.

```
Locate "{ _: _| _}".
```

```
  Notation
  "{ x : A | P }" := sig (fun x : A ⇒ P)
```

```
Definition pred_strong2 (s : {n : nat | n > 0}) : nat :=
  match s with
    | exist O pf ⇒ match zgtz pf with end
    | exist (S n') _ ⇒ n'
  end.
```

To build a value of a subset type, we use the exist constructor; the details of how to do that follow from the output of the earlier Print **sig** command, where we elided the extra information that parameter A is implicit. We need an extra _ here and not in the definition of pred_strong2 because *parameters* of inductive types (like the predicate P for **sig**) are not mentioned in pattern matching but *are* mentioned in construction of terms (if they are not marked as implicit arguments).

```
Eval compute in pred_strong2 (exist _ 2 two_gt0).
```

```
     = 1
     : nat
```

```
Extraction pred_strong2.

(** val pred_strong2 : nat -> nat **)

let pred_strong2 = function
  | O -> assert false (* absurd case *)
  | S n' -> n'
```

We arrive at the same OCaml code as was extracted from pred_strong1. The reason is that a value of **sig** is a pair of two pieces, a value and a proof about it. Extraction erases the proof, which reduces the constructor exist of **sig** to taking just a single argument. An optimization eliminates uses of datatypes with single constructors taking single arguments, and we arrive back where we started.

We can continue the process of refining pred's type. Let us change its result type to capture that the output is really the predecessor of the input.

```
Definition pred_strong3 (s : {n : nat | n > 0})
  : {m : nat | proj1_sig s = S m} :=
  match s return {m : nat | proj1_sig s = S m} with
    | exist 0 pf ⇒ match zgtz pf with end
    | exist (S n') pf ⇒ exist _ n' eq_refl
  end.

Eval compute in pred_strong3 (exist _ 2 two_gt0).

     = exist (fun m : nat ⇒ 2 = S m) 1 eq_refl
     : {m : nat | proj1_sig (exist (lt 0) 2 two_gt0) = S m}
```

A value in a subset type can be thought of as a *dependent pair* (or *sigma type*) of a base value and a proof about it. The function proj1_sig extracts the first component of the pair, but we need to include an explicit `return` clause, since Coq's heuristics do not propagate the result type that we wrote earlier.

The new pred_strong leads to the same OCaml code we have seen several times so far.

```
Extraction pred_strong3.

(** val pred_strong3 : nat -> nat **)

let pred_strong3 = function
  | O -> assert false (* absurd case *)
  | S n' -> n'
```

We have managed to reach a type that is, in a formal sense, the most expressive possible for pred. Any other implementation of the same type must have the same input-output behavior. However, there is still room for improvement in making this kind of code easier to write. Here is a version that takes advantage of tactic-based theorem proving. We switch back to passing a separate proof argument instead of using a subset type for the function's input because this leads to cleaner code. (Recall that False_rec is the Set-level induction principle for **False**, which can be used to produce a value in any Set given a proof of **False**.)

```
Definition pred_strong4 : ∀ n : nat, n > 0 → {m : nat | n = S m}.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ False_rec _ _
      | S n' ⇒ fun _ ⇒ exist _ n' _
    end).
```

We build pred_strong4 using tactic-based proving, beginning with a Definition command that ends in a period before a definition is given. Such a command enters the interactive proving mode, with the type given for the new identifier as the proof goal. It may seem strange to change perspective so implicitly between programming and proving, but recall that programs and proofs are two sides of the same coin in Coq, thanks to the Curry-Howard correspondence.

We do most of the work with the refine tactic, to which we pass a partial "proof" of the type we are trying to prove. There may be some pieces left to fill in, indicated by underscores. Any underscore that Coq cannot reconstruct with type inference is added as a proof subgoal. In this case, we have two subgoals.

```
2 subgoals

  n : nat
  _ : 0 > 0
  ============================
   False

subgoal 2 is

 S n' = S n'
```

The first subgoal comes from the second underscore passed to False_rec, and the second subgoal comes from the second underscore

passed to exist. In the first case, we see that though we bound the proof variable with an underscore, it is still available in the proof context. It is hard to refer to underscore-named variables in manual proofs, but automation makes short work of them. Both subgoals are easy to discharge that way, so let us back up and ask to prove all subgoals automatically.

```
  Undo.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ False_rec _ _
      | S n' ⇒ fun _ ⇒ exist _ n' _
    end); crush.
Defined.
```

We end the "proof" with `Defined` instead of `Qed`, so that the definition we constructed remains visible. This contrasts with the case of ending a proof with `Qed`, where the details of the proof are hidden afterward. (More formally, `Defined` marks an identifier as *transparent*, allowing it to be unfolded, whereas `Qed` marks an identifier as *opaque*, preventing unfolding.) Let us see what the proof script constructed.

```
Print pred_strong4.
```

```
pred_strong4 =
fun n : nat ⇒
match n as n0 return (n0 > 0 → {m : nat | n0 = S m}) with
| 0 ⇒
    fun _ : 0 > 0 ⇒
    False_rec {m : nat | 0 = S m}
      (Bool.diff_false_true
         (Bool.absurd_eq_true false
            (Bool.diff_false_true
               (Bool.absurd_eq_true false
                  (pred_strong4_subproof n _)))))
| S n' ⇒
    fun _ : S n' > 0 ⇒
    exist (fun m : nat ⇒ S n' = S m) n' eq_refl
end
     : ∀ n : nat, n > 0 → {m : nat | n = S m}
```

We see the code we entered, with some proofs filled in. The first proof obligation, the second argument to False_rec, is filled in with a proof term that we can be glad we did not enter by hand. The second proof obligation is a simple reflexivity proof.

```
Eval compute in pred_strong4 two_gt0.

    = exist (fun m : nat ⇒ 2 = S m) 1 eq_refl
    : {m : nat | 2 = S m}
```

A tactic modifier called `abstract` can be helpful for producing shorter terms, by automatically abstracting subgoals into named lemmas.

```
Definition pred_strong4' : ∀ n : nat, n > 0 → {m : nat | n = S m}.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ False_rec _ _
      | S n' ⇒ fun _ ⇒ exist _ n' _
    end); abstract crush.
Defined.

Print pred_strong4'.

pred_strong4' =
fun n : nat ⇒
match n as n0 return (n0 > 0 → {m : nat | n0 = S m}) with
| 0 ⇒
    fun _H : 0 > 0 ⇒
    False_rec {m : nat | 0 = S m} (pred_strong4'_subproof n _H)
| S n' ⇒
    fun _H : S n' > 0 ⇒
    exist (fun m : nat ⇒ S n' = S m) n'
      (pred_strong4'_subproof0 n _H)
end
     : ∀ n : nat, n > 0 → {m : nat | n = S m}
```

We are almost done with the ideal implementation of dependent predecessor. We can use Coq's syntax extension facility to arrive at code with almost no complexity beyond a Haskell or ML program with a complete specification in a comment. In this book, I do not dwell on the details of syntax extensions; the Coq manual gives a straightforward introduction to them.

```
Notation "!" := (False_rec _ _).
Notation "[ e ]" := (exist _ e _).

Definition pred_strong5 : ∀ n : nat, n > 0 → {m : nat | n = S m}.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ !
      | S n' ⇒ fun _ ⇒ [n']
    end); crush.
```

```
Defined.
```

By default, notations are also used in pretty-printing terms, including results of evaluation.

```
Eval compute in pred_strong5 two_gt0.
```

```
    = [1]
    : {m : nat | 2 = S m}
```

One other alternative is worth demonstrating. Recent Coq versions include a facility called `Program` that streamlines this style of definition. Here is a complete implementation using `Program`:

```
Obligation Tactic := crush.

Program Definition pred_strong6 (n : nat) (_ : n > 0)
  : {m : nat | n = S m} :=
  match n with
    | O ⇒ _
    | S n' ⇒ n'
  end.
```

Printing the resulting definition of `pred_strong6` yields a term very similar to what we built with `refine`. `Program` can save time in writing programs that use subset types. Nonetheless, `refine` is often just as effective, and `refine` gives more control over the form of the final term, which can be useful to prove additional theorems about the definition. `Program` will sometimes insert type casts that can complicate theorem proving.

```
Eval compute in pred_strong6 two_gt0.
```

```
    = [1]
    : {m : nat | 2 = S m}
```

In this case, we see that the new definition yields the same computational behavior as before.

6.2 Decidable Proposition Types

Another type in the standard library captures the idea of program values that indicate which of two propositions is true.

```
Print sumbool.
```

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
    left : A → {A} + {B} | right : B → {A} + {B}
```

Here, the constructors of **sumbool** have types written in terms of a registered notation for **sumbool**, such that the result type of each constructor desugars to **sumbool** A B. We can define some notations to make working with **sumbool** more convenient.

```
Notation "'Yes'" := (left _ _).
Notation "'No'" := (right _ _).
Notation "'Reduce' x" := (if x then Yes else No) (at level 50).
```

The `Reduce` notation is notable because it demonstrates how `if` is overloaded in Coq. The `if` form actually works when the test expression has any two-constructor inductive type. Moreover, in the `then` and `else` branches, the appropriate constructor arguments are bound. This is important when working with **sumbool**s, when we want to have the proof stored in the test expression available when proving the proof obligations generated in the appropriate branch.

Now we can write `eq_nat_dec`, which compares two natural numbers, returning either a proof of their equality or a proof of their inequality.

```
Definition eq_nat_dec : ∀ n m : nat, {n = m} + {n ≠ m}.
  refine (fix f (n m : nat) : {n = m} + {n ≠ m} :=
    match n, m with
      | O, O ⇒ Yes
      | S n', S m' ⇒ Reduce (f n' m')
      | _, _ ⇒ No
    end); congruence.
Defined.
```

```
Eval compute in eq_nat_dec 2 2.
```

$$= Yes$$
$$: \{2 = 2\} + \{2 \neq 2\}$$

```
Eval compute in eq_nat_dec 2 3.
```

$$= No$$
$$: \{2 = 3\} + \{2 \neq 3\}$$

Note that the `Yes` and `No` notations are hiding proofs establishing the correctness of the outputs.

The definition extracts to reasonable OCaml code.

```
Extraction eq_nat_dec.
```

```
(** val eq_nat_dec : nat -> nat -> sumbool **)

let rec eq_nat_dec n m =
  match n with
```

```
  | O -> (match m with
          | O -> Left
          | S n0 -> Right)
  | S n' -> (match m with
              | O -> Right
              | S m' -> eq_nat_dec n' m')
```

Proving this kind of decidable equality result is so common that Coq comes with a tactic for automating it.

```
Definition eq_nat_dec' (n m : nat) : {n = m} + {n ≠ m}.
  decide equality.
Defined.
```

Readers can verify that the `decide equality` version extracts to the same OCaml code as the more manual version does. That OCaml code had one undesirable property, which is that it uses `Left` and `Right` constructors instead of the Boolean values built into OCaml. We can fix this by using Coq's facility for mapping Coq inductive types to OCaml variant types.

```
Extract Inductive sumbool ⇒ "bool" ["true" "false"].
Extraction eq_nat_dec'.
```

```
(** val eq_nat_dec' : nat -> nat -> bool **)

let rec eq_nat_dec' n m0 =
  match n with
    | O -> (match m0 with
              | O -> true
              | S n0 -> false)
    | S n0 -> (match m0 with
                | O -> false
                | S n1 -> eq_nat_dec' n0 n1)
```

We can build smart versions of the usual Boolean operators and put them to good use in certified programming. For instance, here is a **sumbool** version of Boolean "or":

```
Notation "x || y" := (if x then Yes else Reduce y).
```

Let us use it for building a function that decides list membership. We need to assume the existence of an equality decision procedure for the type of list elements.

```
Section In_dec.
```

```
Variable A : Set.
Variable A_eq_dec : ∀ x y : A, {x = y} + {x ≠ y}.
```

The final function is easy to write using the techniques we have developed so far.

```
Definition In_dec : ∀ (x : A) (ls : list A), {In x ls} + {¬ In x ls}.
  refine (fix f (x : A) (ls : list A) : {In x ls} + {¬ In x ls} :=
    match ls with
      | nil ⇒ No
      | x' :: ls' ⇒ A_eq_dec x x' || f x ls'
    end); crush.
Defined.
End In_dec.
```

```
Eval compute in In_dec eq_nat_dec 2 (1 :: 2 :: nil).
```

```
= Yes
: {In 2 (1 :: 2 :: nil)} + { ¬ In 2 (1 :: 2 :: nil)}
```

```
Eval compute in In_dec eq_nat_dec 3 (1 :: 2 :: nil).
```

```
= No
: {In 3 (1 :: 2 :: nil)} + { ¬ In 3 (1 :: 2 :: nil)}
```

The In_dec function has a reasonable extraction to OCaml.

```
Extraction In_dec.
```

```
(** val in_dec : ('a1 -> 'a1 -> bool) -> 'a1
    -> 'a1 list -> bool **)

let rec in_dec a_eq_dec x = function
  | Nil -> false
  | Cons (x', ls') ->
      (match a_eq_dec x x' with
         | true -> true
         | false -> in_dec a_eq_dec x ls')
```

This is more or less the code for the corresponding function from the OCaml standard library.

6.3 Partial Subset Types

The final implementation of dependent predecessor used a very specific argument type to ensure that execution could always complete normally. Sometimes we want to allow execution to fail, and we want a more

principled way of signaling failure than returning a default value, as pred does for 0. One approach is to define the type family **maybe**, which is a version of **sig** that allows obligation-free failure.

```
Inductive maybe (A : Set) (P : A → Prop) : Set :=
| Unknown : maybe P
| Found : ∀ x : A, P x → maybe P.
```

We can define some new notations, analogous to those we defined for subset types.

```
Notation "{{ x | P }}" := (maybe (fun x ⇒ P)).
Notation "??" := (Unknown _).
Notation "[| x |]" := (Found _ x _).
```

Now the next version of pred is trivial to write.

```
Definition pred_strong7 : ∀ n : nat, {{m | n = S m}}.
  refine (fun n ⇒
    match n return {{m | n = S m}} with
      | O ⇒ ??
      | S n' ⇒ [|n'|]
    end); trivial.
Defined.
```

```
Eval compute in pred_strong7 2.
```

```
= [|1|]
: {{m | 2 = S m}}
```

```
Eval compute in pred_strong7 0.
```

```
= ??
: {{m | 0 = S m}}
```

Because we used **maybe**, one valid implementation of the type we gave pred_strong7 would return ?? in every case. We can strengthen the type to rule out such vacuous implementations; the type family **sumor** from the standard library provides the easiest starting point. For type A and proposition B, $A + \{B\}$ desugars to **sumor** A B, whose values are either values of A or proofs of B.

```
Print sumor.
```

```
Inductive sumor (A : Type) (B : Prop) : Type :=
    inleft : A → A + {B} | inright : B → A + {B}
```

We add notations for easy use of the **sumor** constructors. The second notation is specialized to **sumor**s whose A parameters are instantiated with regular subset types, since this is how we will use **sumor**.

```
Notation "!!" := (inright _ _).
Notation "[|| x ||]" := (inleft _ [x]).
```

Now we are ready to give the final version of possibly failing predecessor. The **sumor**-based type that we use is maximally expressive; any implementation of the type has the same input-output behavior.

```
Definition pred_strong8 : ∀ n : nat, {m : nat | n = S m} + {n = 0}.
  refine (fun n ⇒
    match n with
      | O ⇒ !!
      | S n' ⇒ [||n'||]
    end); trivial.
Defined.
```

```
Eval compute in pred_strong8 2.
```

```
= [||1||]
: {m : nat | 2 = S m} + {2 = 0}
```

```
Eval compute in pred_strong8 0.
```

```
= !!
: {m : nat | 0 = S m} + {0 = 0}
```

As with the other maximally expressive pred function, we arrive at quite simple output values, thanks to notations.

6.4 Monadic Notations

We can treat **maybe** like a monad [44], in the same way that the Haskell `Maybe` type is interpreted as a failure monad. **maybe** has the wrong type to be a literal monad, but a bind-like notation will still be helpful. Note that the notation definition uses an ASCII `<-`, whereas later code uses (in this rendering) a nicer left arrow ←.

```
Notation "x <- e1 ; e2" := (match e1 with
                              | Unknown ⇒ ??
                              | Found x _ ⇒ e2
                            end)
(right associativity, at level 60).
```

The meaning of $x \leftarrow e1$; $e2$ is the following: First run $e1$. If it fails to find an answer, then announce failure for the derived computation, too. If $e1$ *does* find an answer, pass that answer on to $e2$ to find the final result. The variable x can be considered bound in $e2$.

This notation is very helpful for composing richly typed procedures. For instance, here is a very simple implementation of a function to take the predecessors of two naturals at once:

```
Definition doublePred : ∀ n1 n2 : nat,
  {{p | n1 = S (fst p) ∧ n2 = S (snd p)}}.
  refine (fun n1 n2 ⇒
    m1 ← pred_strong7 n1;
    m2 ← pred_strong7 n2;
    [|(m1, m2)|]); tauto.
Defined.
```

We can build a **sumor** version of the bind notation and use it to write a similarly straightforward version of this function. Again, the notation definition exposes the ASCII syntax with an operator `<--`, and the later code uses a nicer long left arrow ⟵.

```
Notation "x <- e1 ; e2" := (match e1 with
                              | inright _ ⇒ !!
                              | inleft (exist x _) ⇒ e2
                            end)
(right associativity, at level 60).

Definition doublePred' : ∀ n1 n2 : nat,
  {p : nat × nat | n1 = S (fst p) ∧ n2 = S (snd p)}
  + {n1 = 0 ∨ n2 = 0}.
  refine (fun n1 n2 ⇒
    m1 ⟵ pred_strong8 n1;
    m2 ⟵ pred_strong8 n2;
    [||(m1, m2)||]); tauto.
Defined.
```

This example demonstrates how judicious selection of notations can hide complexities in the rich types of programs.

6.5 A Type-Checking Example

We can apply these specification types to build a certified type checker for a simple expression language.

```
Inductive exp : Set :=
| Nat : nat → exp
| Plus : exp → exp → exp
| Bool : bool → exp
```

```
| And : exp → exp → exp.
```

We define a simple language of types and its typing rules in the style introduced in Chapter 4.

```
Inductive type : Set := TNat | TBool.

Inductive hasType : exp → type → Prop :=
| HtNat : ∀ n,
  hasType (Nat n) TNat
| HtPlus : ∀ e1 e2,
  hasType e1 TNat
  → hasType e2 TNat
  → hasType (Plus e1 e2) TNat
| HtBool : ∀ b,
  hasType (Bool b) TBool
| HtAnd : ∀ e1 e2,
  hasType e1 TBool
  → hasType e2 TBool
  → hasType (And e1 e2) TBool.
```

It will be helpful to have a function for comparing two types. We build one using `decide equality`.

```
Definition eq_type_dec : ∀ t1 t2 : type, {t1 = t2} + {t1 ≠ t2}.
  decide equality.
Defined.
```

Another notation complements the monadic notation for **maybe**, defined earlier. Sometimes we want to include assertions in a procedure. That is, we want to run a decision procedure and fail if it fails; otherwise, we want to continue, with the proof that it produced made available to us. This infix notation captures that idea for a procedure that returns an arbitrary two-constructor type.

```
Notation "e1 ;; e2" := (if e1 then e2 else ??)
  (right associativity, at level 60).
```

With that notation defined, we can implement a typeCheck function, whose code is only more complex than what we would write in ML because it needs to include some extra type annotations. Every $[|e|]$ expression adds a **hasType** proof obligation, and *crush* makes short work of them when we add **hasType**'s constructors as hints.

```
Definition typeCheck : ∀ e : exp, {{t | hasType e t}}.
  Hint Constructors hasType.

  refine (fix F (e : exp) : {{t | hasType e t}} :=
    match e return {{t | hasType e t}} with
```

```
      | Nat _ ⇒ [|TNat|]
      | Plus e1 e2 ⇒
        t1 ← F e1;
        t2 ← F e2;
        eq_type_dec t1 TNat;;
        eq_type_dec t2 TNat;;
        [|TNat|]
      | Bool _ ⇒ [|TBool|]
      | And e1 e2 ⇒
        t1 ← F e1;
        t2 ← F e2;
        eq_type_dec t1 TBool;;
        eq_type_dec t2 TBool;;
        [|TBool|]
    end); crush.
Defined.
```

Despite manipulating proofs, this type checker is easy to run.

```
Eval simpl in typeCheck (Nat 0).
```

```
     = [|TNat|]
     : {{t | hasType (Nat 0) t}}
```

```
Eval simpl in typeCheck (Plus (Nat 1) (Nat 2)).
```

```
     = [|TNat|]
     : {{t | hasType (Plus (Nat 1) (Nat 2)) t}}
```

```
Eval simpl in typeCheck (Plus (Nat 1) (Bool false)).
```

```
     = ??
     : {{t | hasType (Plus (Nat 1) (Bool false)) t}}
```

The type checker also extracts to some reasonable OCaml code.

```
Extraction typeCheck.
```

```
(** val typeCheck : exp -> type0 maybe **)

let rec typeCheck = function
  | Nat n -> Found TNat
  | Plus (e1, e2) ->
      (match typeCheck e1 with
         | Unknown -> Unknown
         | Found t1 ->
             (match typeCheck e2 with
```

```
                | Unknown -> Unknown
                | Found t2 ->
                    (match eq_type_dec t1 TNat with
                       | true ->
                           (match eq_type_dec t2 TNat with
                              | true -> Found TNat
                              | false -> Unknown)
                       | false -> Unknown)))
  | Bool b -> Found TBool
  | And (e1, e2) ->
      (match typeCheck e1 with
         | Unknown -> Unknown
         | Found t1 ->
             (match typeCheck e2 with
                | Unknown -> Unknown
                | Found t2 ->
                    (match eq_type_dec t1 TBool with
                       | true ->
                           (match eq_type_dec t2 TBool with
                              | true -> Found TBool
                              | false -> Unknown)
                       | false -> Unknown)))
```

We can adapt this implementation to use **sumor**, so that we know the type checker only fails on ill-typed inputs. First, we define an analogue to the assertion notation.

```
Notation "e1 ;;; e2" := (if e1 then e2 else !!)
  (right associativity, at level 60).
```

Next, we prove a helpful lemma, which states that a given expression can have at most one type.

```
Lemma hasType_det : ∀ e t1,
  hasType e t1
  → ∀ t2, hasType e t2
    → t1 = t2.
  induction 1; inversion 1; crush.
Qed.
```

Now we can define the type checker. Its type expresses that it only fails on untypable expressions.

```
Definition typeCheck' : ∀ e : exp,
  {t : type | hasType e t} + {∀ t, ¬ hasType e t}.
```

```
Hint Constructors hasType.
```

We register all the typing rules as hints.

```
Hint Resolve hasType_det.
```

The lemma hasType_det will also be useful for proving proof obligations with contradictory contexts. Since its statement includes ∀-bound variables that do not appear in its conclusion, only `eauto` will apply this hint.

Finally, the implementation of typeCheck can be transcribed literally, simply switching notations as needed.

```
refine (fix F (e : exp) : {t : type | hasType e t}
  + {∀ t, ¬ hasType e t} :=
  match e return {t : type | hasType e t}
    + {∀ t, ¬ hasType e t} with
    | Nat _ ⇒ [||TNat||]
    | Plus e1 e2 ⇒
      t1 ⟵ F e1;
      t2 ⟵ F e2;
      eq_type_dec t1 TNat;;;
      eq_type_dec t2 TNat;;;
      [||TNat||]
    | Bool _ ⇒ [||TBool||]
    | And e1 e2 ⇒
      t1 ⟵ F e1;
      t2 ⟵ F e2;
      eq_type_dec t1 TBool;;;
      eq_type_dec t2 TBool;;;
      [||TBool||]
  end); clear F; crush' tt hasType; eauto.
```

We clear F, the local name for the recursive function, to avoid strange proofs that refer to recursive calls that we never make. Such a step is usually warranted when defining a recursive function with `refine`. The *crush* variant *crush'* performs automatic inversion on instances of the predicates specified in its second argument. Once we include `eauto` to apply hasType_det, we have discharged all the subgoals.

```
Defined.
```

The short implementation here hides just how time-saving automation is. Every use of one of the notations adds a proof obligation, giving us twelve in total. Most of these obligations require multiple inversions and either uses of hasType_det or applications of **hasType** rules.

The new function remains easy to test.

```
Eval simpl in typeCheck' (Nat 0).
```

```
= [||TNat||]
: {t : type | hasType (Nat 0) t} +
  {(∀ t : type, ¬ hasType (Nat 0) t)}
```

```
Eval simpl in typeCheck' (Plus (Nat 1) (Nat 2)).
```

```
= [||TNat||]
: {t : type | hasType (Plus (Nat 1) (Nat 2)) t} +
  {(∀ t : type, ¬ hasType (Plus (Nat 1) (Nat 2)) t)}
```

```
Eval simpl in typeCheck' (Plus (Nat 1) (Bool false)).
```

```
= !!
: {t : type | hasType (Plus (Nat 1) (Bool false)) t} +
  {(∀ t : type, ¬ hasType (Plus (Nat 1) (Bool false)) t)}
```

The results of simplifying calls to typeCheck' look deceptively similar to the results for typeCheck, but now the types of the results provide more information.