5 Infinite Data and Proofs

In lazy functional programming languages like Haskell, infinite data structures are everywhere [16]. Infinite lists and more exotic datatypes provide convenient abstractions for communication between parts of a program. Achieving similar convenience without infinite lazy structures would, in many cases, require acrobatic inversions of control flow.

Laziness is easy to implement in Haskell, where all the definitions in a program may be thought of as mutually recursive. In such an unconstrained setting, it is easy to implement an infinite loop when we really meant to build an infinite list, where any finite prefix of the list should be forceable in finite time. Haskell programmers learn how to avoid such slipups. In Coq, such a laissez-faire policy is not good enough.

Chapter 4 discussed the Curry-Howard isomorphism, where proofs are identified with functional programs. In such a setting, infinite loops, intended or otherwise, are disastrous. If Coq allowed the full breadth of definitions that Haskell does, we could code an infinite loop and use it to prove any proposition vacuously. That is, the addition of general recursion would make the Calculus of Inductive Constructions (CIC) *inconsistent*. For an arbitrary proposition P, we could write

```
Fixpoint bad (u : unit) : P := bad u.
```

This would leave us with `bad tt` as a proof of P.

There are also algorithmic considerations that make universal termination very desirable. We have seen how tactics like `reflexivity` compare terms up to equivalence under computational rules. Calls to recursive, pattern-matching functions are simplified automatically, with no need for explicit proof steps. It would be very hard to hold onto that kind of benefit if it became possible to write nonterminating programs; we would run into the halting problem.

One solution is to use types to contain the possibility of nontermination. For instance, we can create a nontermination monad, inside which

we must write all general-recursive programs; several such approaches are surveyed in Chapter 7. This is a heavyweight solution, so we would like to avoid it whenever possible.

Luckily, Coq has special support for a class of lazy data structures that happens to contain most examples found in Haskell. That mechanism, *co-inductive types*, is the subject of this chapter.

5.1 Computing with Infinite Data

Let us begin with the most basic type of infinite data, *streams*, or lazy lists.

```
Section stream.
  Variable A : Type.

  CoInductive stream : Type :=
  | Cons : A → stream → stream.
End stream.
```

The definition is surprisingly simple. Starting from the definition of **list**, we just need to change the keyword `Inductive` to `CoInductive`. We could have left a Nil constructor in the definition, but we will leave it out to force all streams to be infinite.

How do we write a stream constant? The simple application of constructors is not good enough, since we could only denote finite objects that way. Rather, whereas recursive definitions were necessary to *use* values of recursive inductive types effectively, here we find that we need *co-recursive definitions* to *build* values of co-inductive types effectively.

We can define a stream consisting only of zeroes.

```
CoFixpoint zeroes : stream nat := Cons 0 zeroes.
```

We can also define a stream that alternates between true and false.

```
CoFixpoint trues_falses : stream bool := Cons true falses_trues
with falses_trues : stream bool := Cons false trues_falses.
```

Co-inductive values can be used as arguments to recursive functions, and we can use that fact to write a function to take a finite approximation of a stream.

```
Fixpoint approx A (s : stream A) (n : nat) : list A :=
  match n with
    | O ⇒ nil
    | S n' ⇒
      match s with
```

```
      | Cons h t ⇒ h :: approx t n'
    end
  end.
```

```
Eval simpl in approx zeroes 10.
```

```
    = 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: nil
    : list nat
```

```
Eval simpl in approx trues_falses 10.
```

```
    = true
      :: false
          :: true
              :: false
                  :: true :: false :: true :: false :: true :: false :: nil
    : list bool
```

So far, it looks like co-inductive types might allow us to import all the Haskeller's usual tricks. However, some important restrictions are dual to the restrictions on the use of inductive types. Fixpoints *consume* values of inductive types, with restrictions on which *arguments* may be passed in recursive calls. Dually, co-fixpoints *produce* values of co-inductive types, with restrictions on what may be done with the *results* of co-recursive calls.

The restriction for co-inductive types shows up as the *guardedness condition*. First, consider this stream definition, which would be legal in Haskell:

```
CoFixpoint looper : stream nat := looper.
```

```
Error:
Recursive definition of looper is ill-formed.
In environment
looper : stream nat

unguarded recursive call in "looper"
```

The rule we have run afoul of here is that *every co-recursive call must be guarded by a constructor*; that is, every co-recursive call must be a direct argument to a constructor of the co-inductive type we are generating. It is a good thing that this rule is enforced. If the definition of looper were accepted, the approx function would run forever when passed looper, and we would have fallen into inconsistency.

Some familiar functions are easy to write in co-recursive fashion.

```
Section map.
  Variables A B : Type.
  Variable f : A → B.

  CoFixpoint map (s : stream A) : stream B :=
    match s with
      | Cons h t ⇒ Cons (f h) (map t)
    end.
End map.
```

This code is a literal copy of that for the list map function, with the nil case removed and `Fixpoint` changed to `CoFixpoint`. Many other standard functions on lazy data structures can be implemented just as easily. Some, like filter, cannot be implemented. Since the predicate passed to filter may reject every element of the stream, we cannot satisfy the guardedness condition.

The implications of the condition can be subtle. To illustrate, we start off with another co-recursive function definition that *is* legal. The function interleave takes two streams and produces a new stream that alternates between their elements.

```
Section interleave.
  Variable A : Type.

  CoFixpoint interleave (s1 s2 : stream A) : stream A :=
    match s1, s2 with
      | Cons h1 t1, Cons h2 t2 ⇒ Cons h1 (Cons h2 (interleave t1 t2))
    end.
End interleave.
```

Now suppose we want to write a weird stuttering version of map that repeats elements in a particular way, based on interleaving.

```
Section map'.
  Variables A B : Type.
  Variable f : A → B.

  CoFixpoint map' (s : stream A) : stream B :=
    match s with
      | Cons h t ⇒
      interleave (Cons (f h) (map' t)) (Cons (f h) (map' t))
    end.
```

We get another error message about an unguarded recursive call.

```
End map'.
```

What went wrong here? Imagine that instead of interleave we had called some other, less well-behaved function on streams. Here is one simpler example demonstrating the pitfall. We start by defining a standard function for taking the tail of a stream. Since streams are infinite, this operation is total.

```
Definition tl A (s : stream A) : stream A :=
  match s with
    | Cons _ s' ⇒ s'
  end.
```

Coq rejects the following definition that uses tl.

```
CoFixpoint bad : stream nat := tl (Cons 0 bad).
```

Imagine that Coq had accepted the definition, and consider how we might evaluate approx bad 1. We would be trying to calculate the first element in the stream bad. However, the definition of bad begs the question: unfolding the definition of tl, we see that we essentially say "define bad to equal itself." Of course such an equation admits no single well-defined solution, which does not fit well with the determinism of Gallina reduction.

Coq's complete rule for co-recursive definitions includes not just the basic guardedness condition but also a requirement about where co-recursive calls may occur. In particular, a co-recursive call must be a direct argument to a constructor, *nested only inside of other constructor calls or* `fun` *or* `match` *expressions*. In the definition of bad, we erroneously nested the co-recursive call inside a call to tl, and we nested inside a call to interleave in the definition of map'.

Coq helps the user by performing the guardedness check after using computation to simplify terms. For instance, any co-recursive function definition can be expanded by inserting extra calls to an identity function, and this change preserves guardedness. However, in other cases computational simplification can reveal why definitions are dangerous. Consider what happens when we inline the definition of tl in bad.

```
CoFixpoint bad : stream nat := bad.
```

This is the same looping definition we rejected earlier. A similar inlining process reveals a different view on the failed definition of map'.

```
CoFixpoint map' (s : stream A) : stream B :=
  match s with
    | Cons h t ⇒ Cons (f h) (Cons (f h) (interleave (map' t) (map' t)))
  end.
```

Clearly in this case the map' calls are not immediate arguments to constructors, so we violate the guardedness condition.

A more interesting question is why that condition is the right one. We can make an intuitive argument that the original map' definition is perfectly reasonable and denotes a well-understood transformation on streams, such that every output would behave properly with approx. The guardedness condition is an example of a syntactic check for *productivity* of co-recursive definitions. A productive definition can be thought of as one whose outputs can be forced in finite time to any finite approximation level, as with approx. If we replaced the guardedness condition with more involved checks, we might be able to detect and allow a broader range of productive definitions. However, mistakes in these checks could cause inconsistency, and programmers would need to understand the new, more complex checks. Coq's design strikes a balance between consistency and simplicity with its choice of guard condition, though we can imagine other worthwhile balances being struck.

5.2 Infinite Proofs

Let us say we want to give two different definitions of a stream of all ones and then prove that they are equivalent.

```
CoFixpoint ones : stream nat := Cons 1 ones.
Definition ones' := map S zeroes.
```

The obvious statement of the equality is this:

```
Theorem ones_eq : ones = ones'.
```

However, with the initial subgoal, it is not at all clear how this theorem can be proved. In fact, it is unprovable. The **eq** predicate is fundamentally limited to equalities that can be demonstrated by finite, syntactic arguments. To prove this equivalence, we need to introduce a new relation.

```
Abort.
```

Co-inductive datatypes make sense by analogy from Haskell. What we need now is a *co-inductive proposition*. That is, we want to define a proposition whose proofs may be infinite, subject to the guardedness condition. The idea of infinite proofs does not show up in usual mathematics, but it can be very useful for reasoning about infinite data structures. Besides examples from Haskell, infinite data and proofs will

also turn out to be useful for modeling inherently infinite mathematical objects, like program executions.

We are ready for our first co-inductive predicate.

```
Section stream_eq.
  Variable A : Type.

  CoInductive stream_eq : stream A → stream A → Prop :=
  | Stream_eq : ∀ h t1 t2,
    stream_eq t1 t2
    → stream_eq (Cons h t1) (Cons h t2).
End stream_eq.
```

We say that two streams are equal if and only if they have the same heads and their tails are equal. We use the normal finite-syntactic equality for the heads, and we refer to the new equality recursively for the tails.

We can try restating the theorem with **stream_eq**.

```
Theorem ones_eq : stream_eq ones ones'.
```

Coq does not support tactical co-inductive proofs as well as it supports tactical inductive proofs. The usual starting point is the `cofix` tactic, which asks to structure this proof as a co-fixpoint.

```
  cofix.

  ones_eq : stream_eq ones ones'
  ============================
   stream_eq ones ones'
```

It looks like this proof might be easier than we expected.

```
  assumption.

Proof completed.
```

Unfortunately, we are due for some disappointment.

```
Qed.

Error:
Recursive definition of ones_eq is ill-formed.

In environment
ones_eq : stream_eq ones ones'

unguarded recursive call in "ones_eq"
```

Via the Curry-Howard correspondence, the same guardedness condition applies to co-inductive proofs as to co-inductive data structures. If it did not, the same proof structure could be used to prove any co-inductive theorem vacuously, by direct appeal to itself.

Thinking about how Coq would generate a proof term from the previous proof script, we see that the problem is that we are violating the guardedness condition. During proofs, Coq can help us check whether we have yet gone wrong in this way. We can run the command `Guarded` in any context to see if it is possible to finish the proof in a way that will yield a properly guarded proof term.

```
Guarded.
```

Running `Guarded` here gives the same error message that we got when we tried to run `Qed`. In larger proofs, `Guarded` can be helpful in detecting problems *before* we think we are ready to run `Qed`.

We need to start the co-induction by applying **stream_eq**'s constructor. To do that, we need to know that both arguments to the predicate are Conses. Informally, this is trivial, but `simpl` is not able to help.

```
  Undo.
  simpl.


  ones_eq : stream_eq ones ones'
  ============================
   stream_eq ones ones'
```

It turns out that we are best served by proving an auxiliary lemma.

```
Abort.
```

First, we need to define a function that seems pointless at first glance.

```
Definition frob A (s : stream A) : stream A :=
  match s with
    | Cons h t ⇒ Cons h t
  end.
```

Next, we need to prove a theorem that seems equally pointless.

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

But this theorem turns out to be just what we needed.

```
Theorem ones_eq : stream_eq ones ones'.
  cofix.
```

We can use the theorem to rewrite the two streams.

```
rewrite (frob_eq ones).
rewrite (frob_eq ones').
```

```
ones_eq : stream_eq ones ones'
============================
 stream_eq (frob ones) (frob ones')
```

Now `simpl` is able to reduce the streams.

```
simpl.
```

```
ones_eq : stream_eq ones ones'
============================
 stream_eq (Cons 1 ones)
   (Cons 1
      ((cofix map (s : stream nat) : stream nat :=
          match s with
          | Cons h t ⇒ Cons (S h) (map t)
          end) zeroes))
```

Note the `cofix` notation for anonymous co-recursion, which is analogous to the `fix` notation we have already seen for recursion. Since we have exposed the Cons structure of each stream, we can apply the constructor of **stream_eq**.

```
constructor.
```

```
ones_eq : stream_eq ones ones'
============================
 stream_eq ones
   ((cofix map (s : stream nat) : stream nat :=
       match s with
       | Cons h t ⇒ Cons (S h) (map t)
       end) zeroes)
```

Now, modulo unfolding of the definition of map, we have matched the assumption.

```
  assumption.
Qed.
```

Why did this work-around help? The answer has to do with the constraints placed on Coq's evaluation rules by the need for termination. The `cofix`-related restriction that foiled the first attempt at using `simpl` is dual to a restriction for `fix`. In particular, an application of an anonymous `fix` only reduces when the top-level structure of the recursive argument is known. Otherwise, we would be unfolding the recursive definition ad infinitum.

Fixpoints only reduce when enough is known about the *definitions* of their arguments. Dually, co-fixpoints only reduce when enough is known about *how their results will be used*. In particular, a `cofix` is only expanded when it is the discriminee of a `match`. Rewriting with the new lemma wrapped new `match`es around the two `cofix`es, triggering reduction.

If `cofix`es reduced haphazardly, it would be easy to run into infinite loops in evaluation, since we are, after all, building infinite objects.

One common source of difficulty with co-inductive proofs is bad interaction with standard Coq automation machinery. If we try to prove `ones_eq` with automation, as with previous inductive proofs, we get an invalid proof.

```
Theorem ones_eq' : stream_eq ones ones'.
  cofix; crush.
  Guarded.
Abort.
```

The standard `auto` machinery sees that the goal matches an assumption and so applies that assumption, even though this violates guardedness. A correct proof strategy for a theorem like this usually starts by `destruct`ing some parameter and running a custom tactic to figure out the first proof rule to apply for each case. Alternatively, there are tricks for "hiding" the co-inductive hypothesis.

Induction seems to have dual versions of the same pitfalls inherent in it, which can be avoided by encapsulating safe Curry-Howard recursion schemes inside named induction principles. We can usually do the same with *co-induction principles*. Let us do that here, so that we can arrive at an `induction` x; *crush*-style proof for `ones_eq'`.

An induction principle is parameterized over a predicate characterizing what we mean to prove, *as a function of the inductive fact that we already know*. Dually, a co-induction principle ought to be parameterized over a predicate characterizing what we need to assume, *as a function of the arguments to the co-inductive predicate that we are trying to prove*.

To state a useful principle for **stream_eq**, it will be useful first to define the stream head function.

```
Definition hd A (s : stream A) : A :=
  match s with
    | Cons x _ ⇒ x
  end.
```

Now we enter a section for the co-induction principle, based on Park's principle as introduced in a tutorial by Giménez [12].

```
Section stream_eq_coind.
  Variable A : Type.
  Variable R : stream A → stream A → Prop.
```

This relation generalizes the theorem we want to prove, defining a set of pairs of streams that we must eventually prove contains the particular pair we care about.

```
  Hypothesis Cons_case_hd : ∀ s1 s2, R s1 s2 → hd s1 = hd s2.
  Hypothesis Cons_case_tl : ∀ s1 s2, R s1 s2 → R (tl s1) (tl s2).
```

Two hypotheses characterize what makes a good choice of R: it enforces equality of stream heads, and it is hereditary in the sense that an R stream pair passes on R-ness to its tails. An established technical term for such a relation is *bisimulation*.

Now it is straightforward to prove the principle, which says that any stream pair in R is equal. Readers may wish to step through the proof script to see what is going on.

```
  Theorem stream_eq_coind : ∀ s1 s2, R s1 s2 → stream_eq s1 s2.
    cofix; destruct s1; destruct s2; intro.
    generalize (Cons_case_hd H); intro Heq;
      simpl in Heq; rewrite Heq.
    constructor.
    apply stream_eq_coind.
    apply (Cons_case_tl H).
  Qed.
End stream_eq_coind.
```

To see why this proof is guarded, we can print it and verify that the one co-recursive call is an immediate argument to a constructor.

```
Print stream_eq_coind.
```

We omit the output and proceed to proving ones_eq' again. The only bit of ingenuity lies in choosing R, and in this case the most restrictive predicate works.

```
Theorem ones_eq' : stream_eq ones ones'.
  apply (stream_eq_coind (fun s1 s2 ⇒ s1 = ones ∧ s2 = ones')); crush.
Qed.
```

Note that this proof achieves the proper reduction behavior via hd and tl rather than frob, as in the last proof. All three functions pattern-match on their arguments, catalyzing computation steps.

Compared to inductive proofs, it still seems unsatisfactory that we had to write a choice of R in the last proof. An alternative is to capture a common pattern of co-recursion in a more specialized co-induction principle. For the current example, that pattern is, prove **stream_eq** *s1 s2*, where *s1* and *s2* are defined as their own tails.

```
Section stream_eq_loop.
  Variable A : Type.
  Variables s1 s2 : stream A.

  Hypothesis Cons_case_hd : hd s1 = hd s2.
  Hypothesis loop1 : tl s1 = s1.
  Hypothesis loop2 : tl s2 = s2.
```

The proof of the principle includes a choice of R so that we no longer need to make such choices.

```
  Theorem stream_eq_loop : stream_eq s1 s2.
    apply (stream_eq_coind (fun s1' s2' ⇒ s1' = s1 ∧ s2' = s2));
      crush.
  Qed.
End stream_eq_loop.

Theorem ones_eq'' : stream_eq ones ones'.
  apply stream_eq_loop; crush.
Qed.
```

Let us put stream_eq_coind through its paces a bit more, considering two different ways to compute infinite streams of all factorial values. First, we import the fact factorial function from the standard library.

```
Require Import Arith.
Print fact.

fact =
fix fact (n : nat) : nat :=
  match n with
  | 0 ⇒ 1
  | S n0 ⇒ S n0 × fact n0
  end
     : nat → nat
```

The simplest way to compute the factorial stream involves calling fact afresh at each position.

```
CoFixpoint fact_slow' (n : nat) := Cons (fact n) (fact_slow' (S n)).
Definition fact_slow := fact_slow' 1.
```

An optimized method maintains an accumulator of the previous factorial so that each new entry can be computed with a single multiplication.

```
CoFixpoint fact_iter' (cur acc : nat) :=
  Cons acc (fact_iter' (S cur) (acc × cur)).
Definition fact_iter := fact_iter' 2 1.
```

We can verify that the streams are equal up to particular finite bounds.

```
Eval simpl in approx fact_iter 5.
```

```
= 1 :: 2 :: 6 :: 24 :: 120 :: nil
: list nat
```

```
Eval simpl in approx fact_slow 5.
```

```
= 1 :: 2 :: 6 :: 24 :: 120 :: nil
: list nat
```

Now, to prove that the two versions are equivalent, it is helpful to prove (and add as a proof hint) a lemma about the computational behavior of fact.

```
Lemma fact_def : ∀ x n,
  fact_iter' x (fact n × S n) = fact_iter' x (fact (S n)).
  simpl; intros; f_equal; ring.
Qed.

Hint Resolve fact_def.
```

With the hint added, it is easy to prove an auxiliary lemma relating fact_iter' and fact_slow'. The key is introduction of an existential quantifier for the shared parameter n.

```
Lemma fact_eq' : ∀ n,
  stream_eq (fact_iter' (S n) (fact n)) (fact_slow' n).
  intro; apply (stream_eq_coind (fun s1 s2 ⇒ ∃ n,
    s1 = fact_iter' (S n) (fact n)
    ∧ s2 = fact_slow' n)); crush; eauto.
Qed.
```

The final theorem is a direct corollary of fact_eq'.

```
Theorem fact_eq : stream_eq fact_iter fact_slow.
```

```
  apply fact_eq'.
Qed.
```

As in the case of ones_eq', we may be unsatisfied that we need to write a choice of R that seems to duplicate information already present in a lemma statement. We can facilitate a simpler proof by defining a co-induction principle specialized to goals that begin with single universal quantifiers, and the strategy can be extended in a straightforward way to principles for other counts of quantifiers. (The stream_eq_loop principle is effectively the instantiation of this technique to zero quantifiers.)

```
Section stream_eq_onequant.
  Variables A B : Type.
```

We have the type A, the domain of the one quantifier; and type B, the type of data found in the streams.

```
  Variables f g : A → stream B.
```

The two streams we compare must be of the forms $f\ x$ and $g\ x$, for some shared x. Note that this falls out naturally when x is a shared universally quantified variable in a lemma statement.

```
  Hypothesis Cons_case_hd : ∀ x, hd (f x) = hd (g x).
  Hypothesis Cons_case_tl : ∀ x, ∃ y, tl (f x) = f y ∧ tl (g x) = g y.
```

These conditions are inspired by the bisimulation requirements, with a more general version of the R choice we made for fact_eq' inlined into the hypotheses of stream_eq_coind.

```
  Theorem stream_eq_onequant : ∀ x, stream_eq (f x) (g x).
    intro; apply (stream_eq_coind (fun s1 s2 ⇒ ∃ x,
      s1 = f x ∧ s2 = g x)); crush; eauto.
  Qed.
End stream_eq_onequant.

Lemma fact_eq'' : ∀ n,
  stream_eq (fact_iter' (S n) (fact n)) (fact_slow' n).
  apply stream_eq_onequant; crush; eauto.
Qed.
```

We have arrived at a customary automated proof, thanks to the new principle.

5.3 Simple Modeling of Nonterminating Programs

This chapter closes with a brief example of more complex uses of co-inductive types. We define a co-inductive semantics for a simple

imperative programming language and use that semantics to prove the correctness of a trivial optimization that removes spurious additions by 0. We follow the technique of *co-inductive big-step operational semantics* [20].

We define a suggestive synonym for **nat**, as we will consider programs over infinitely many variables, represented as **nat**s.

```
Definition var := nat.
```

We define a type vars of maps from variables to values. To define a function set for setting a variable's value in a map, we use the standard library function beq_nat for comparing natural numbers.

```
Definition vars := var → nat.
Definition set (vs : vars) (v : var) (n : nat) : vars :=
  fun v' ⇒ if beq_nat v v' then n else vs v'.
```

We define a simple arithmetic expression language with variables and give it a semantics via an interpreter.

```
Inductive exp : Set :=
| Const : nat → exp
| Var : var → exp
| Plus : exp → exp → exp.

Fixpoint evalExp (vs : vars) (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Var v ⇒ vs v
    | Plus e1 e2 ⇒ evalExp vs e1 + evalExp vs e2
  end.
```

Finally, we define a language of commands. It includes variable assignment, sequencing, and a `while` form that repeats as long as its test expression evaluates to a nonzero value.

```
Inductive cmd : Set :=
| Assign : var → exp → cmd
| Seq : cmd → cmd → cmd
| While : exp → cmd → cmd.
```

We could define an inductive relation to characterize the results of command evaluation. However, such a relation would not capture *nonterminating* executions. With a co-inductive relation, we can capture both cases. The parameters of the relation are an initial state, a command, and a final state. A program that does not terminate in a particular initial state is related to *any* final state. For more realistic languages than this one, it is often possible for programs to *crash*,

in which case a semantics would generally relate their executions to no final states. Thus, relating safely nonterminating programs to all final states provides a crucial distinction.

```
CoInductive evalCmd : vars → cmd → vars → Prop :=
| EvalAssign : ∀ vs v e,
  evalCmd vs (Assign v e) (set vs v (evalExp vs e))
| EvalSeq : ∀ vs1 vs2 vs3 c1 c2, evalCmd vs1 c1 vs2
  → evalCmd vs2 c2 vs3
  → evalCmd vs1 (Seq c1 c2) vs3
| EvalWhileFalse : ∀ vs e c, evalExp vs e = 0
  → evalCmd vs (While e c) vs
| EvalWhileTrue : ∀ vs1 vs2 vs3 e c, evalExp vs1 e ≠ 0
  → evalCmd vs1 c vs2
  → evalCmd vs2 (While e c) vs3
  → evalCmd vs1 (While e c) vs3.
```

Before proceeding, we build a co-induction principle for **evalCmd**.

```
Section evalCmd_coind.
  Variable R : vars → cmd → vars → Prop.

  Hypothesis AssignCase : ∀ vs1 vs2 v e, R vs1 (Assign v e) vs2
    → vs2 = set vs1 v (evalExp vs1 e).

  Hypothesis SeqCase : ∀ vs1 vs3 c1 c2, R vs1 (Seq c1 c2) vs3
    → ∃ vs2, R vs1 c1 vs2 ∧ R vs2 c2 vs3.

  Hypothesis WhileCase : ∀ vs1 vs3 e c, R vs1 (While e c) vs3
    → (evalExp vs1 e = 0 ∧ vs3 = vs1)
    ∨ ∃ vs2, evalExp vs1 e ≠ 0 ∧ R vs1 c vs2
      ∧ R vs2 (While e c) vs3.
```

The proof is routine. We make use of a form of `destruct` that takes an *intro pattern* in an `as` clause. These patterns control how deeply we break apart the components of an inductive value (see the Coq manual for more details).

```
  Theorem evalCmd_coind : ∀ vs1 c vs2, R vs1 c vs2
    → evalCmd vs1 c vs2.
    cofix; intros; destruct c.
    rewrite (AssignCase H); constructor.
    destruct (SeqCase H) as [? [? ?]]; econstructor; eauto.
    destruct (WhileCase H) as [[? ?] | [? [? [? ?]]]];
      subst; econstructor; eauto.
  Qed.
End evalCmd_coind.
```

Now that we have a co-induction principle, we should use it to prove something. The example is a trivial program optimizer that finds places to replace $0 + e$ with e.

```
Fixpoint optExp (e : exp) : exp :=
  match e with
    | Plus (Const 0) e ⇒ optExp e
    | Plus e1 e2 ⇒ Plus (optExp e1) (optExp e2)
    | _ ⇒ e
  end.

Fixpoint optCmd (c : cmd) : cmd :=
  match c with
    | Assign v e ⇒ Assign v (optExp e)
    | Seq c1 c2 ⇒ Seq (optCmd c1) (optCmd c2)
    | While e c ⇒ While (optExp e) (optCmd c)
  end.
```

Before proving correctness of optCmd, we prove a lemma about optExp. This is where we have to do the most work, choosing pattern-matching opportunities automatically.

```
Lemma optExp_correct : ∀ vs e, evalExp vs (optExp e) = evalExp vs e.
  induction e; crush;
    repeat (match goal with
              | [ ⊢ context[match ?E with Const _ ⇒ _
                                        | _ ⇒ _ end] ] ⇒ destruct E
              | [ ⊢ context[match ?E with O ⇒ _
                                        | S _ ⇒ _ end] ] ⇒ destruct E
            end; crush).
Qed.

Hint Rewrite optExp_correct.
```

The final theorem is easy to establish using the co-induction principle and a bit of Ltac proof automation (see Chapter 14). At a high level, we show inclusions between behaviors, going in both directions between original and optimized programs.

```
Ltac finisher := match goal with
                   | [ H : evalCmd _ _ _ ⊢ _ ] ⇒ ((inversion H; [])
                     || (inversion H; [||])); subst
                 end; crush; eauto 10.

Lemma optCmd_correct1 : ∀ vs1 c vs2, evalCmd vs1 c vs2
  → evalCmd vs1 (optCmd c) vs2.
  intros; apply (evalCmd_coind (fun vs1 c' vs2 ⇒
```

```
    ∃ c, evalCmd vs1 c vs2 ∧ c' = optCmd c));
    eauto; crush;
    match goal with
      | [ H : _ = optCmd ?E ⊢ _ ] ⇒
        destruct E; simpl in *; discriminate
        || injection H; intros; subst
    end; finisher.
Qed.

Lemma optCmd_correct2 : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  → evalCmd vs1 c vs2.
  intros; apply (evalCmd_coind (fun vs1 c vs2 ⇒
    evalCmd vs1 (optCmd c) vs2)); crush; finisher.
Qed.

Theorem optCmd_correct : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  ↔ evalCmd vs1 c vs2.
  intuition; apply optCmd_correct1 || apply optCmd_correct2;
    assumption.
Qed.
```

In this form, the theorem tells us that the optimizer preserves observable behavior of both terminating and nonterminating programs, but we did not have to do more work than for the case of terminating programs alone. We merely took the natural inductive definition for terminating executions, made it co-inductive, and applied the appropriate co-induction principle. Readers might want to experiment with adding command constructs like `if`; the same proof script should continue working after the co-induction principle is extended to the new evaluation rules.