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.

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

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

The `Search` command is a good way to look for theorems involving objects of specific types. See Lists for a reminder of how to use it.

If you want to find out how or where a notation is defined, the `Locate` command is useful. For example, where is the natural addition operation defined in the standard library?

```
Locate "+".
```

There are several uses for that notation, but only one for naturals.

```
Print Init.Nat.add.
```

Identifiers

First, we need a type for the keys that we use to index into our maps. In `Lists.v` we introduced a fresh type `id` for a similar purpose; here and for the rest of *Software Foundations* we will use the `string` type from Coq's standard library.

To compare strings, we define the function `eqb_string`, which internally uses the function `string_dec` from Coq's string library.

```
Definition eqb_string (x y : string) : bool :=
  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 $\{x = y\} + \{x \neq y\}$, called a `sumbool`, which can be thought of as an "evidence-carrying boolean." Formally, an element of $\{x = y\} + \{x \neq y\}$ is either a proof that x and y 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`.)

Now we need a few basic properties of string equality...

```
Theorem eqb_string_refl : ∀ s : string, true = eqb_string s s.
```
+

Two strings are equal according to `eqb_string` iff they are equal according to `=`. So `=` is reflected in `eqb_string`, in the sense of "reflection" as discussed in IndProp.

```
Theorem eqb_string_true_iff : ∀ x y : string,
    eqb_string x y = true ↔ x = y.
```
+

Similarly:

```
Theorem eqb_string_false_iff : ∀ x y : string,
    eqb_string x y = false ↔ x ≠ y.
```
+

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 eqb_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 "'_' '!->' v" := (t_empty v)
  (at level 100, right associativity).

Example example_empty := (_ !-> false).
```

We then introduce a convenient notation for extending an existing map with some bindings.
```
Notation "x '!->' v ';' m" := (t_update m x v)
                              (at level 100, v at next level, right associativity).
```

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

```
Definition examplemap' :=
  ( "bar" !-> true;
    "foo" !-> true;
    _ !-> false
  ).
```

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, standard, 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, standard, 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 : Type) (m : total_map A) x v,
    (x !-> v ; m) x = v.
Proof.
  (* FILL IN HERE *) Admitted.
  □
```

Exercise: 2 stars, standard, 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 : ∀ (A : Type) (m : total_map A) x₁ x₂ v,
    x₁ ≠ x₂ →
    (x₁ !-> v ; m) x₂ = m x₂.
Proof.
  (* FILL IN HERE *) Admitted.
  □
```

Exercise: 2 stars, standard, 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 : Type) (m : total_map A) x v₁ v₂,
    (x !-> v₂ ; x !-> v₁ ; m) = (x !-> v₂ ; m).
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 strings with the boolean function `eqb_string`.

Exercise: 2 stars, standard, optional (eqb_stringP)

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

```
Lemma eqb_stringP : ∀ x y : string,
    reflect (x = y) (eqb_string x y).
Proof.
    (* FILL IN HERE *) Admitted.
□
```

Now, given strings x_1 and x_2, we can use the tactic `destruct (eqb_stringP x_1 x_2)` to simultaneously perform case analysis on the result of `eqb_string x_1 x_2` and generate hypotheses about the equality (in the sense of `=`) of x_1 and x_2.

Exercise: 2 stars, standard (t_update_same)

With the example in chapter IndProp as a template, use `eqb_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 : ∀ (A : Type) (m : total_map A) x,
    (x !-> m x ; m) = m.
Proof.
    (* FILL IN HERE *) Admitted.
□
```

Exercise: 3 stars, standard, especially useful (t_update_permute)

Use `eqb_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 : ∀ (A : Type) (m : total_map A)
                                v_1 v_2 x_1 x_2,
    x_2 ≠ x_1 →
    (x_1 !-> v_1 ; x_2 !-> v_2 ; m)
    =
    (x_2 !-> v_2 ; x_1 !-> v_1 ; m).
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) :=
    (x !-> Some v ; m).
```

We introduce a similar notation for partial maps:
```
Notation "x '|->' v ';' m" := (update m x v)
    (at level 100, v at next level, right associativity).
```

We can also hide the last case when it is empty.
```
Notation "x '|->' v" := (update empty x v)
    (at level 100).

Example examplepmap :=
    ("Church" |-> true ; "Turing" |-> false).
```

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 : Type) (m : partial_map A) x v,
    (x |-> v ; m) x = Some v.
+
```

```
Theorem update_neq : ∀ (A : Type) (m : partial_map A) x_1 x_2 v,
    x_2 ≠ x_1 →
    (x_2 |-> v ; m) x_1 = m x_1.
+
```

```
Lemma update_shadow : ∀ (A : Type) (m : partial_map A) x v_1 v_2,
    (x |-> v_2 ; x |-> v_1 ; m) = (x |-> v_2 ; m).
+
```

```
Theorem update_same : ∀ (A : Type) (m : partial_map A) x v,
    m x = Some v →
    (x |-> v ; m) = m.
+
```

```
Theorem update_permute : ∀ (A : Type) (m : partial_map A)
                                x_1 x_2 v_1 v_2,
    x_2 ≠ x_1 →
    (x_1 |-> v_1 ; x_2 |-> v_2 ; m) = (x_2 |-> v_2 ; x_1 |-> v_1 ; m).
+
```

Finally, for partial maps we introduce a notion of map inclusion, stating that one map includes another:

```
Definition inclusion {A : Type} (m m' : partial_map A) :=
    ∀ x v, m x = Some v → m' x = Some v.
```

We show that map update preserves map inclusion, that is:

```
Lemma inclusion_update : ∀ (A : Type) (m m': partial_map A)
                                x vx,
    inclusion m m' →
    inclusion (x |-> vx ; m) (x |-> vx ; m').
+
```

This property is useful for reasoning about the lambda-calculus, where maps are used to keep track of which program variables are defined at a given point.

```
(* 2020-09-09 20:51 *)
```