7 General Recursion

Termination of all programs is a crucial property of Gallina. Nonterminating programs introduce logical inconsistency, that is, any theorem can be proved with an infinite loop. Coq uses a small set of conservative, syntactic criteria to check termination of all recursive definitions. These criteria are insufficient to support the natural encodings of a variety of important programming idioms. Further, since Coq makes it so convenient to encode mathematics computationally, with functional programs, we may want to employ more complicated recursion in mathematical definitions.

What exactly are the Coq criteria for checking termination? For *recursive* definitions, recursive calls are only allowed on *syntactic subterms* of the original primary argument, a restriction known as *primitive recursion.* In fact, Coq's handling of reflexive inductive types (those defined in terms of functions returning the same type) gives a bit more flexibility than in traditional primitive recursion, but the term is still applied commonly. Chapter 5 showed how *co-recursive* definitions are checked against a syntactic guardedness condition that guarantees productivity.

Many natural recursion patterns satisfy neither condition. For instance, in the simple running example in this chapter, we will study three different approaches to more flexible recursion, two of which support definitions that may fail to terminate on certain inputs without any up-front characterization of which inputs those may be.

The problem here is not as fundamental as it may appear. The final example of Chapter 5 demonstrated *deep embedding* of the syntax and semantics of a programming language. That is, it gave a mathematical definition of a language of programs and their meanings. This language clearly admitted nontermination, and we could think of writing all sophisticated recursive functions with such explicit syntax types. However, that would forfeit Coq's very good built-in support for reasoning about Gallina programs. It is preferable to use *shallow embedding*, where informal constructs are modeled by encoding them as normal Gallina programs. Each of the three techniques of this chapter follows that style.

7.1 Well-Founded Recursion

The essence of terminating recursion is that there are no infinite chains of nested recursive calls. This intuition is commonly mapped to the mathematical idea of a *well-founded relation*, and the associated standard technique in Coq is *well-founded recursion.* The syntactic subterm relation that Coq applies by default is well-founded, but many cases demand alternative well-founded relations. To demonstrate, let us see where we get stuck on attempting a standard merge sort implementation.

```
Section mergeSort.
  Variable A : Type.
  Variable le : A → A → bool.
```

We have a set equipped with some less-than-or-equal-to test.

A standard function inserts an element into a sorted list, preserving sortedness.

```
  Fixpoint insert (x : A) (ls : list A) : list A :=
    match ls with
      | nil ⇒ x :: nil
      | h :: ls' ⇒
        if le x h
          then x :: ls
          else h :: insert x ls'
    end.
```

We also need a function to merge two sorted lists. (We use a less efficient implementation than usual because the more efficient implementation already forces us to think about well-founded recursion, whereas here we are only interested in setting up the example of merge sort.)

```
  Fixpoint merge (ls1 ls2 : list A) : list A :=
    match ls1 with
      | nil ⇒ ls2
      | h :: ls' ⇒ insert h (merge ls' ls2)
    end.
```

The last helper function for classic merge sort is the one that follows, to split a list arbitrarily into two pieces of approximately equal length.

```
Fixpoint split (ls : list A) : list A × list A :=
  match ls with
    | nil ⇒ (nil, nil)
    | h :: nil ⇒ (h :: nil, nil)
    | h1 :: h2 :: ls' ⇒
      let (ls1, ls2) := split ls' in
        (h1 :: ls1, h2 :: ls2)
  end.
```

Now, let us try to write the final sorting function, using a natural number $\leq$ test leb from the standard library.

```
Fixpoint mergeSort (ls : list A) : list A :=
  if leb (length ls) 1
    then ls
    else let lss := split ls in
      merge (mergeSort (fst lss)) (mergeSort (snd lss)).
```

```
Recursive call to mergeSort has principal argument equal to
"fst (split ls)" instead of a subterm of "ls".
```

The definition is rejected for not following the simple primitive recursion criterion. In particular, it is not apparent that recursive calls to mergeSort are syntactic subterms of the original argument *ls*; indeed, they are not, yet we know this is a well-founded recursive definition.

To produce an acceptable definition, we need to choose a well-founded relation and prove that mergeSort respects it. A good starting point is an examination of how well-foundedness is formalized in the Coq standard library.

```
Print well_founded.
```

```
well_founded =
fun (A : Type) (R : A → A → Prop) ⇒ ∀ a : A, Acc R a
```

The bulk of the definitional work devolves to the *accessibility* relation **Acc**, whose definition we may also examine.

```
Print Acc.
```

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

In prose, an element x is accessible for a relation R if every element "less than" x according to R is also accessible. Since **Acc** is defined inductively, we know that any accessibility proof involves a finite chain of invocations, in a certain sense that we can make formal. Building on the examples from Chapter 5, let us define a co-inductive relation that is closer to the usual informal notion of "absence of infinite decreasing chains."

```
CoInductive infiniteDecreasingChain A (R : A → A → Prop)
  : stream A → Prop :=
| ChainCons : ∀ x y s, infiniteDecreasingChain R (Cons y s)
  → R y x
  → infiniteDecreasingChain R (Cons x (Cons y s)).
```

We can now prove that any accessible element cannot be the beginning of any infinite decreasing chain.

```
Lemma noBadChains' : ∀ A (R : A → A → Prop) x, Acc R x
  → ∀ s, ¬infiniteDecreasingChain R (Cons x s).
  induction 1; crush;
    match goal with
      | [ H : infiniteDecreasingChain _ _ ⊢ _ ] ⇒
        inversion H; eauto
    end.
Qed.
```

From here, the absence of infinite decreasing chains in well-founded sets is immediate.

```
Theorem noBadChains : ∀ A (R : A → A → Prop), well_founded R
  → ∀ s, ¬infiniteDecreasingChain R s.
  destruct s; apply noBadChains'; auto.
Qed.
```

Absence of infinite decreasing chains implies absence of infinitely nested recursive calls, for any recursive definition that respects the well-founded relation. The Fix combinator from the standard library formalizes that intuition.

```
Check Fix.
```

```
Fix
     : ∀ (A : Type) (R : A → A → Prop),
       well_founded R →
       ∀ P : A → Type,
       (∀ x : A, (∀ y : A, R y x → P y) → P x) →
       ∀ x : A, P x
```

A call to Fix must present a relation R and a proof of its well-foundedness. The next argument, P, is the possibly dependent range type of the function we build; the domain A of R is the function's domain. The subsequent argument has this type:

$$\forall\ x : A,\ (\forall\ y : A,\ R\ y\ x \rightarrow P\ y) \rightarrow P\ x$$

This is an encoding of the function body. The input x stands for the function argument, and the next input stands for the function we are defining. Recursive calls are encoded as calls to the second argument, whose type tells us it expects a value y and a proof that y is "less than" x, according to R. In this way, we enforce the well-foundedness restriction on recursive calls.

The rest of `Fix`'s type tells us that it returns a function of exactly the type we expect, so we are now ready to use it to implement mergeSort. Notice that `Fix` has a dependent type of the sort shown in Chapter 6.

Before writing mergeSort, we need to settle on a well-founded relation. The right one for this example is based on lengths of lists.

```
Definition lengthOrder (ls1 ls2 : list A) :=
  length ls1 < length ls2.
```

We must prove that the relation is truly well-founded. To save space, we skip right to automated proof scripts; the details of the principles behind such scripts are given in Part III of the book. (Readers may still replace semicolons with periods and newlines to step through these scripts interactively.)

```
Hint Constructors Acc.

Lemma lengthOrder_wf' : ∀ len, ∀ ls, length ls ≤ len
  → Acc lengthOrder ls.
  unfold lengthOrder; induction len; crush.
Defined.

Theorem lengthOrder_wf : well_founded lengthOrder.
  red; intro; eapply lengthOrder_wf'; eauto.
Defined.
```

Notice that these proofs end with `Defined`, not `Qed`. Recall that `Defined` marks the theorems as transparent, so that the details of their proofs may be used during program execution. Why could such details possibly matter for computation? It turns out that `Fix` satisfies the primitive recursion restriction by declaring itself as *recursive in the structure of **Acc** proofs*. This is possible because **Acc** proofs follow a predictable inductive structure. We must do work, as in the last theorem's proof, to establish that all elements of a type belong to **Acc**, but the automatic unwinding of those proofs during recursion is straightforward. If the proof ended with `Qed`, the proof details would be hidden from computation, in which case the unwinding process would get stuck.

To justify the two recursive mergeSort calls, we also need to prove that `split` respects the lengthOrder relation. These proofs, too, must be kept transparent, to avoid the stuckness of `Fix` evaluation. We use the syntax @*foo* to reference identifier *foo* with its implicit argument behavior turned off. The following proof uses Ltac features that are explained in Chapter 14.

```
Lemma split_wf : ∀ len ls, 2 ≤ length ls ≤ len
  → let (ls1, ls2) := split ls in
    lengthOrder ls1 ls ∧ lengthOrder ls2 ls.
  unfold lengthOrder; induction len; crush;
    do 2 (destruct ls; crush);
    destruct (le_lt_dec 2 (length ls));
      repeat (match goal with
                   | [ _ : length ?E < 2 ⊢ _ ] ⇒ destruct E
                   | [ _ : S (length ?E) < 2 ⊢ _ ] ⇒ destruct E
                   | [ IH : _ ⊢ context[split ?L] ] ⇒
                     specialize (IH L);
                       destruct (split L); destruct IH
                 end; crush).
Defined.

Ltac split_wf := intros ls ?; intros;
  generalize (@split_wf (length ls) ls);
  destruct (split ls); destruct 1; crush.

Lemma split_wf1 : ∀ ls, 2 ≤ length ls
  → lengthOrder (fst (split ls)) ls.
  split_wf.
Defined.

Lemma split_wf2 : ∀ ls, 2 ≤ length ls
  → lengthOrder (snd (split ls)) ls.
  split_wf.
Defined.

Hint Resolve split_wf1 split_wf2.
```

To write the function definition itself, we use the `refine` tactic as a convenient way to write a program that needs to manipulate proofs, without writing out those proofs manually. We also use a replacement le_lt_dec for leb that has a more interesting dependent type. (Note that we would not be able to complete the definition without this change, since `refine` will generate subgoals for the `if` branches based only on the *type* of the test expression, not its *value*.)

```
Definition mergeSort : list A → list A.
  refine (Fix lengthOrder_wf (fun _ ⇒ list A)
    (fun (ls : list A)
          (mergeSort : ∀ ls' : list A, lengthOrder ls' ls → list A) ⇒
          if le_lt_dec 2 (length ls)
            then let lss := split ls in
              merge (mergeSort (fst lss) _) (mergeSort (snd lss) _)
            else ls)); subst lss; eauto.
  Defined.
End mergeSort.
```

The important thing is that it is now easy to evaluate calls to mergeSort.

```
Eval compute in mergeSort leb (1 :: 2 :: 36 :: 8 :: 19 :: nil).
  = 1 :: 2 :: 8 :: 19 :: 36 :: nil
```

Since the subject of this chapter is how to define functions with unusual recursion structure, we do not prove any further correctness theorems about mergeSort, instead proving only that mergeSort has the expected computational behavior for all inputs, not merely the one just tested.

```
Theorem mergeSort_eq : ∀ A (le : A → A → bool) ls,
  mergeSort le ls = if le_lt_dec 2 (length ls)
    then let lss := split ls in
      merge le (mergeSort le (fst lss)) (mergeSort le (snd lss))
    else ls.
  intros; apply (Fix_eq (@lengthOrder_wf A) (fun _ ⇒ list A));
    intros.
```

The library theorem Fix_eq imposes one more subgoal. We must prove that the function body is unable to distinguish between "self" arguments that map equal inputs to equal outputs. One might think this should be true of any Gallina code, but in fact this general *function extensionality* property is neither provable nor disprovable within Coq. The type of Fix_eq makes clear what we must show manually:

```
  Check Fix_eq.
```

```
Fix_eq
     : ∀ (A : Type) (R : A → A → Prop) (Rwf : well_founded R)
         (P : A → Type)
         (F : ∀ x : A, (∀ y : A, R y x → P y) → P x),
       (∀ (x : A) (f g : ∀ y : A, R y x → P y),
        (∀ (y : A) (p : R y x), f y p = g y p) → F x f = F x g) →
       ∀ x : A,
       Fix Rwf P F x
       = F x (fun (y : A) (_ : R y x) ⇒ Fix Rwf P F y)
```

Most such obligations are dischargeable with straightforward proof automation, and this example is no exception.

```
  match goal with
    | [ ⊢ context[match ?E with left _ ⇒ _ | right _ ⇒ _ end] ] ⇒
      destruct E
  end; simpl; f_equal; auto.
Qed.
```

As a final test of the definition's suitability, we can extract to OCaml.

```
Extraction mergeSort.
```

```
let rec mergeSort le x =
  match le_lt_dec (S (S O)) (length x) with
  | Left ->
    let lss = split x in
    merge le (mergeSort le (fst lss)) (mergeSort le (snd lss))
  | Right -> x
```

We get almost the same definition we would have written manually in OCaml. Readers could use the commands we saw in the previous chapter to clean up some remaining differences from idiomatic OCaml.

One more piece of the full picture is missing. To prove correctness of mergeSort, we would need more than a way of unfolding its definition. We also need an appropriate induction principle matched to the well-founded relation. Such a principle is available in the standard library.

```
Check well_founded_induction.
```

```
well_founded_induction
      : ∀ (A : Type) (R : A → A → Prop),
        well_founded R →
        ∀ P : A → Set,
        (∀ x : A, (∀ y : A, R y x → P y) → P x) →
        ∀ a : A, P a
```

Some recent Coq features provide more convenient syntax for defining recursive functions. Interested readers can consult the Coq manual about the commands `Function` and `Program Fixpoint`.

7.2 A Nontermination Monad Inspired by Domain Theory

The key insights of domain theory [49] inspire the next approach to modeling nontermination. Domain theory is based on *information* *orders* that relate values representing computation results according to how much information these values convey. For instance, a simple domain might include values "the program does not terminate" and "the program terminates with the answer 5." The former is considered to be an *approximation* of the latter, whereas the latter is *not* an approximation of "the program terminates with the answer 6." The details of domain theory are not important in what follows; we merely borrow the notion of an approximation ordering on computation results.

Consider this definition of a type of computations.

```
Section computation.
  Variable A : Type.
```

The type A describes the result a computation will yield if it terminates.

We give a rich dependent type to computations themselves.

```
  Definition computation :=
    {f : nat → option A
      | ∀ (n : nat) (v : A),
        f n = Some v
        → ∀ (n' : nat), n' ≥ n
          → f n' = Some v}.
```

A computation is fundamentally a function f from an *approximation level* n to an optional result. Intuitively, higher n values enable termination in more cases than lower values. A call to f may return None to indicate that n was not high enough to run the computation to completion; higher n values may yield Some. Further, the proof obligation within the subset type asserts that f is *monotone* in an appropriate sense: when some n is sufficient to produce termination, so are all higher n values, and they all yield the same program result v.

It is easy to define a relation characterizing when a computation runs to a particular result at a particular approximation level.

```
  Definition runTo (m : computation) (n : nat) (v : A) :=
    proj1_sig m n = Some v.
```

On top of runTo, we also define run, which is the most abstract notion of when a computation runs to a value.

```
  Definition run (m : computation) (v : A) :=
    ∃ n, runTo m n v.
End computation.
```

The book source code contains at this point some tactics, lemma proofs, and hint commands to be used in proving facts about computations.

As a simple first example of a computation, we can define Bottom, which corresponds to an infinite loop. For any approximation level, it fails to terminate (returns None). Note the use of `abstract` to create a new opaque lemma for the proof found by the *run* tactic. In contrast to the previous section, opaque proofs are fine here, since the proof components of computations do not influence evaluation behavior. It is generally preferable to make proofs opaque when possible, as this enforces a kind of modularity in the code to follow, preventing it from depending on any details of the proof.

```
Section Bottom.
  Variable A : Type.

  Definition Bottom : computation A.
    exists (fun _ : nat ⇒ @None A); abstract run.
  Defined.

  Theorem run_Bottom : ∀ v, ¬run Bottom v.
    run.
  Qed.
End Bottom.
```

A slightly more complicated example is Return, which gives the same terminating answer at every approximation level.

```
Section Return.
  Variable A : Type.
  Variable v : A.

  Definition Return : computation A.
    exists (fun _ : nat ⇒ Some v); abstract run.
  Defined.

  Theorem run_Return : run Return v.
    run.
  Qed.
End Return.
```

The name Return was meant to suggest the standard operations of monads [44]. The other standard operation is `Bind`, which lets us run one computation and, if it terminates, pass its result off to another computation. We implement bind using the notation `let` $(x, y) :=$ *e1* `in` *e2*, for pulling apart the value *e1*, which may be thought of as a pair. The second component of a computation is a proof, which we do not need to mention directly in the definition of `Bind`.

```
Section Bind.
  Variables A B : Type.
  Variable m1 : computation A.
  Variable m2 : A → computation B.

  Definition Bind : computation B.
    exists (fun n ⇒
      let (f1, _) := m1 in
      match f1 n with
        | None ⇒ None
        | Some v ⇒
          let (f2, _) := m2 v in
            f2 n
      end); abstract run.
  Defined.

  Theorem run_Bind : ∀ (v1 : A) (v2 : B),
    run m1 v1
    → run (m2 v1) v2
    → run Bind v2.
    run; match goal with
             | [ x : nat, y : nat ⊢ _ ] ⇒ exists (max x y)
           end; run.
  Qed.
End Bind.
```

A simple notation lets us write Bind calls the way they appear in Haskell.

```
Notation "x <- m1 ; m2" :=
  (Bind m1 (fun x ⇒ m2)) (right associativity, at level 70).
```

We can verify that we have indeed defined a monad, by proving the standard monad laws. Part of the exercise is choosing an appropriate notion of equality between computations. We use "equality at all approximation levels."

```
Definition meq A (m1 m2 : computation A) :=
  ∀ n, proj1_sig m1 n = proj1_sig m2 n.

Theorem left_identity : ∀ A B (a : A) (f : A → computation B),
  meq (Bind (Return a) f) (f a).
  run.
Qed.

Theorem right_identity : ∀ A (m : computation A),
  meq (Bind m (@Return _)) m.
  run.
Qed.
```

```
Theorem associativity : ∀ A B C (m : computation A)
  (f : A → computation B) (g : B → computation C),
  meq (Bind (Bind m f) g) (Bind m (fun x ⇒ Bind (f x) g)).
  run.
Qed.
```

Now we come to the piece most directly inspired by domain theory. We want to support general recursive function definitions, but domain theory says that not all definitions are reasonable; some fail to be *continuous* and thus represent unrealizable computations. To formalize an analogous notion of continuity for the nontermination monad, we write down the approximation relation on computation results that we have had in mind all along.

```
Section lattice.
  Variable A : Type.

  Definition leq (x y : option A) :=
    ∀ v, x = Some v → y = Some v.
End lattice.
```

We now have the tools we need to define a new `Fix` combinator that, unlike the one in Section 7.1, does not require a termination proof, and in fact admits recursive definition of functions that fail to terminate on some or all inputs.

```
Section Fix.
```

First, we have the function domain and range types.

```
  Variables A B : Type.
```

Next comes the function body, which is written as though it can be parameterized over itself, for recursive calls.

```
  Variable f : (A → computation B) → (A → computation B).
```

Finally, we impose an obligation to prove that the body f is continuous. That is, when f terminates according to one recursive version of itself, it also terminates with the same result at the same approximation level when passed a recursive version that refines the original, according to leq.

```
  Hypothesis f_continuous : ∀ n v v1 x,
    runTo (f v1 x) n v
    → ∀ (v2 : A → computation B),
      (∀ x, leq (proj1_sig (v1 x) n) (proj1_sig (v2 x) n))
      → runTo (f v2 x) n v.
```

The computational part of the Fix combinator is easy to define. At approximation level 0, we diverge; at higher levels, we run the body with a functional argument drawn from the next lower level.

```
Fixpoint Fix' (n : nat) (x : A) : computation B :=
  match n with
    | O ⇒ Bottom _
    | S n' ⇒ f (Fix' n') x
  end.
```

Now it is straightforward to package Fix' as a computation combinator Fix.

```
Hint Extern 1 (_ ≥ _) ⇒ omega.
Hint Unfold leq.

Lemma Fix'_ok : ∀ steps n x v, proj1_sig (Fix' n x) steps = Some v
  → ∀ n', n' ≥ n
    → proj1_sig (Fix' n' x) steps = Some v.
  unfold runTo in *; induction n; crush;
    match goal with
      | [ H : _ ≥ _ ⊢ _ ] ⇒ inversion H; crush; eauto
    end.
Qed.

Hint Resolve Fix'_ok.

Hint Extern 1 (proj1_sig _ _ = _) ⇒ simpl;
  match goal with
    | [ ⊢ proj1_sig ?E _ = _ ] ⇒ eapply (proj2_sig E)
  end.

Definition Fix : A → computation B.
  intro x; exists (fun n ⇒ proj1_sig (Fix' n x) n); abstract run.
Defined.
```

Finally, we can prove that Fix obeys the expected computation rule.

```
Theorem run_Fix : ∀ x v,
  run (f Fix x) v
  → run (Fix x) v.
  run; match goal with
          | [ n : nat ⊢ _ ] ⇒ exists (S n); eauto
        end.
Qed.
End Fix.
```

After all that work, it is now fairly painless to define a version of mergeSort that requires no proof of termination. We appeal to a program-specific tactic (its definition is in the book source code).

```
Definition mergeSort' : ∀ A, (A → A → bool) → list A
  → computation (list A).
  refine (fun A le ⇒ Fix
    (fun (mergeSort : list A → computation (list A))
      (ls : list A) ⇒
      if le_lt_dec 2 (length ls)
        then let lss := split ls in
          ls1 ← mergeSort (fst lss);
          ls2 ← mergeSort (snd lss);
          Return (merge le ls1 ls2)
        else Return ls) _); abstract mergeSort'.
Defined.
```

Running mergeSort' on concrete inputs is as easy as choosing a sufficiently high approximation level and letting Coq's computation rules do the rest. Contrast this with the proof work that goes into deriving an evaluation fact for a deeply embedded language, with one explicit proof rule application per execution step.

```
Lemma test_mergeSort' : run (mergeSort' leb
  (1 :: 2 :: 36 :: 8 :: 19 :: nil))
  (1 :: 2 :: 8 :: 19 :: 36 :: nil).
  exists 4; reflexivity.
Qed.
```

There is another benefit of the new Fix compared with the one in Section 7.1: we can now write recursive functions that sometimes fail to terminate without losing easy reasoning principles for the terminating cases. Consider this simple example (which appeals to another tactic whose definition we elide here).

```
Definition looper : bool → computation unit.
  refine (Fix (fun looper (b : bool) ⇒
    if b then Return tt else looper b) _); abstract looper.
Defined.
```

```
Lemma test_looper : run (looper true) tt.
  exists 1; reflexivity.
Qed.
```

As before, proving outputs for specific inputs is as easy as demonstrating a high enough approximation level.

There are other theorems that are important to prove about combinators like Return, Bind, and Fix. In general, for a computation c, we sometimes have a hypothesis proving run c v for some v, and we want to perform inversion to deduce what v must be. Each combinator should ideally have a theorem of that kind, for c built directly from that combinator. Such theorems are omitted here, but they are not hard to prove. In general, the approach inspired by domain theory avoids the type-theoretic problems in approaches that try to mix normal Coq computation with explicit syntax types.

The next section of this chapter demonstrates two alternative approaches of that sort. The final section reviews the pros and cons of the different choices, concluding that none is better than any other for all situations.

7.3 Co-inductive Nontermination Monads

There are two key downsides to both of the previous approaches: both require unusual syntax based on explicit calls to fixpoint combinators, and both generate immediate proof obligations about the bodies of recursive definitions. Chapter 5 showed how co-inductive types support recursive definitions that exhibit certain well-behaved varieties of nontermination. We can leverage that co-induction support for encoding of general recursive definitions, by adding layers of co-inductive syntax. In effect, we mix elements of shallow and deep embeddings.

Our first example of this kind, proposed by Capretta [4], defines a type of thunks; that is, computations that may be forced to yield results if they terminate.

```
CoInductive thunk (A : Type) : Type :=
| Answer : A → thunk A
| Think : thunk A → thunk A.
```

A computation is either an immediate Answer or another computation wrapped inside Think. Since **thunk** is co-inductive, every **thunk** type is inhabited by an infinite nesting of Thinks, standing for nontermination. Terminating results are Answer wrapped inside some finite number of Thinks.

Why bother to write such an odd definition? The definition of **thunk** is motivated by the ability it gives to define a bind operation similar to the one defined in Section 7.2.

```
CoFixpoint TBind A B (m1 : thunk A) (m2 : A → thunk B)
  : thunk B :=
  match m1 with
    | Answer x ⇒ m2 x
    | Think m1' ⇒ Think (TBind m1' m2)
  end.
```

Note that the definition would violate the co-recursion guardedness restriction if we left out the seemingly superfluous Think on the right side of the second `match` branch.

We can prove that Answer and TBind form a monad for **thunk** (the proof is in the book source code). As usual for this sort of proof, a key element is choosing an appropriate notion of equality for **thunk**s.

In the following proofs, we need a function similar to one in Chapter 5, to pull apart and reassemble a **thunk** in a way that provokes reduction of co-recursive calls.

```
Definition frob A (m : thunk A) : thunk A :=
  match m with
    | Answer x ⇒ Answer x
    | Think m' ⇒ Think m'
  end.

Theorem frob_eq : ∀ A (m : thunk A), frob m = m.
  destruct m; reflexivity.
Qed.
```

As a simple example, here is how we might define a tail-recursive factorial function:

```
CoFixpoint fact (n acc : nat) : thunk nat :=
  match n with
    | O ⇒ Answer acc
    | S n' ⇒ Think (fact n' (S n' × acc))
  end.
```

To test the definition, we need an evaluation relation that characterizes results of evaluating **thunk**s.

```
Inductive eval A : thunk A → A → Prop :=
| EvalAnswer : ∀ x, eval (Answer x) x
| EvalThink : ∀ m x, eval m x → eval (Think m) x.

Hint Rewrite frob_eq.

Lemma eval_frob : ∀ A (c : thunk A) x,
  eval (frob c) x
  → eval c x.
  crush.
```

```
Qed.

Theorem eval_fact : eval (fact 5 1) 120.
  repeat (apply eval_frob; simpl; constructor).
Qed.
```

We need to apply constructors of `eval` explicitly, but the process is easy to automate completely for concrete input programs.

Now consider another very similar definition, this time of a Fibonacci number function.

```
Notation "x <- m1 ; m2" :=
  (TBind m1 (fun x ⇒ m2)) (right associativity, at level 70).

CoFixpoint fib (n : nat) : thunk nat :=
  match n with
    | 0 ⇒ Answer 1
    | 1 ⇒ Answer 1
    | _ ⇒ n1 ← fib (pred n);
      n2 ← fib (pred (pred n));
      Answer (n1 + n2)
  end.
```

Coq complains that the guardedness condition is violated. The two recursive calls are immediate arguments to TBind, but TBind is not a constructor of **thunk**. Rather, it is a defined function. This example shows a very serious limitation of **thunk** for traditional functional programming: it is not, in general, possible to make recursive calls and then make further recursive calls, depending on the first call's result. The fact example succeeded because it was already tail-recursive, meaning no further computation is needed after a recursive call.

I know no easy fix for this problem of **thunk**, but we can define a different co-inductive monad that avoids the problem, based on a proposal by Megacz [24]. We ran into trouble because TBind was not a constructor of **thunk**, so let us define a new type family where "bind" is a constructor.

```
CoInductive comp (A : Type) : Type :=
| Ret : A → comp A
| Bnd : ∀ B, comp B → (B → comp A) → comp A.
```

This example shows off Coq's support for *recursively nonuniform parameters*, as in the case of the parameter A, where each constructor's type ends in **comp** A but there is a recursive use of **comp** with a different parameter B. Beside that technical wrinkle, we see the simplest possible definition of a monad, via a type whose two constructors are precisely the monad operators.

It is easy to define the semantics of terminating **comp** computations.

```
Inductive exec A : comp A → A → Prop :=
| ExecRet : ∀ x, exec (Ret x) x
| ExecBnd : ∀ B (c : comp B) (f : B → comp A) x1 x2,
  exec (A := B) c x1
  → exec (f x1) x2
  → exec (Bnd c f) x2.
```

We can also prove that Ret and Bnd form a monad according to a notion of **comp** equality based on **exec** (the proof is in the book source code).

Not only can we define the Fibonacci function with the new monad but even the running example of merge sort becomes definable. By shadowing the previous notation for “bind,” we can write almost exactly the same code as in the previous mergeSort' definition, but with less syntactic clutter.

```
Notation "x <- m1 ; m2" := (Bnd m1 (fun x ⇒ m2)).

CoFixpoint mergeSort'' A (le : A → A → bool) (ls : list A)
  : comp (list A) :=
  if le_lt_dec 2 (length ls)
    then let lss := split ls in
      ls1 ← mergeSort'' le (fst lss);
      ls2 ← mergeSort'' le (snd lss);
      Ret (merge le ls1 ls2)
    else Ret ls.
```

To execute this function, we go through the usual exercise of writing a function to catalyze evaluation of co-recursive calls.

```
Definition frob' A (c : comp A) :=
  match c with
    | Ret x ⇒ Ret x
    | Bnd _ c' f ⇒ Bnd c' f
  end.

Lemma exec_frob : ∀ A (c : comp A) x,
  exec (frob' c) x
  → exec c x.
  destruct c; crush.
Qed.
```

Now the same sort of proof script that we applied for testing **thunk** will get the job done.

```
Lemma test_mergeSort'' : exec (mergeSort'' leb
  (1 :: 2 :: 36 :: 8 :: 19 :: nil))
  (1 :: 2 :: 8 :: 19 :: 36 :: nil).
  repeat (apply exec_frob; simpl; econstructor).
Qed.
```

Have we finally reached the ideal solution for encoding general recursive definitions, with minimal hassle in syntax and proof obligations? Unfortunately, we have not, as **comp** has a serious expressivity weakness. Consider the following definition of a curried addition function.

```
Definition curriedAdd (n : nat) := Ret (fun m : nat ⇒ Ret (n + m)).
```

This definition works fine, but we run into trouble when we try to apply it in a trivial way.

```
Definition testCurriedAdd := Bnd (curriedAdd 2) (fun f ⇒ f 3).
```

```
Error: Universe inconsistency.
```

The problem has to do with rules for inductive definitions (see Chapter 12). Briefly, recall that the type of the constructor Bnd quantifies over a type B. To make testCurriedAdd work, we would need to instantiate B as **nat** → **comp nat**. However, Coq enforces a *predicativity restriction* that (roughly) no quantifier in an inductive or co-inductive type's definition may ever be instantiated with a term that contains the type being defined. Chapter 12 presents the exact mechanism by which this restriction is enforced, but for now our conclusion is that **comp** is fatally flawed as a way of encoding higher-order functional programs that use general recursion.

7.4 Comparing the Alternatives

We have seen four different approaches to encoding general recursive definitions in Coq. Among them there is no clear champion that dominates the others in every important way. Instead, we close the chapter by comparing the techniques along a number of dimensions. Every technique allows recursive definitions with termination arguments that go beyond Coq's built-in termination checking, so we must turn to subtler points to highlight differences.

One useful property is automatic integration with normal Coq programming. That is, we would like the type of a function to be the same, whether or not that function is defined using an interesting recursion pattern. Only the first of the four techniques, well-founded recursion, meets this criterion. It is also the only one of the four to meet the related criterion that evaluation of function calls can take place entirely inside Coq's built-in computation machinery. The monad inspired by domain theory occupies some middle ground in this dimension, since generally standard computation is enough to evaluate a term once a high enough approximation level is provided.

Another useful property is that a function and its termination argument may be developed separately. We may even want to define functions that fail to terminate on some or all inputs. The well-founded recursion technique does not have this property, but the other three do.

One minor plus is the ability to write recursive definitions in natural syntax rather than with calls to higher-order combinators. This downside of the first two techniques is actually easy to get around using Coq's notation mechanism, though I leave the details as an exercise for the reader. (For this and other details of notations, see Chapter 12 of the Coq 8.4 manual.)

The first two techniques impose proof obligations that are more basic than termination arguments, where well-founded recursion requires a proof of extensionality and domain-theoretic recursion requires a proof of continuity. A function may not be defined, and thus may not be computed with, until these obligations are proved. The co-inductive techniques avoid this problem, as recursive definitions may be made without any proof obligations.

We can also consider support for common idioms in functional programming. For instance, the **thunk** monad effectively only supports recursion that is tail recursion, whereas the others allow arbitrary recursion schemes.

On the other hand, the **comp** monad does not support the effective mixing of higher-order functions and general recursion, whereas all the other techniques do. For instance, we can finish the failed curriedAdd example in the domain-theoretic monad.

```
Definition curriedAdd' (n : nat) :=
  Return (fun m : nat ⇒ Return (n + m)).

Definition testCurriedAdd := Bind (curriedAdd' 2) (fun f ⇒ f 3).
```

The same techniques also apply to more interesting higher-order functions like list map, and as in all four techniques, we can mix primitive and general recursion, preferring the former when possible to avoid proof obligations.

```
Fixpoint map A B (f : A → computation B) (ls : list A)
  : computation (list B) :=
  match ls with
    | nil ⇒ Return nil
    | x :: ls' ⇒ Bind (f x) (fun x' ⇒
      Bind (map f ls') (fun ls'' ⇒
        Return (x' :: ls'')))
  end.

Theorem test_map : run (map (fun x ⇒ Return (S x))
  (1 :: 2 :: 3 :: nil))
  (2 :: 3 :: 4 :: nil).
  exists 1; reflexivity.
Qed.
```

One further disadvantage of **comp** is that we cannot prove an inversion lemma for executions of `Bind` without appealing to an axiom (see Chapter 12). The other three techniques allow proof of all the important theorems within the normal logic of Coq.

Perhaps one theme of this comparison is that one must trade off between, on one hand, functional programming expressiveness and compatibility with normal Coq types and computation; and, on the other hand, the level of proof obligations one is willing to handle at function definition time.