8 More Dependent Types

Subset types and their relatives help us integrate verification with programming. Though they reorganize the programmer's work flow, they tend not to have deep effects on proofs. We write largely the same proofs as we would for classical verification, with some of the structure moved into the programs themselves. It turns out that when we use dependent types to their full potential, we warp the development and proving process even more, picking up "free theorems" to the extent that often a certified program is hardly more complex than its uncertified counterpart in Haskell or ML.

In particular, we have only scratched the surface of Coq's inductive definition mechanism. The inductive types we have seen so far have their counterparts in the other proof assistants that we surveyed in Chapter 1. This chapter explores the world of dependent inductive datatypes outside `Prop`, a possibility that sets Coq apart from all of the competition not based on type theory.

8.1 Length-Indexed Lists

Many introductions to dependent types start out by showing how to use them to eliminate array bounds checks. When the type of an array reveals how many elements it has, the compiler can detect out-of-bounds dereferences statically. Since we are working in a pure functional language, the next best thing is length-indexed lists, which the following code defines.

```
Section ilist.
  Variable A : Set.

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

We see that, within its section, **ilist** is given type **nat** → `Set`. Previously, every inductive type had either plain `Set` as its type or was a predicate with some type ending in `Prop`. The full generality of inductive definitions lets us integrate the expressivity of predicates directly into normal programming.

The **nat** argument to **ilist** tells us the length of the list. The types of **ilist**'s constructors tell us that a Nil list has length O and that a Cons list has length one greater than the length of its tail. We may apply **ilist** to any natural number, even natural numbers that are only known at run-time. It is this breaking of the *phase distinction* that characterizes **ilist** as dependently typed.

In expositions of list types, we usually see the length function defined first, but here that would not be a very productive function to code. Instead, let us implement list concatenation.

```
Fixpoint app n1 (ls1 : ilist n1) n2 (ls2 : ilist n2)
  : ilist (n1 + n2) :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons _ x ls1' ⇒ Cons x (app ls1' ls2)
  end.
```

Past Coq versions signaled an error for this definition. The code is still invalid within Coq's core language, but current Coq versions automatically add annotations to the original program, producing a valid core program. These are the annotations on `match` discriminees introduced in Chapter 6. We can rewrite `app` to give the annotations explicitly.

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2)
  : ilist (n1 + n2) :=
  match ls1 in (ilist n1) return (ilist (n1 + n2)) with
    | Nil ⇒ ls2
    | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
  end.
```

Using `return` alone allowed us to express a dependency of the `match` result type on the *value* of the discriminee. What `in` adds is a way of expressing a dependency on the *type* of the discriminee. Specifically, the *n1* in the `in` clause is a *binding occurrence* whose scope is the `return` clause.

We may use `in` clauses only to bind names for the arguments of an inductive type family. That is, each `in` clause must be an inductive type family name applied to a sequence of underscores and variable names of the proper length. The positions for *parameters* to the type family

must all be underscores. Parameters are those arguments declared with section variables or with entries to the left of the first colon in an inductive definition. They cannot vary depending on which constructor was used to build the discriminee, so Coq prohibits pointless matches on them. It is those arguments defined in the type to the right of the colon that we may name with `in` clauses.

Our `app` function could be typed in *stratified* type systems, which avoid true dependency. That is, we could consider the length indices to lists to live in a separate, compile-time-only universe from the lists themselves. Compile-time data may be *erased* such that we can still execute a program. As an example where erasure would not work, consider an injection function from regular lists to length-indexed lists. Here the run-time computation actually depends on details of the compile-time argument, if we decide that the list to inject can be considered compile-time. More commonly, we think of lists as run-time data. Neither case will work with naïve erasure. (It is not too important to grasp the details of this run-time/compile-time distinction, since Coq's expressive power comes from avoiding such restrictions.)

```
Fixpoint inject (ls : list A) : ilist (length ls) :=
  match ls with
    | nil ⇒ Nil
    | h :: t ⇒ Cons h (inject t)
  end.
```

We can define an inverse conversion and prove that it really is an inverse.

```
Fixpoint unject n (ls : ilist n) : list A :=
  match ls with
    | Nil ⇒ nil
    | Cons _ h t ⇒ h :: unject t
  end.

Theorem inject_inverse : ∀ ls, unject (inject ls) = ls.
  induction ls; crush.
Qed.
```

Now let us attempt a function that is surprisingly tricky to write. In ML, the list head function raises an exception when passed an empty list. With length-indexed lists, we can rule out such invalid calls statically. Here is a first attempt. We write ??? as a placeholder for a term that we do not know how to write, not for any real Coq notation.

```
Definition hd n (ls : ilist (S n)) : A :=
```

```
match ls with
  | Nil ⇒ ???
  | Cons _ h _ ⇒ h
end.
```

It is not clear what to write for the Nil case, so we are stuck before we even turn the function over to the type checker. We could try omitting the Nil case.

```
Definition hd n (ls : ilist (S n)) : A :=
  match ls with
    | Cons _ h _ ⇒ h
  end.
```

```
Error: Non exhaustive pattern-matching: no clause found for
pattern Nil
```

Unlike in ML, we cannot use inexhaustive pattern matching, because there is no conception of a `Match` exception to be thrown. In fact, recent versions of Coq *do* allow this, by implicit translation to a `match` that considers all constructors; the error message was generated by an older Coq version. It is educational to discover the encoding that the most recent Coq versions use. We might try using an `in` clause.

```
Definition hd n (ls : ilist (S n)) : A :=
  match ls in (ilist (S n)) with
    | Cons _ h _ ⇒ h
  end.
```

```
Error: The reference n was not found in the current
environment
```

In this and other cases, we want `in` clauses with type family arguments that are not variables. Unfortunately, Coq only supports variables in those positions. A completely general mechanism could only be supported with a solution to the problem of higher-order unification [15], which is undecidable. There *are* useful heuristics for handling nonvariable indices that are gradually making their way into Coq, but we will use only the primitive `match` annotations in this and the next few chapters on effective pattern matching on dependent types.

The final working attempt at hd uses an auxiliary function and a surprising `return` annotation.

```
Definition hd' n (ls : ilist n) :=
  match ls in (ilist n)
```

```
    return (match n with O ⇒ unit | S _ ⇒ A end) with
    | Nil ⇒ tt
    | Cons _ h _ ⇒ h
  end.

Check hd'.

hd'
     : ∀ n : nat, ilist n → match n with
                            | 0 ⇒ unit
                            | S _ ⇒ A
                            end

Definition hd n (ls : ilist (S n)) : A := hd' ls.

End ilist.
```

We annotate the main `match` with a type that is itself a `match`. We write that the function `hd'` returns **unit** when the list is empty and returns the carried type A in all other cases. In the definition of `hd`, we just call `hd'`. Because the index of ls is known to be nonzero, the type checker reduces the `match` in the type of `hd'` to A.

8.2 The One Rule of Dependent Pattern Matching in Coq

The rest of this chapter demonstrates a few other elegant applications of dependent types in Coq. Readers encountering such ideas for the first time often feel overwhelmed, concluding that there is some magic at work whereby Coq sometimes solves the halting problem for the programmer and sometimes does not, applying automated program understanding in a way far beyond what is found in conventional languages. The point of this section is to preempt that sort of thinking. Dependent type checking in Coq follows just a few algorithmic rules. Chapters 10 and 12 introduce many of those rules more formally, and the main additional rule is centered on dependent pattern matching of the kind discussed in Section 8.1.

A dependent pattern match is a `match` expression where the type of the overall `match` is a function of the value and/or the type of the *discriminee*, the value being matched on. In other words, the `match` type *depends* on the discriminee.

When exactly will Coq accept a dependent pattern match as well-typed? Some other dependently typed languages employ elaborate decision procedures to determine when programs satisfy their very

expressive types. The situation in Coq is just the opposite. Only very straightforward symbolic rules are applied. Such a design choice has its drawbacks, as it forces programmers to do more work to convince the type checker of program validity. However, the great advantage of a simple type-checking algorithm is that its action on *invalid* programs is easier to understand.

We come now to the one rule of dependent pattern matching in Coq. A general dependent pattern match assumes this form (with unnecessary parentheses included to make the syntax easier to parse):

```
match E as y in (T x1 ... xn) return U with
  | C z1 ... zm ⇒ B
  | ...
end
```

The discriminee is a term E, a value in some inductive type family T, which takes n arguments. An `as` clause binds the name y to refer to the discriminee E. An `in` clause binds an explicit name xi for the ith argument passed to T in the type of E.

We bind these new variables y and xi so that they may be referred to in U, a type given in the `return` clause. The overall type of the `match` will be U, with E substituted for y, and with each xi substituted by the actual argument appearing in that position within E's type.

In general, each case of a `match` may have a pattern built up in several layers from the constructors of various inductive type families. To keep this exposition simple, we focus on patterns that are just single applications of inductive type constructors to lists of variables. Coq actually compiles the more general kind of pattern matching into this more restricted kind automatically, so understanding the typing of `match` requires understanding the typing of `match`es lowered to match one constructor at a time.

The last piece of the typing rule tells how to type-check a `match` case. A generic constructor application C *z1 ... zm* has some type T *x1' ... xn'*, an application of the type family used in E's type, probably with occurrences of the *zi* variables. From here, a simple recipe determines what type is required for the case body B. The type of B should be U with the following two substitutions applied: replace y (the `as` clause variable) with C *z1 ... zm*, and replace each xi (the `in` clause variables) with *xi'*. In other words, we specialize the result type based on what we learn based on which pattern has matched the discriminee.

This is an exhaustive description of the ways to specify how to take advantage of which pattern has matched. No other mechanisms come into play. For instance, there is no way to specify that the types of

certain free variables should be refined based on which pattern has matched. Later chapters present design patterns for achieving similar effects, where each technique leads to an encoding only in terms of `in`, `as`, and `return` clauses.

A few details have been omitted here. Chapter 3 showed that inductive type families may have both parameters and regular arguments. Within an `in` clause, a parameter position must have the wildcard `_` written instead of a variable. (In general, Coq uses the wildcard `_` either to indicate pattern variables that will not be mentioned again or to indicate positions where we would like type inference to infer the appropriate terms.) Recent Coq versions are adding more and more heuristics to infer dependent `match` annotations in certain conditions. The general annotation inference problem is undecidable, so there will always be serious limitations on how much work these heuristics can do. When in doubt about why a particular dependent `match` is failing to type-check, add an explicit `return` annotation. At that point, the mechanical rule sketched in this section will provide a complete account of "what the type checker is thinking." Be sure to avoid the common pitfall of writing a `return` annotation that does not mention any variables bound by `in` or `as`; such a `match` will never refine typing requirements based on which pattern has matched. (One simple exception to this rule, when the discriminee is a variable, is that the same variable may be treated as if it were repeated as an `as` clause.)

8.3 A Tagless Interpreter

A favorite example for motivating the power of functional programming is implementation of a simple expression language interpreter. In ML and Haskell, such interpreters are often implemented using an algebraic datatype of values, where at many points it is checked that a value was built with the right constructor of the value type. With dependent types, we can implement a *tagless* interpreter that both removes this source of run-time inefficiency and gives more confidence that the implementation is correct.

```
Inductive type : Set :=
| Nat : type
| Bool : type
| Prod : type → type → type.

Inductive exp : type → Set :=
| NConst : nat → exp Nat
| Plus : exp Nat → exp Nat → exp Nat
```

```
| Eq : exp Nat → exp Nat → exp Bool

| BConst : bool → exp Bool
| And : exp Bool → exp Bool → exp Bool
| If : ∀ t, exp Bool → exp t → exp t → exp t

| Pair : ∀ t1 t2, exp t1 → exp t2 → exp (Prod t1 t2)
| Fst : ∀ t1 t2, exp (Prod t1 t2) → exp t1
| Snd : ∀ t1 t2, exp (Prod t1 t2) → exp t2.
```

We have a standard algebraic datatype `type`, defining a type language of naturals, Booleans, and product (pair) types. Then we have the indexed inductive type **exp**, where the argument to **exp** reveals the encoded type of an expression. In effect, we are defining the typing rules for expressions simultaneously with the syntax.

We can give types and expressions semantics in a new style, based critically on the chance for *type-level computation*.

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Nat ⇒ nat
    | Bool ⇒ bool
    | Prod t1 t2 ⇒ typeDenote t1 × typeDenote t2
  end%type.
```

The typeDenote function compiles types of the object language into native Coq types. It is deceptively easy to implement. The only new thing is the *%type* annotation, which tells Coq to parse the `match` expression using the notations associated with types. Without this annotation, the × would be interpreted as multiplication on naturals rather than as the product type constructor. The token *type* is one example of an identifier bound to a *notation scope delimiter* (see the Coq manual).

We can define a function expDenote that is typed in terms of typeDenote.

```
Fixpoint expDenote t (e : exp t) : typeDenote t :=
  match e with
    | NConst n ⇒ n
    | Plus e1 e2 ⇒ expDenote e1 + expDenote e2
    | Eq e1 e2 ⇒
      if eq_nat_dec (expDenote e1) (expDenote e2)
        then true else false
```

```
    | BConst b ⇒ b
    | And e1 e2 ⇒ expDenote e1 && expDenote e2
    | If _ e' e1 e2 ⇒
      if expDenote e' then expDenote e1 else expDenote e2

    | Pair _ _ e1 e2 ⇒ (expDenote e1, expDenote e2)
    | Fst _ _ e' ⇒ fst (expDenote e')
    | Snd _ _ e' ⇒ snd (expDenote e')
  end.
```

The function definition is routine. In fact, it is less complicated than what we would write in ML or Haskell 98, since we do not need to worry about pushing final values in and out of an algebraic datatype. The only unusual thing is the use of an expression of the form `if` E `then true else false` in the Eq case. Remember that eq_nat_dec has a rich dependent type rather than a simple Boolean type. Coq's native `if` is overloaded to work on a test of any two-constructor type, so we can use `if` to build a simple Boolean from the **sumbool** that eq_nat_dec returns.

We can implement a constant folding function and prove it correct. It will be useful to write a function pairOut that checks if an **exp** of Prod type is a pair, returning its two components if so. Unsurprisingly, a first attempt leads to a type error.

```
Definition pairOut t1 t2 (e : exp (Prod t1 t2))
  : option (exp t1 × exp t2) :=
  match e in (exp (Prod t1 t2)) return option (exp t1 × exp t2) with
    | Pair _ _ e1 e2 ⇒ Some (e1, e2)
    | _ ⇒ None
  end.
```

```
Error: The reference t2 was not found in the current
environment
```

We run again into the problem of not being able to specify nonvariable arguments in `in` clauses. The problem would be unsolvable without the use of an `in` clause, since the result type of the `match` depends on an argument to **exp**. The solution is to use a more general type, as we did for hd. First, we define a type-valued function to use in assigning a type to pairOut.

```
Definition pairOutType (t : type) :=
  option (match t with
            | Prod t1 t2 ⇒ exp t1 × exp t2
```

```
    | _ ⇒ unit
  end).
```

When passed a type that is a product, pairOutType returns the final desired type. On any other input type, pairOutType returns the harmless **option unit**, since we do not care about extracting components of nonpairs. Now pairOut is easy to write.

```
Definition pairOut t (e : exp t) :=
  match e in (exp t) return (pairOutType t) with
    | Pair _ _ e1 e2 ⇒ Some (e1, e2)
    | _ ⇒ None
  end.
```

With pairOut available, we can write cfold in a straightforward way. There are really no surprises beyond that Coq verifies that this code has such an expressive type, given the small annotation burden. In some places, we see that Coq's `match` annotation inference is too smart for its own good, and we have to turn that inference off with explicit `return` clauses.

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

    | BConst b ⇒ BConst b
    | And e1 e2 ⇒
      let e1' := cfold e1 in
      let e2' := cfold e2 in
```

```
      match e1', e2' return exp Bool with
        | BConst b1, BConst b2 ⇒ BConst (b1 && b2)
        | _, _ ⇒ And e1' e2'
      end
    | If _ e e1 e2 ⇒
      let e' := cfold e in
      match e' with
        | BConst true ⇒ cfold e1
        | BConst false ⇒ cfold e2
        | _ ⇒ If e' (cfold e1) (cfold e2)
      end

    | Pair _ _ e1 e2 ⇒ Pair (cfold e1) (cfold e2)
    | Fst _ _ e ⇒
      let e' := cfold e in
      match pairOut e' with
        | Some p ⇒ fst p
        | None ⇒ Fst e'
      end
    | Snd _ _ e ⇒
      let e' := cfold e in
      match pairOut e' with
        | Some p ⇒ snd p
        | None ⇒ Snd e'
      end
  end.
```

The correctness theorem for cfold turns out to be easy to prove, once we get over one serious hurdle.

```
Theorem cfold_correct : ∀ t (e : exp t),
  expDenote e = expDenote (cfold e).
  induction e; crush.
```

The first remaining subgoal is

```
  expDenote (cfold e1) + expDenote (cfold e2) =
   expDenote
     match cfold e1 with
     | NConst n1 ⇒
         match cfold e2 with
         | NConst n2 ⇒ NConst (n1 + n2)
         | Plus _ _ ⇒ Plus (cfold e1) (cfold e2)
```

```
      | Eq _ _ ⇒ Plus (cfold e1) (cfold e2)
      | BConst _ ⇒ Plus (cfold e1) (cfold e2)
      | And _ _ ⇒ Plus (cfold e1) (cfold e2)
      | If _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Pair _ _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Fst _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      | Snd _ _ _ ⇒ Plus (cfold e1) (cfold e2)
      end
  | Plus _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Eq _ _ ⇒ Plus (cfold e1) (cfold e2)
  | BConst _ ⇒ Plus (cfold e1) (cfold e2)
  | And _ _ ⇒ Plus (cfold e1) (cfold e2)
  | If _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Pair _ _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Fst _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  | Snd _ _ _ ⇒ Plus (cfold e1) (cfold e2)
  end
```

We would like to do a case analysis on cfold *e1*, and we attempt to do so in the way that has worked so far.

```
  destruct (cfold e1).
```

```
User error: e1 is used in hypothesis e
```

Coq gives us another cryptic error message. Like so many others, this one basically means that Coq is not able to build some proof about dependent types. It is hard to generate helpful and specific error messages for problems like this, since that would require some kind of understanding of the dependency structure of a piece of code. We will encounter many examples of case-specific tricks for recovering from errors like this one.

For the current proof, we can use a tactic *dep_destruct* defined in the CpdtTactics module from the source code to this book. General elimination/inversion of dependently typed hypotheses is undecidable, as shown by a simple reduction from the known-undecidable problem of higher-order unification, which has come up a few times already. The tactic *dep_destruct* makes a best effort to handle some common cases, relying upon the more primitive `dependent destruction` tactic that comes with Coq. Chapter 10 discusses the explicit manipulation of equality proofs that is behind `dependent destruction`'s implementation, but for now we treat it as a useful black box. (Chapter 12 shows

how `dependent destruction` forces us to make a larger philosophical commitment about our logic than we might like and gives some work-arounds.)

dep_destruct (cfold *e1*).

This successfully breaks the subgoal into five new subgoals, one for each constructor of **exp** that could produce an **exp** Nat. Note that *dep_destruct* is successful in ruling out the other cases automatically, in effect automating some of the work that we did manually in implementing functions like hd and pairOut.

This is the only new technique we need to learn to complete the proof. A short automated proof uses Ltac features that are explained in Chapter 14.

```
Restart.

induction e; crush;
  repeat (match goal with
            | [ ⊢ context[match cfold ?E with NConst _ ⇒ _
                                            | _ ⇒ _ end] ] ⇒
              dep_destruct (cfold E)
            | [ ⊢ context[match pairOut (cfold ?E) with
                                         Some _ ⇒ _ | None ⇒ _ end] ] ⇒
              dep_destruct (cfold E)
            | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
          end; crush).
Qed.
```

With this example, we get a first taste of how to build automated proofs that adapt automatically to changes in function definitions.

8.4 Dependently Typed Red-Black Trees

Red-black trees are a favorite purely functional data structure with an interesting invariant. We can use dependent types to guarantee that operations on red-black trees preserve the invariant. For simplicity, we specialize the red-black trees to represent sets of **nat**s.

```
Inductive color : Set := Red | Black.

Inductive rbtree : color → nat → Set :=
| Leaf : rbtree Black 0
| RedNode : ∀ n, rbtree Black n → nat → rbtree Black n
  → rbtree Red n
| BlackNode : ∀ c1 c2 n, rbtree c1 n → nat → rbtree c2 n
```

$\rightarrow$ **rbtree** Black (S n).

A value of type **rbtree** c d is a red-black tree with root of color c and a black depth d. The latter property means that there are exactly d black-colored nodes on any path from the root to a leaf.

At first, it can be unclear that this choice of type indices tracks any useful property, so we prove that every red-black tree is balanced. We phrase the theorem in terms of a depth-calculating function that ignores the extra information in the types. It is useful to parameterize this function over a combining operation so that we can reuse the same code to calculate the minimum or maximum height among all paths from root to leaf.

```
Require Import Max Min.

Section depth.
  Variable f : nat → nat → nat.

  Fixpoint depth c n (t : rbtree c n) : nat :=
    match t with
      | Leaf ⇒ 0
      | RedNode _ t1 _ t2 ⇒ S (f (depth t1) (depth t2))
      | BlackNode _ _ _ t1 _ t2 ⇒ S (f (depth t1) (depth t2))
    end.
End depth.
```

The proof of balancedness decomposes naturally into a lower bound and an upper bound. We prove the lower bound first. Unsurprisingly, a tree's black depth provides such a bound on the minimum path length. We use the richly typed procedure min_dec to do case analysis on whether min X Y equals X or Y.

```
Check min_dec.

min_dec
     : ∀ n m : nat, {min n m = n} + {min n m = m}

Theorem depth_min : ∀ c n (t : rbtree c n), depth min t ≥ n.
  induction t; crush;
    match goal with
      | [ ⊢ context[min ?X ?Y] ] ⇒ destruct (min_dec X Y)
    end; crush.
Qed.
```

There is an analogous upper-bound theorem based on black depth. Unfortunately, a symmetric proof script does not suffice to establish it.

```
Theorem depth_max : ∀ c n (t : rbtree c n), depth max t ≤ 2 × n + 1.
```

```
  induction t; crush;
    match goal with
      | [ ⊢ context[max ?X ?Y] ] ⇒ destruct (max_dec X Y)
    end; crush.
```

Two subgoals remain. One of them is

```
n : nat
t1 : rbtree Black n
n0 : nat
t2 : rbtree Black n
IHt1 : depth max t1 ≤ n + (n + 0) + 1
IHt2 : depth max t2 ≤ n + (n + 0) + 1
e : max (depth max t1) (depth max t2) = depth max t1
============================
 S (depth max t1) ≤ n + (n + 0) + 1
```

We see that *IHt1* is almost the fact we need, but it is not quite strong enough. We need to strengthen the induction hypothesis to get the proof to go through.

```
Abort.
```

In particular, we prove a lemma that provides a stronger upper bound for trees with black root nodes. We got stuck in a case about a red root node. Since red nodes have only black children, our IH strengthening enables us to finish the proof.

```
Lemma depth_max' : ∀ c n (t : rbtree c n),
  match c with
    | Red ⇒ depth max t ≤ 2 × n + 1
    | Black ⇒ depth max t ≤ 2 × n
  end.
  induction t; crush;
    match goal with
      | [ ⊢ context[max ?X ?Y] ] ⇒ destruct (max_dec X Y)
    end; crush;
    repeat (match goal with
              | [ H : context[match ?C with Red ⇒ _
                                          | Black ⇒ _ end] ⊢ _ ] ⇒
                destruct C
            end; crush).
Qed.
```

The original theorem follows easily from the lemma. We use the tactic `generalize` *pf*, which, when *pf* proves the proposition P, changes the

goal from Q to $P \to Q$. This transformation is useful because it makes the truth of P manifest syntactically, so that automation machinery can rely on P, even if that machinery is not smart enough to establish P on its own.

```
Theorem depth_max : ∀ c n (t : rbtree c n), depth max t ≤ 2 × n + 1.
  intros; generalize (depth_max' t); destruct c; crush.
Qed.
```

The final balance theorem establishes that the minimum and maximum path lengths of any tree are within a factor of 2 of each other.

```
Theorem balanced : ∀ c n (t : rbtree c n),
  2 × depth min t + 1 ≥ depth max t.
  intros; generalize (depth_min t); generalize (depth_max t);
    crush.
Qed.
```

Now we are ready to implement an example operation on the trees, insertion. Insertion can be thought of as breaking the tree invariants locally but then rebalancing. In particular, in intermediate states we find red nodes that may have red children. The type **rtree** captures the idea of such a node, continuing to track black depth as a type index.

```
Inductive rtree : nat → Set :=
| RedNode' : ∀ c1 c2 n, rbtree c1 n → nat → rbtree c2 n → rtree n.
```

Before starting to define insert, we define predicates capturing when a data value is in the set represented by a normal or possibly invalid tree.

```
Section present.
  Variable x : nat.

  Fixpoint present c n (t : rbtree c n) : Prop :=
    match t with
      | Leaf ⇒ False
      | RedNode _ a y b ⇒ present a ∨ x = y ∨ present b
      | BlackNode _ _ _ a y b ⇒ present a ∨ x = y ∨ present b
    end.

  Definition rpresent n (t : rtree n) : Prop :=
    match t with
      | RedNode' _ _ _ a y b ⇒ present a ∨ x = y ∨ present b
    end.
End present.
```

Insertion relies on two balancing operations. It is useful to give types to these operations using a relative of the subset types discussed in Chapter 6. While subset types let us pair a value with a proof about that value, here we want to pair a value with another nonproof dependently typed value. The **sigT** type fills this role.

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

Notation Scope
"{ x : A & P }" := sigT (fun x : A ⇒ P)

Print sigT.

Inductive sigT (A : Type) (P : A → Type) : Type :=
    existT : ∀ x : A, P x → sigT P
```

It is helpful to define a concise notation for the constructor of **sigT**.

```
Notation "{< x >}" := (existT _ _ x).
```

Each balance function is used to construct a new tree whose keys include the keys of two input trees as well as a new key. One of the two input trees may violate the red-black alternation invariant (that is, it has an **rtree** type), while the other tree is known to be valid. Crucially, the two input trees have the same black depth.

A balance operation may return a tree whose root is of either color. Thus, we use a **sigT** type to package the result tree with the color of its root. Here is the definition of the first balance operation, which applies when the possibly invalid **rtree** belongs to the left of the valid **rbtree**.

A quick word of encouragement: After writing this code, even I do not understand the precise details of how balancing works. I consulted Chris Okasaki's paper "Red-Black Trees in a Functional Setting" [30] and transcribed the code to use dependent types. Luckily, the details are not so important here; types alone indicate that insertion preserves balancedness, and we will prove that insertion produces trees containing the right keys.

```
Definition balance1 n (a : rtree n) (data : nat) c2 :=
  match a in rtree n return rbtree c2 n
    → { c : color & rbtree c (S n) } with
    | RedNode' _ c0 _ t1 y t2 ⇒
      match t1 in rbtree c n return rbtree c0 n → rbtree c2 n
        → { c : color & rbtree c (S n) } with
        | RedNode _ a x b ⇒ fun c d ⇒
          {<RedNode (BlackNode a x b) y (BlackNode c data d)>}
        | t1' ⇒ fun t2 ⇒
          match t2 in rbtree c n return rbtree Black n
```

```
          → rbtree c2 n → { c : color & rbtree c (S n) } with
          | RedNode _ b x c ⇒ fun a d ⇒
            {<RedNode (BlackNode a y b) x (BlackNode c data d)>}
          | b ⇒ fun a t ⇒ {<BlackNode (RedNode a y b) data t>}
        end t1'
    end t2
end.
```

We apply a trick that I call the *convoy pattern*. Recall that `match` annotations only make it possible to describe a dependence of a `match` *result type* on the discriminee. There is no automatic refinement of the types of free variables. However, it is possible to effect such a refinement by finding a way to encode free variable type dependencies in the `match` result type, so that a `return` clause can express the connection.

In particular, we can extend the `match` to return *functions over the free variables whose types we want to refine*. In the case of balance1, we only want to refine the type of one tree variable at a time. We match on one subtree of a node, and we want the type of the other subtree to be refined based on what we learn. We indicate this with a `return` clause starting like **rbtree** _ $n \rightarrow$ ..., where n is bound in an `in` pattern. Such a `match` expression is applied immediately to the "old version" of the variable to be refined, and the type checker is satisfied.

Here is the symmetric function balance2, for cases where the possibly invalid tree appears on the right rather than on the left:

```
Definition balance2 n (a : rtree n) (data : nat) c2 :=
  match a in rtree n
    return rbtree c2 n → { c : color & rbtree c (S n) } with
    | RedNode' _ c0 _ t1 z t2 ⇒
      match t1 in rbtree c n return rbtree c0 n → rbtree c2 n
        → { c : color & rbtree c (S n) } with
        | RedNode _ b y c ⇒ fun d a ⇒
          {<RedNode (BlackNode a data b) y (BlackNode c z d)>}
        | t1' ⇒ fun t2 ⇒
          match t2 in rbtree c n return rbtree Black n
            → rbtree c2 n → { c : color & rbtree c (S n) } with
            | RedNode _ c z' d ⇒ fun b a ⇒
              {<RedNode (BlackNode a data b) z
                (BlackNode c z' d)>}
            | b ⇒ fun a t ⇒ {<BlackNode t data (RedNode a z b)>}
          end t1'
      end t2
  end.
```

Now we are almost ready to write an insert function. First, we enter a section that declares a variable x for the key we want to insert.

```
Section insert.
  Variable x : nat.
```

Most of the work of insertion is done by a helper function ins, whose return types are expressed using a type-level function insResult.

```
  Definition insResult c n :=
    match c with
      | Red ⇒ rtree n
      | Black ⇒ { c' : color & rbtree c' n }
    end.
```

That is, inserting into a tree with root color c and black depth n, the variety of tree we get out depends on c. If we started with a red root, then we get back a possibly invalid tree of depth n. If we started with a black root, we get back a valid tree of depth n with a root node of an arbitrary color.

Here is the definition of ins. Again, we do not want to dwell on the functional details.

```
  Fixpoint ins c n (t : rbtree c n) : insResult c n :=
    match t with
      | Leaf ⇒ {< RedNode Leaf x Leaf >}
      | RedNode _ a y b ⇒
        if le_lt_dec x y
          then RedNode' (projT2 (ins a)) y b
          else RedNode' a y (projT2 (ins b))
      | BlackNode c1 c2 _ a y b ⇒
        if le_lt_dec x y
          then
            match c1 return insResult c1 _ → _ with
              | Red ⇒ fun ins_a ⇒ balance1 ins_a y b
              | _ ⇒ fun ins_a ⇒ {< BlackNode (projT2 ins_a) y b >}
            end (ins a)
          else
            match c2 return insResult c2 _ → _ with
              | Red ⇒ fun ins_b ⇒ balance2 ins_b y a
              | _ ⇒ fun ins_b ⇒ {< BlackNode a y (projT2 ins_b) >}
            end (ins b)
    end.
```

The one new trick is a variation of the convoy pattern. In each of the last two pattern matches, we want to take advantage of the typing connection between the trees a and b. We might naïvely apply the convoy pattern directly on a in the first `match` and on b in the second. This satisfies the type checker per se, but it does not satisfy the termination checker. Inside each `match`, we would be calling ins recursively on a locally bound variable. The termination checker is not smart enough to trace the data flow into that variable, so the checker does not know that this recursive argument is smaller than the original argument. We make this fact clearer by applying the convoy pattern on *the result of a recursive call* rather than just on that call's argument.

We are almost done defining insert. We just need a few more definitions of nonrecursive functions. First, we need to give the final characterization of insert's return type. Inserting into a red-rooted tree gives a black-rooted tree whose black depth has increased, and inserting into a black-rooted tree gives a tree whose black depth has stayed the same and whose root is an arbitrary color.

```
Definition insertResult c n :=
  match c with
    | Red ⇒ rbtree Black (S n)
    | Black ⇒ { c' : color & rbtree c' n }
  end.
```

A simple cleanup procedure translates insResults into insertResults.

```
Definition makeRbtree c n : insResult c n → insertResult c n :=
  match c with
    | Red ⇒ fun r ⇒
      match r with
        | RedNode' _ _ _ a x b ⇒ BlackNode a x b
      end
    | Black ⇒ fun r ⇒ r
  end.
```

We modify Coq's default choice of implicit arguments for makeRbtree so that we do not need to specify the c and n arguments explicitly in later calls.

```
Implicit Arguments makeRbtree [c n].
```

Finally, we define insert as a simple composition of ins and makeRbtree.

```
Definition insert c n (t : rbtree c n) : insertResult c n :=
  makeRbtree (ins t).
```

As noted earlier, the type of insert guarantees that it outputs balanced trees whose depths have not increased too much. We also want to know that insert operates correctly on trees interpreted as finite sets, so we finish this section with a proof of that fact.

```
Section present.
  Variable z : nat.
```

The variable z stands for an arbitrary key. We reason about z's presence in particular trees. As usual, outside the section the theorems we prove quantify over all possible keys, giving us the facts we wanted.

We start by proving the correctness of the balance operations. It is useful to define a custom tactic *present_balance* that encapsulates the reasoning common to the two proofs. We use the keyword Ltac to assign a name to a proof script. This particular script just iterates between *crush* and identification of a tree that is being pattern-matched and should be destructed.

```
Ltac present_balance :=
  crush;
  repeat (match goal with
            | [ _ : context[match ?T with Leaf ⇒ _
                                        | _ ⇒ _ end] ⊢ _ ] ⇒
              dep_destruct T
            | [ ⊢ context[match ?T with Leaf ⇒ _
                                      | _ ⇒ _ end] ] ⇒
              dep_destruct T
          end; crush).
```

The balance correctness theorems are simple first-order logic equivalences, where we use the function projT2 to project the payload of a sigT value.

```
Lemma present_balance1 : ∀ n (a : rtree n) (y : nat) c2
  (b : rbtree c2 n),
  present z (projT2 (balance1 a y b))
  ↔ rpresent z a ∨ z = y ∨ present z b.
  destruct a; present_balance.
Qed.

Lemma present_balance2 : ∀ n (a : rtree n) (y : nat) c2
  (b : rbtree c2 n),
  present z (projT2 (balance2 a y b))
  ↔ rpresent z a ∨ z = y ∨ present z b.
  destruct a; present_balance.
Qed.
```

To state the theorem for ins, it is useful to define a new type-level function, since ins returns different result types based on the type indices passed to it. Recall that x is the section variable standing for the key we are inserting.

```
Definition present_insResult c n :=
  match c return (rbtree c n → insResult c n → Prop) with
    | Red ⇒ fun t r ⇒ rpresent z r ↔ z = x ∨ present z t
    | Black ⇒ fun t r ⇒ present z (projT2 r) ↔
      z = x ∨ present z t
  end.
```

Now the statement and proof of the ins correctness theorem are straightforward. We proceed by induction on the structure of a tree, followed by finding case analysis opportunities on expressions being analyzed in `if` or `match` expressions. After that, we pattern-match to find opportunities to use the theorems we proved about balancing. Finally, we identify two variables that are asserted by some hypothesis to be equal, and we use that hypothesis to replace one variable with the other everywhere.

```
Theorem present_ins : ∀ c n (t : rbtree c n),
  present_insResult t (ins t).
  induction t; crush;
    repeat (match goal with
              | [ _ : context[if ?E then _ else _] ⊢ _ ] ⇒
                destruct E
              | [ ⊢ context[if ?E then _ else _] ] ⇒
                destruct E
              | [ _ : context[match ?C with Red ⇒ _
                                          | Black ⇒ _ end]
                  ⊢ _ ] ⇒ destruct C
            end; crush);
    try match goal with
          | [ _ : context[balance1 ?A ?B ?C] ⊢ _ ] ⇒
            generalize (present_balance1 A B C)
        end;
    try match goal with
          | [ _ : context[balance2 ?A ?B ?C] ⊢ _ ] ⇒
            generalize (present_balance2 A B C)
        end;
    try match goal with
          | [ ⊢ context[balance1 ?A ?B ?C] ] ⇒
            generalize (present_balance1 A B C)
```

```
      end;
    try match goal with
          | [ ⊢ context[balance2 ?A ?B ?C] ] ⇒
            generalize (present_balance2 A B C)
      end;
    crush;
      match goal with
        | [ z : nat, x : nat ⊢ _ ] ⇒
          match goal with
            | [ H : z = x ⊢ _ ] ⇒ rewrite H in *; clear H
          end
      end;
      tauto.
  Qed.
```

The hard work is done. The most readable way to state correctness of insert involves splitting the property into two color-specific theorems. We write a tactic to encapsulate the reasoning steps that work to establish both facts.

```
  Ltac present_insert :=
    unfold insert; intros n t; inversion t;
      generalize (present_ins t); simpl;
        dep_destruct (ins t); tauto.

  Theorem present_insert_Red : ∀ n (t : rbtree Red n),
    present z (insert t)
    ↔ (z = x ∨ present z t).
    present_insert.
  Qed.

  Theorem present_insert_Black : ∀ n (t : rbtree Black n),
    present z (projT2 (insert t))
    ↔ (z = x ∨ present z t).
    present_insert.
  Qed.
End present.
End insert.
```

We can generate executable OCaml code with the command `Recursive Extraction insert`, which also automatically outputs the OCaml versions of all of insert's dependencies. In previous extractions, we wound up with clean OCaml code. Here, we find uses of `Obj.magic`, OCaml's unsafe cast operator for tweaking the apparent type of an expression in an arbitrary way. Casts appear for this example because

the return type of insert depends on the *value* of the function's argument, a pattern that OCaml cannot handle. Coq's type system is much more expressive than OCaml's, so such casts are unavoidable in general. Since the OCaml type checker is no longer checking full safety of programs, we must rely on Coq's extractor to use casts only in provably safe ways.

8.5 A Certified Regular Expression Matcher

Another interesting example is regular expressions with dependent types that express which predicates over strings particular expressions implement. We can then assign a dependent type to a regular expression matching function, guaranteeing that it always decides the string property that we expect it to decide.

Before defining the syntax of expressions, it is helpful to define an inductive type capturing the meaning of the Kleene star. That is, a string s matches regular expression **star** e if and only if s can be decomposed into a sequence of substrings that all match e. We use Coq's string support, which comes through a combination of the String library and some parsing notations built into Coq. Operators like ++ and functions like length that we know from lists are defined again for strings. Notation scopes help us control which versions we want to use in particular contexts.

```
Require Import Ascii String.
Open Scope string_scope.

Section star.
  Variable P : string → Prop.

  Inductive star : string → Prop :=
  | Empty : star ""
  | Iter : ∀ s1 s2,
    P s1
    → star s2
    → star (s1 ++ s2).
End star.
```

Now we can make a first attempt at defining a **regexp** type that is indexed by predicates on strings, such that the index of a **regexp** tells us which language (string predicate) it recognizes. The following definition, which is restricted to constant characters and concatenation, seems reasonable. We use the constructor String, which is the analogue of list cons for the type **string**, where "" is like list nil.

```
Inductive regexp : (string → Prop) → Set :=
| Char : ∀ ch : ascii,
  regexp (fun s ⇒ s = String ch "")
| Concat : ∀ (P1 P2 : string → Prop) (r1 : regexp P1)
  (r2 : regexp P2),
  regexp (fun s ⇒ ∃ s1, ∃ s2, s = s1 ++ s2 ∧ P1 s1 ∧ P2 s2).
```

```
User error: Large non-propositional inductive types must be
in Type
```

What is a large inductive type? In Coq, it is an inductive type with a constructor that quantifies over some type of type `Type`. We have not worked with `Type` very much to this point. Every term of the Calculus of Inductive Constructions has a type, including `Set` and `Prop`, which are assigned type `Type`. The type **string** → `Prop` from the failed definition also has type `Type`.

It turns out that allowing large inductive types in `Set` leads to contradictions when combined with certain kinds of classical logic reasoning. Thus, by default, such types are ruled out. There is a simple fix for the **regexp** definition, which is to place the new type in `Type`. While fixing the problem, we also expand the list of constructors to cover the remaining regular expression operators.

```
Inductive regexp : (string → Prop) → Type :=
| Char : ∀ ch : ascii,
  regexp (fun s ⇒ s = String ch "")
| Concat : ∀ P1 P2 (r1 : regexp P1) (r2 : regexp P2),
  regexp (fun s ⇒ ∃ s1, ∃ s2, s = s1 ++ s2 ∧ P1 s1 ∧ P2 s2)
| Or : ∀ P1 P2 (r1 : regexp P1) (r2 : regexp P2),
  regexp (fun s ⇒ P1 s ∨ P2 s)
| Star : ∀ P (r : regexp P),
  regexp (star P).
```

Many theorems about strings are useful for implementing a certified regexp matcher, and few of them are in the String library. The book source includes statements, proofs, and hint commands for a handful of such omitted theorems.

A few auxiliary functions help in the final matcher definition. The function `split` is used to implement the regexp concatenation case.

```
Section split.
  Variables P1 P2 : string → Prop.
  Variable P1_dec : ∀ s, {P1 s} + {¬ P1 s}.
  Variable P2_dec : ∀ s, {P2 s} + {¬ P2 s}.
```

We require a choice of two arbitrary string predicates and functions for deciding them.

```
Variable s : string.
```

The computation takes place relative to a single fixed string, so it is easiest to make it a `Variable` rather than an explicit argument to the functions.

The function split' is the workhorse behind `split`. It searches through the possible ways of splitting s into two pieces, checking the two predicates against each such pair. The execution of split' progresses right to left, from splitting all of s into the first piece to splitting all of s into the second piece. It takes an extra argument, n, which specifies how far along we are in this search process.

```
Definition split' : ∀ n : nat, n ≤ length s
  → {∃ s1, ∃ s2, length s1 ≤ n ∧ s1 ++ s2 = s ∧ P1 s1 ∧ P2 s2}
  + {∀ s1 s2, length s1 ≤ n → s1 ++ s2 = s → ¬ P1 s1 ∨ ¬ P2 s2}.
  refine (fix F (n : nat) : n ≤ length s
    → {∃ s1, ∃ s2,
      length s1 ≤ n ∧ s1 ++ s2 = s ∧ P1 s1 ∧ P2 s2}
    + {∀ s1 s2,
      length s1 ≤ n → s1 ++ s2 = s → ¬ P1 s1 ∨ ¬ P2 s2} :=
    match n with
      | O ⇒ fun _ ⇒ Reduce (P1_dec "" && P2_dec s)
      | S n' ⇒ fun _ ⇒ (P1_dec (substring 0 (S n') s)
          && P2_dec (substring (S n') (length s - S n') s))
        || F n' _
    end); clear F; crush; eauto 7;
  match goal with
    | [ _ : length ?S ≤ 0 ⊢ _ ] ⇒ destruct S
    | [ _ : length ?S' ≤ S ?N ⊢ _ ] ⇒
      destruct (eq_nat_dec (length S') (S N))
  end; crush.
Defined.
```

There is one subtle point in the split' code that is worth mentioning. The main body of the function is a `match` on n. In the case where n is known to be S n', we write S n' in several places where we might be tempted to write n. However, without further work to craft proper `match` annotations, the type checker does not use the equality between n and S n'. Thus, it is common to see patterns repeated in `match` case bodies in dependently typed Coq code. We can at least use a `let` expression to avoid copying the pattern more than once, replacing the first case body with

```
    | S n' ⇒ fun _ ⇒ let n := S n' in
      (P1_dec (substring 0 n s)
        && P2_dec (substring n (length s - n) s))
      || F n' _
```

The `split` function itself is trivial to implement in terms of split'. We just ask split' to begin its search with $n =$ length s.

```
Definition split : {∃ s1, ∃ s2, s = s1 ++ s2 ∧ P1 s1 ∧ P2 s2}
  + {∀ s1 s2, s = s1 ++ s2 → ¬ P1 s1 ∨ ¬ P2 s2}.
  refine (Reduce (split' (n := length s) _)); crush; eauto.
Defined.
End split.

Implicit Arguments split [P1 P2].
```

One more helper function will come in handy: dec_star, for implementing another linear search through ways of splitting a string, this time for implementing the Kleene star.

```
Section dec_star.
  Variable P : string → Prop.
  Variable P_dec : ∀ s, {P s} + {¬ P s}.
```

Some new lemmas and hints about the **star** type family are included in the book source at this point.

The function dec_star'' implements a single iteration of the star. That is, it tries to find a string prefix matching P, and it calls a parameter function on the remainder of the string.

```
  Section dec_star''.
    Variable n : nat.
```

Variable n is the length of the prefix of s that we have already processed.

```
    Variable P' : string → Prop.
    Variable P'_dec : ∀ n' : nat, n' > n
      → {P' (substring n' (length s - n') s)}
      + {¬ P' (substring n' (length s - n') s)}.
```

When we use dec_star'', we instantiate P'_dec with a function for continuing the search for more instances of P in s.

Now we come to dec_star'' itself. It takes as an input a natural l that records how much of the string has been searched so far, as we did for split'. The return type expresses that dec_star'' is looking for an index into s that splits s into a nonempty prefix and a suffix, such that the prefix satisfies P and the suffix satisfies P'.

```
Definition dec_star'' : ∀ l : nat,
  {∃ l', S l' ≤ l
    ∧ P (substring n (S l') s) ∧ P' (substring (n + S l')
      (length s - (n + S l')) s)}
  + {∀ l', S l' ≤ l
    → ¬ P (substring n (S l') s)
    ∨ ¬ P' (substring (n + S l') (length s - (n + S l')) s)}.
  refine (fix F (l : nat) : {∃ l', S l' ≤ l
      ∧ P (substring n (S l') s) ∧ P' (substring (n + S l')
        (length s - (n + S l')) s)}
    + {∀ l', S l' ≤ l
      → ¬ P (substring n (S l') s)
      ∨ ¬ P' (substring (n + S l') (length s - (n + S l')) s)} :=
    match l with
      | O ⇒ _
      | S l' ⇒
        (P_dec (substring n (S l') s)
          && P'_dec (n' := n + S l') _)
        || F l'
    end); clear F; crush; eauto 7;
    match goal with
      | [ H : ?X ≤ S ?Y ⊢ _ ] ⇒
        destruct (eq_nat_dec X (S Y)); crush
    end.
  Defined.
End dec_star''.
```

The work of dec_star'' is nested inside another linear search by dec_star', which provides the final functionality we need, but for arbitrary suffixes of s rather than just for s overall.

```
Definition dec_star' : ∀ n n' : nat, length s - n' ≤ n
  → {star P (substring n' (length s - n') s)}
  + {¬ star P (substring n' (length s - n') s)}.
  refine (fix F (n n' : nat) : length s - n' ≤ n
    → {star P (substring n' (length s - n') s)}
    + {¬ star P (substring n' (length s - n') s)} :=
    match n with
      | O ⇒ fun _ ⇒ Yes
      | S n'' ⇒ fun _ ⇒
        le_gt_dec (length s) n'
        || dec_star'' (n := n') (star P)
          (fun n0 _ ⇒ Reduce (F n'' n0 _)) (length s - n')
```

```
    end); clear F; crush; eauto;
  match goal with
    | [ H : star _ _ ⊢ _ ] ⇒ apply star_substring_inv in H;
      crush; eauto
  end;
  match goal with
    | [ H1 : _ < _ - _, H2 : ∀ l' : nat, _ ≤ _ - _ → _ ⊢ _ ] ⇒
      generalize (H2 _ (lt_le_S _ _ H1)); tauto
  end.
Defined.
```

Finally, we have dec_star, defined by straightforward reduction from dec_star'.

```
Definition dec_star : {star P s} + {¬ star P s}.
  refine (Reduce (dec_star' (n := length s) 0 _)); crush.
Defined.
End dec_star.
```

With these helper functions completed, the implementation of the matches function is straightforward. We only need one small piece of specific tactic work beyond *crush*.

```
Definition matches : ∀ P (r : regexp P) s, {P s} + {¬ P s}.
  refine (fix F P (r : regexp P) s : {P s} + {¬ P s} :=
    match r with
      | Char ch ⇒ string_dec s (String ch "")
      | Concat _ _ r1 r2 ⇒ Reduce (split (F _ r1) (F _ r2) s)
      | Or _ _ r1 r2 ⇒ F _ r1 s || F _ r2 s
      | Star _ r ⇒ dec_star _ _ _
    end); crush;
  match goal with
    | [ H : _ ⊢ _ ] ⇒ generalize (H _ _ (eq_refl _))
  end; tauto.
Defined.
```

It is interesting to consider alternative implementations of matches. Dependent types offer much latitude in how specific correctness properties may be encoded with types. For instance, we could have made **regexp** a nonindexed inductive type, along the lines of what is possible in traditional ML and Haskell. We could then have implemented a recursive function to map **regexp**s to their intended meanings, much as was done with types and programs in other examples. That style is compatible with the `refine`-based approach that we have used here,

and it might be an interesting exercise to redo the code from this subsection in that style or some other encoding of the reader's choice. The main advantage of indexed inductive types is that they generally lead to the least code.

Many regular expression-matching problems are easy to test. The reader could run each of the following queries to verify that it gives the correct answer. We use evaluation strategy `hnf` to reduce each term to *head-normal form*, where the datatype constructor used to build its value is known. (Further reduction would involve wasteful simplification of proof terms justifying the answers of the procedures.)

```
Example a_star := Star (Char "a"%char).
Eval hnf in matches a_star "".
Eval hnf in matches a_star "a".
Eval hnf in matches a_star "b".
Eval hnf in matches a_star "aa".
```

Evaluation inside Coq does not scale very well, so it is easy to build other tests that run for hours or more. Such cases are better suited to execution with the extracted OCaml code.