MAPS

TOTAL AND PARTIAL MAPS

Maps (or *dictionaries*) are ubiquitous data structures both generally and in the theory of programming languages in particular; we're going to need them in many places in the coming chapters. They also make a nice case study using ideas we've seen in previous chapters, including building data structures out of higher-order functions (from `Basics` and `Poly`) and the use of reflection to streamline proofs (from `IndProp`).

We'll define two flavors of maps: *total* maps, which include a "default" element to be returned when a key being looked up doesn't exist, and *partial* maps, which return an `option` to indicate success or failure. The latter is defined in terms of the former, using `None` as the default element.

The Coq Standard Library

One small digression before we begin...

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (and, transitively, all the earlier chapters). Instead, in this chapter and from now, on we're going to import the definitions and theorems we need directly from Coq's standard library stuff. You should not notice much difference, though, because we've been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
Require Import Coq.Arith.Arith.
Require Import Coq.Bool.Bool.
Require Export Coq.Strings.String.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import Coq.Lists.List.
Import ListNotations.
```

Documentation for the standard library can be found at http://coq.inria.fr/library/.

The `Search` command is a good way to look for theorems involving objects of specific types. Take a minute now to experiment with it.

Identifiers

First, we need a type for the keys that we use to index into our maps. For this purpose, we will simply use plain `string`s.

To compare strings, we define the function `beq_string`, which internally uses the function `string_dec` from Coq's string library. We then establish its fundamental properties.

```
Definition beq_string x y :=
  if string_dec x y then true else false.
```

(The function `string_dec` comes from Coq's string library. If you check the result type of `string_dec`, you'll see that it does not actually return a `bool`, but rather a type that looks like `{x = y} + {x ≠ y}`, called a `sumbool`, which can be thought of as an "evidence-carrying boolean." Formally, an element of `sumbool` is either a proof that two things are equal or a proof that they are unequal, together with a tag indicating which. But for present purposes you can think of it as just a fancy `bool`.)

```
Theorem beq_string_refl : ∀ s, true = beq_string s s.
+
```

The following useful property of `beq_string` follows from an analogous lemma about strings:

```
Theorem beq_string_true_iff : ∀ x y : string,
  beq_string x y = true ↔ x = y.
+
```

Similarly:

```
Theorem beq_string_false_iff : ∀ x y : string,
  beq_string x y = false
  ↔ x ≠ y.
+
```

This useful variant follows just by rewriting:

```
Theorem false_beq_string : ∀ x y : string,
   x ≠ y → beq_string x y = false.
+
```

Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the Lists chapter, plus accompanying lemmas about its behavior.

This time around, though, we're going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more *extensional* view of maps, where two maps that respond to queries in the same way will be represented as literally the same thing (the very same function), rather than just "equivalent" data structures. This, in turn, simplifies proofs that use maps.

We build partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

```
Definition total_map (A:Type) := string → A.
```

Intuitively, a total map over an element type `A` is just a function that can be used to look up `string`s, yielding `A`s.

The function `t_empty` yields an empty total map, given a default element; this map always returns the default element when applied to any string.

```
Definition t_empty {A:Type} (v : A) : total_map A :=
  (fun _ ⇒ v).
```

More interesting is the `update` function, which (as before) takes a map `m`, a key `x`, and a value `v` and returns a new map that takes `x` to `v` and takes every other key to whatever `m` does.

```
Definition t_update {A:Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x' ⇒ if beq_string x x' then v else m x'.
```

This definition is a nice example of higher-order programming: `t_update` takes a *function* `m` and yields a new function `fun x' ⇒ ...` that behaves like the desired map.

For example, we can build a map taking `string`s to `bool`s, where `"foo"` and `"bar"` are mapped to `true` and every other key is mapped to `false`, like this:

```
Definition examplemap :=
  t_update (t_update (t_empty false) "foo" true)
           "bar" true.
```

Next, let's introduce some new notations to facilitate working with maps.

First, we will use the following notation to create an empty total map with a default value.

```
Notation "{ --> d }" := (t_empty d) (at level 0).
```

We then introduce a convenient notation for extending an existing map with some bindings.

(The definition of the notation is a bit ugly, but because the notation mechanism of Coq is not very well suited for recursive notations, it's the best we can do.)

```
Notation "m '&' { a --> x }" :=
  (t_update m a x) (at level 20).
Notation "m '&' { a --> x ; b --> y }" :=
  (t_update (m & { a --> x }) b y) (at level 20).
Notation "m '&' { a --> x ; b --> y ; c --> z }" :=
  (t_update (m & { a --> x ; b --> y }) c z) (at level 20).
Notation "m '&' { a --> x ; b --> y ; c --> z ; d --> t }" :=
    (t_update (m & { a --> x ; b --> y ; c --> z }) d t) (at level
20).
Notation "m '&' { a --> x ; b --> y ; c --> z ; d --> t ; e --> u }"
:=
    (t_update (m & { a --> x ; b --> y ; c --> z ; d --> t }) e u)
(at level 20).
Notation "m '&' { a --> x ; b --> y ; c --> z ; d --> t ; e --> u ; f
--> v }" :=
    (t_update (m & { a --> x ; b --> y ; c --> z ; d --> t ; e --> u
}) f v) (at level 20).
```

The `examplemap` above can now be defined as follows:

```
Definition examplemap' :=
  { --> false } & { "foo" --> true ; "bar" --> true }.
```

This completes the definition of total maps. Note that we don't need to define a `find` operation because it is just function application!

To use maps in later chapters, we'll need several fundamental facts about how they behave.

Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas!

(Some of the proofs require the functional extensionality axiom, which is discussed in the Logic chapter.)

Exercise: 1 star, optional (t_apply_empty)

First, the empty map returns its default element for all keys:

```
Lemma t_apply_empty: ∀ (A:Type) (x: string) (v: A), { --> v } x =
v.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars, optional (t_update_eq)

Next, if we update a map `m` at a key `x` with a new value `v` and then look up `x` in the map resulting from the `update`, we get back `v`:

```
Lemma t_update_eq : ∀ A (m: total_map A) x v,
  (m & {x --> v}) x = v.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars, optional (t_update_neq)

On the other hand, if we update a map `m` at a key x_1 and then look up a *different* key x_2 in the resulting map, we get the same result that `m` would have given:

```
Theorem t_update_neq : ∀ (X:Type) v x1 x2
                         (m : total_map X),
  x1 ≠ x2 →
  (m & {x1 --> v}) x2 = m x2.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars, optional (t_update_shadow)

If we update a map `m` at a key `x` with a value v_1 and then update again with the same key `x` and another value v_2, the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second `update` on `m`:

```
Lemma t_update_shadow : ∀ A (m: total_map A) v1 v2 x,
    m & {x --> v1 ; x --> v2} = m & {x --> v2}.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

For the final two lemmas about total maps, it's convenient to use the reflection idioms introduced in chapter IndProp. We begin by proving a fundamental *reflection lemma* relating the equality proposition on `id`s with the boolean function `beq_id`.

Exercise: 2 stars, optional (beq_stringP)

Use the proof of `beq_natP` in chapter IndProp as a template to prove the following:

```
Lemma beq_stringP : ∀ x y, reflect (x = y) (beq_string x y).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Now, given `string`s x_1 and x_2, we can use the `destruct (beq_stringP x1 x2)` to simultaneously perform case analysis on the result of `beq_string` x_1 x_2 and generate hypotheses about the equality (in the sense of `=`) of x_1 and x_2.

Exercise: 2 stars (t_update_same)

With the example in chapter IndProp as a template, use `beq_stringP` to prove the following theorem, which states that if we update a map to assign key `x` the same value as it already has in `m`, then the result is equal to `m`:

```
Theorem t_update_same : ∀ X x (m : total_map X),
    m & { x --> m x } = m.
```

```
  Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, recommended (t_update_permute)

Use `beq_stringP` to prove one final property of the `update` function: If we update a map `m` at two distinct keys, it doesn't matter in which order we do the updates.

```
Theorem t_update_permute : ∀ (X:Type) v1 v2 x1 x2
                             (m : total_map X),
  x2 ≠ x1 →
  m & { x2 --> v2 ; x1 --> v1 }
  = m & { x1 --> v1 ; x2 --> v2 }.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Partial maps

Finally, we define *partial maps* on top of total maps. A partial map with elements of type `A` is simply a total map with elements of type `option A` and default element `None`.

```
Definition partial_map (A:Type) := total_map (option A).

Definition empty {A:Type} : partial_map A :=
  t_empty None.

Definition update {A:Type} (m : partial_map A)
           (x : string) (v : A) :=
  m & { x --> (Some v) }.
```

We introduce a similar notation for partial maps, using double curly-brackets.

```
Notation "m '&' {{ a --> x }}" :=
  (update m a x) (at level 20).
Notation "m '&' {{ a --> x ; b --> y }}" :=
  (update (m & {{ a --> x }}) b y) (at level 20).
Notation "m '&' {{ a --> x ; b --> y ; c --> z }}" :=
  (update (m & {{ a --> x ; b --> y }}) c z) (at level 20).
Notation "m '&' {{ a --> x ; b --> y ; c --> z ; d --> t }}" :=
    (update (m & {{ a --> x ; b --> y ; c --> z }}) d t) (at level
20).
Notation "m '&' {{ a --> x ; b --> y ; c --> z ; d --> t ; e --> u }}"
:=
    (update (m & {{ a --> x ; b --> y ; c --> z ; d --> t }}) e u)
(at level 20).
Notation "m '&' {{ a --> x ; b --> y ; c --> z ; d --> t ; e --> u ;
f --> v }}" :=
    (update (m & {{ a --> x ; b --> y ; c --> z ; d --> t ; e --> u
}}) f v) (at level 20).
```

We now straightforwardly lift all of the basic lemmas about total maps to partial maps.

```
Lemma apply_empty : ∀ (A: Type) (x: string), @empty A x = None.
+

Lemma update_eq : ∀ A (m: partial_map A) x v,
    (m & {{ x --> v }}) x = Some v.
+

Theorem update_neq : ∀ (X:Type) v x₁ x₂
                       (m : partial_map X),
  x₂ ≠ x₁ →
  (m & {{ x₂ --> v }}) x₁ = m x₁.
+

Lemma update_shadow : ∀ A (m: partial_map A) v₁ v₂ x,
    m & {{ x --> v₁ ; x --> v₂ }} = m & {{x --> v₂}}.
+

Theorem update_same : ∀ X v x (m : partial_map X),
  m x = Some v →
  m & {{x --> v}} = m.
+

Theorem update_permute : ∀ (X:Type) v₁ v₂ x₁ x₂
                                  (m : partial_map X),
  x₂ ≠ x₁ →
  m & {{x₂ --> v₂ ; x₁ --> v₁}}
  = m & {{x₁ --> v₁ ; x₂ --> v₂}}.
+
```