15 Maps and Hash Tables

Lots of programming problems require dealing with data organized as key/value pairs. Maybe the simplest way of representing such data in OCaml is an *association list*, which is simply a list of pairs of keys and values. For example, you could represent a mapping between the 10 digits and their English names as follows:

```
# open Base;;
# let digit_alist =
    [ 0, "zero"; 1, "one"; 2, "two"  ; 3, "three"; 4, "four"
    ; 5, "five"; 6, "six"; 7, "seven"; 8, "eight"; 9, "nine" ];;
val digit_alist : (int * string) list =
  [(0, "zero"); (1, "one"); (2, "two"); (3, "three"); (4, "four");
   (5, "five"); (6, "six"); (7, "seven"); (8, "eight"); (9, "nine")]
```

We can use functions from the `List.Assoc` module to manipulate this data:

```
# List.Assoc.find ~equal:Int.equal digit_alist 6;;
- : string option = Some "six"
# List.Assoc.find ~equal:Int.equal digit_alist 22;;
- : string option = None
# List.Assoc.add ~equal:Int.equal digit_alist 0 "zilch";;
- : (int, string) Base.List.Assoc.t =
[(0, "zilch"); (1, "one"); (2, "two"); (3, "three"); (4, "four");
 (5, "five"); (6, "six"); (7, "seven"); (8, "eight"); (9, "nine")]
```

Association lists are simple and easy to use, but their performance is not ideal, since almost every nontrivial operation on an association list requires a linear-time scan of the list.

In this chapter, we'll talk about two more efficient alternatives to association lists: *maps* and *hash tables*. A map is an immutable tree-based data structure where most operations take time logarithmic in the size of the map, whereas a hash table is a mutable data structure where most operations have constant time complexity. We'll describe both of these data structures in detail and provide some advice as to how to choose between them.

15.1 Maps

Let's consider an example of how one might use a map in practice. In Chapter 5 (Files, Modules, and Programs), we showed a module `Counter` for keeping frequency counts on a set of strings. Here's the interface:

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
open Base

(** A collection of string frequency counts *)
type t

(** The empty set of frequency counts *)
val empty : t

(** Bump the frequency count for the given string. *)
val touch : t -> string -> t

(** Converts the set of frequency counts to an association list. Every
    string in the list will show up at most once, and the integers
    will be at least 1. *)
val to_list : t -> (string * int) list
```

The intended behavior here is straightforward. `Counter.empty` represents an empty collection of frequency counts; `touch` increments the frequency count of the specified string by 1; and `to_list` returns the list of nonzero frequencies.

Here's the implementation.

```
open Base

type t = (string, int, String.comparator_witness) Map.t

let empty = Map.empty (module String)
let to_list t = Map.to_alist t

let touch t s =
  let count =
    match Map.find t s with
    | None -> 0
    | Some x -> x
  in
  Map.set t ~key:s ~data:(count + 1)
```

Take a look at the definition of the type `t` above. You'll see that the `Map.t` has three type parameters. The first two are what you might expect; one for the type of the key, and one for the type of the data. The third type parameter, the *comparator witness*, requires some explaining.

The comparator witness is used to indicate which comparison function was used to construct the map, rather than saying something about the type of data stored in the map. The type `String.comparator_witness` in particular indicates that this map was built with the default comparison function from the `String` module. We'll talk about why the comparator witness is important later in the chapter.

The call to `Map.empty` is also worth explaining, in that, unusually, it takes a first-class module as an argument. The point of the first class module is to provide the comparison function that is required for building the map, along with an s-expression converter for generating useful error messages (we'll talk more about s-expressions in Chapter 21 (Data Serialization with S-Expressions)). We don't need to provide the module again for functions like `Map.find` or `Map.add`, because the map itself contains a reference to the comparison function it uses.

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

Not every module can be used for creating maps, but the standard ones in `Base` can. Later in the chapter, we'll show how you can set up a module of your own so it can be used in this way.

15.1.1 Sets

In addition to maps, `Base` also provides a set data type that's designed along similar lines. In some sense, sets are little more than maps where you ignore the data. But while you could encode sets in terms of maps, it's more natural, and more efficient, to use `Base`'s specialized set type. Here's a simple example.

```
# Set.of_list (module Int) [1;2;3] |> Set.to_list;;
- : int list = [1; 2; 3]
# Set.union
    (Set.of_list (module Int) [1;2;3;2])
    (Set.of_list (module Int) [3;5;1])
  |> Set.to_list;;
- : int list = [1; 2; 3; 5]
```

In addition to the operators you would expect to have for maps, sets support the traditional set operations, including union, intersection, and set difference.

15.1.2 Modules and Comparators

It's easy enough to create a map or set based on a type represented by a module in `Base`. Here, we'll create a map from digits to their English names, based on `digit_alist`, which was defined earlier in the chapter.

```
# let digit_map = Map.of_alist_exn (module Int) digit_alist;;
val digit_map : (int, string, Int.comparator_witness) Map.t = <abstr>
# Map.find digit_map 3;;
- : string option = Some "three"
```

The function `Map.of_alist_exn` constructs a map from a provided association list, throwing an exception if a key is used more than once. Let's take a look at the type signature of `Map.of_alist_exn`.

```
# #show Map.of_alist_exn;;
val of_alist_exn :
  ('a, 'cmp) Map.comparator -> ('a * 'b) list -> ('a, 'b, 'cmp) Map.t
```

The type `Map.comparator` is actually an alias for a first-class module type, representing any module that matches the signature `Comparator.S`, shown below.

```
# #show Base.Comparator.S;;
module type S =
  sig
    type t
    type comparator_witness
    val comparator : (t, comparator_witness) Comparator.t
  end
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

Such a module must contain the type of the key itself, as well as the `comparator_witness` type, which serves as a type-level identifier of the comparison function in question, and finally, the concrete comparator itself, a value that contains the necessary comparison function.

Modules from `Base` like `Int` and `String` already satisfy this interface. But what if you want to satisfy this interface with a new module? Consider, for example, the following type representing a book, for which we've written a comparison function and an s-expression serializer.

```
# module Book = struct

    type t = { title: string; isbn: string }

    let compare t1 t2 =
      let cmp_title = String.compare t1.title t2.title in
      if cmp_title <> 0 then cmp_title
      else String.compare t1.isbn t2.isbn

    let sexp_of_t t : Sexp.t =
      List [ Atom t.title; Atom t.isbn ]
  end;;
module Book :
  sig
    type t = { title : string; isbn : string; }
    val compare : t -> t -> int
    val sexp_of_t : t -> Sexp.t
  end
```

This module has the basic functionality we need, but doesn't satisfy the `Comparator.S` interface, so we can't use it for creating a map, as you can see.

```
# Map.empty (module Book);;
Line 1, characters 19-23:
Error: Signature mismatch:
       ...
       The type `comparator_witness' is required but not provided
       File "duniverse/base/src/comparator.mli", line 19, characters
    2-25:
         Expected declaration
       The value `comparator' is required but not provided
       File "duniverse/base/src/comparator.mli", line 21, characters
    2-53:
         Expected declaration
```

In order to satisfy the interface, we need to use the `Comparator.Make` functor to extend the module. Here, we use a common idiom where we create a submodule, called `T` containing the basic functionality for the type in question, and then include both that module and the result of applying a functor to that module.

```
module Book = struct
  module T = struct

    type t = { title: string; isbn: string }

    let compare t1 t2 =
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
      let cmp_title = String.compare t1.title t2.title in
      if cmp_title <> 0 then cmp_title
      else String.compare t1.isbn t2.isbn

    let sexp_of_t t : Sexp.t =
      List [ Atom t.title; Atom t.isbn ]

  end
  include T
  include Comparator.Make(T)
end;;
```

With this module in hand, we can now build a set of `Book.t`'s.

```
# let some_programming_books =
    Set.of_list (module Book)
      [ { title = "Real World OCaml"
        ; isbn = "978-1449323912" }
      ; { title = "Structure and Interpretation of Computer Programs"
        ; isbn = "978-0262510875" }
      ; { title = "The C Programming Language"
        ; isbn = "978-0131101630" } ];;
val some_programming_books : (Book.t, Book.comparator_witness) Set.t =
  <abstr>
```

While we used `Comparator.Make` in the above, it's often preferable to use `Comparable.Make` instead, since it provides extra helper functions, like infix comparison operators and min and max functions, in addition to the comparator itself.

15.1.3 Why Do We Need Comparator Witnesses?

The comparator witness is quite different from other types that we've seen. Instead of tracking the kind of data being used, it's used to single out a particular value, a comparison function. Why do we even need such a thing?

The comparator witness matters because some of the operations on maps and sets, in particular those that combine multiple maps or sets together, depend for their correctness on the fact that both objects being combined are ordered according to the same total order, which in turn is determined by the comparison function.

Consider, for example, `Map.symmetric_diff`, which computes the difference between two maps.

```
# let left = Map.of_alist_exn (module String) ["foo",1; "bar",3;
    "snoo",0];;
val left : (string, int, String.comparator_witness) Map.t = <abstr>
# let right = Map.of_alist_exn (module String) ["foo",0; "snoo",0];;
val right : (string, int, String.comparator_witness) Map.t = <abstr>
# Map.symmetric_diff ~data_equal:Int.equal left right |>
    Sequence.to_list;;
- : (string, int) Map.Symmetric_diff_element.t list =
[("bar", `Left 3); ("foo", `Unequal (1, 0))]
```

As you can see below, the type of `Map.symmetric_diff` requires that the two maps

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

it compares have the same comparator witness, in addition to the same key and value type.

```
# #show Map.symmetric_diff;;
val symmetric_diff :
  ('k, 'v, 'cmp) Map.t ->
  ('k, 'v, 'cmp) Map.t ->
  data_equal:('v -> 'v -> bool) ->
  ('k, 'v) Map.Symmetric_diff_element.t Sequence.t
```

Without this constraint, we could run `Map.symmetric_diff` on maps that are sorted in different orders, which could lead to garbled results.

To see this constraint in action, we'll need to create two maps with the same key and data types, but different comparison functions. In the following, we do this by minting a new module `Reverse`, which represents strings sorted in the reverse of the usual lexicographic order.

```
module Reverse = struct
  module T = struct
    type t = string
    let sexp_of_t = String.sexp_of_t
    let t_of_sexp = String.t_of_sexp
    let compare x y = String.compare y x
  end
  include T
  include Comparator.Make(T)
end;;
```

As you can see in the following, both `Reverse` and `String` can be used to create maps with a key type of `string`:

```
# let alist = ["foo", 0; "snoo", 3];;
val alist : (string * int) list = [("foo", 0); ("snoo", 3)]
# let ord_map = Map.of_alist_exn (module String) alist;;
val ord_map : (string, int, String.comparator_witness) Map.t = <abstr>
# let rev_map = Map.of_alist_exn (module Reverse) alist;;
val rev_map : (string, int, Reverse.comparator_witness) Map.t =
    <abstr>
```

`Map.min_elt` returns the key and value for the smallest key in the map, which confirms that these two maps do indeed use different comparison functions.

```
# Map.min_elt ord_map;;
- : (string * int) option = Some ("foo", 0)
# Map.min_elt rev_map;;
- : (string * int) option = Some ("snoo", 3)
```

As a result, the algorithm in `Map.symmetric_diff` just wouldn't work correctly when applied to these values. Happily, the type system will give us a compile-time error if we try, instead of throwing an error at run time, or worse, silently returning the wrong result.

```
# Map.symmetric_diff ord_map rev_map;;
Line 1, characters 28-35:
Error: This expression has type
         (string, int, Reverse.comparator_witness) Map.t
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
       but an expression was expected of type
         (string, int, String.comparator_witness) Map.t
       Type Reverse.comparator_witness is not compatible with type
         String.comparator_witness
```

15.1.4 The Polymorphic Comparator

We don't need to generate specialized comparators for every type we want to build a map on. We can instead build a map based on OCaml's built-in polymorphic comparison function, which was discussed in Chapter 4 (Lists and Patterns).

```
# Map.Poly.of_alist_exn digit_alist;;
- : (int, string) Map.Poly.t = <abstr>
```

Note that maps based on the polymorphic comparator have different comparator witnesses than those based on the type-specific comparison function. Thus, the compiler rejects the following:

```
# Map.symmetric_diff
    (Map.Poly.singleton 3 "three")
    (Map.singleton (module Int) 3 "four" );;
Line 3, characters 5-43:
Error: This expression has type (int, string, Int.comparator_witness)
    Map.t
       but an expression was expected of type
         (int, string, Comparator.Poly.comparator_witness) Map.t
       Type Int.comparator_witness is not compatible with type
         Comparator.Poly.comparator_witness
```

This is rejected for good reason: there's no guarantee that the comparator associated with a given type will order things in the same way that polymorphic compare does.

The Perils of Polymorphic Compare

Polymorphic compare is awfully convenient, but it has serious downsides and should mostly be avoided in production code. To understand why, it helps to understand how polymorphic compare works.

Polymorphic compare operates directly on the runtime representation of OCaml values, walking the structure of those values without regard for their type.

And despite ignoring types, it mostly behaves as you would hope. Comparisons on `int`s and `float`s respect the ordinary ordering of numeric values, and containers like strings, lists, and arrays are compared lexicographically. And it works on almost every OCaml type, with some important exceptions like functions.

But the type-oblivious nature of polymorphic compare means that it peeks under ordinary abstraction boundaries, and that can lead to some deeply confusing results. Maps themselves provide a great example of this. Consider the following two maps.

```
# let m1 = Map.of_alist_exn (module Int) [1, "one";2, "two"];;
val m1 : (int, string, Int.comparator_witness) Map.t = <abstr>
# let m2 = Map.of_alist_exn (module Int) [2, "two";1, "one"];;
val m2 : (int, string, Int.comparator_witness) Map.t = <abstr>
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

Logically, these two maps should be equal, and that's the result that you get if you call `Map.equal` on them:

```
# Map.equal String.equal m1 m2;;
- : bool = true
```

But because the elements were added in different orders, the layout of the trees underlying the maps will be different. As such, polymorphic compare will conclude that they're different.

We can see this below. Note that `Base` hides polymorphic comparison by default, but it is available within the `Poly` module.

```
# Poly.(m1 = m2);;
Exception: (Invalid_argument "compare: functional value")
```

This comparison failed because polymorphic compare doesn't work on functions, and maps store the comparison function they were created with. Happily, there's a function, `Map.Using_comparator.to_tree` which exposes the underlying binary tree without the attached comparison function. We can use that to compare the underlying trees:

```
# Poly.((Map.Using_comparator.to_tree m1) =
  (Map.Using_comparator.to_tree m2));;
- : bool = false
```

As you can see, polymorphic compare now produces a result, but it's not the result we want.

The abstraction-breaking nature of polymorphic compare can cause real and quite subtle bugs. If, for example, you build a map whose keys are sets (which have the same issues with polymorphic compare that maps do), then the map built with the polymorphic comparator will behave incorrectly, separating out keys that should be aggregated together. Even worse, it will behave inconsistently, since the behavior of polymorphic compare will depend on the order in which the sets were built.

15.1.5 Satisfying `Comparator.S` with `[@@deriving]`

Using maps and sets on a new type requires satisfying the `Comparator.S` interface, which in turn requires s-expression converters and comparison functions for the type in question. Writing such functions by hand is annoying and error prone, but there's a better way. `Base` comes along with a set of syntax extensions that automate these tasks away.

Let's return to an example from earlier in the chapter, where we created a type `Book.t` and set it up for use in creating maps and sets.

```
module Book = struct
  module T = struct

    type t = { title: string; isbn: string }

    let compare t1 t2 =
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
      let cmp_title = String.compare t1.title t2.title in
      if cmp_title <> 0 then cmp_title
      else String.compare t1.isbn t2.isbn

    let sexp_of_t t : Sexp.t =
      List [ Atom t.title; Atom t.isbn ]

  end
  include T
  include Comparator.Make(T)
end
```

Much of the code here is devoted to creating a comparison function and s-expression converter for the type `Book.t`. But if we have the `ppx_sexp_conv` and `ppx_compare` syntax extensions enabled, then we can request that default implementations of these functions be created for us. We can enable both of these extensions via the omnibus `ppx_jane` package.

```
# #require "ppx_jane";;
```

And we can use the extensions in our definition of `Book` as follows:

```
module Book = struct
  module T = struct
    type t = { title: string; isbn: string }
    [@@deriving compare, sexp_of]
  end
  include T
  include Comparator.Make(T)
end;;
```

If you want a comparison function that orders things in a particular way, you can always write your own by hand; but if all you need is a total order suitable for creating maps and sets with, then `[@@deriving compare]` is a good choice.

=, ==, and phys_equal

OCaml has multiple notions of equality, and picking the right one can be tricky. If you don't open `Base`, you'll find that the == operator tests for *physical* equality, while the = operator is the polymorphic equality function.

Two values are considered physically equal if they are the same pointer in memory. Two data structures that have identical contents but are constructed separately will not be considered equal by ==. Polymorphic equality, on the other hand, is *structural*, which effectively means that it considers values to be equal if they have the same contents.

Most of the time you don't want either of these forms of equality! Polymorphic equality is problematic for reasons we explained earlier in the chapter, and physical equality, while useful, is something that's needed in particular cases, most often when you're dealing with mutable objects, where the physical identity of the object matters.

`Base` hides polymorphic equality, instead reserving = for equality functions associated with particular types. At the top-level = is specialized to integers.

```
# 1 = 2;;
- : bool = false
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
# "one" = "two";;
Line 1, characters 1-6:
Error: This expression has type string but an expression was expected
    of type
         int
```

Other type-specific equality functions are found in their associated modules

```
# String.("one" = "two");;
- : bool = false
```

It's quite easy to mix up = and ==, and so `Base` deprecates == and provides `phys_equal` instead, a function with a clear and descriptive name.

```
# ref 1 == ref 1;;
Line 1, characters 7-9:
Alert deprecated: Base.==
[2016-09] this element comes from the stdlib distributed with OCaml.
Use [phys_equal] instead.
- : bool = false
# phys_equal (ref 1) (ref 1);;
- : bool = false
```

This is just a small way in which `Base` tries to avoid error-prone APIs.

15.1.6 Applying `[@@deriving]` to Maps and Sets

In the previous section, we showed how to use `[@@deriving]` annotations to set up a type so it could be used to create a map or set type. But what if we want to put a `[@@deriving]` annotation on a map or set type itself?

```
# type string_int_map =
    (string,int,String.comparator_witness) Map.t [@@deriving sexp];;
Line 2, characters 44-49:
Error: Unbound value Map.t_of_sexp
Hint: Did you mean m__t_of_sexp?
```

This fails because there is no existing `Map.t_of_sexp`. This isn't a simple omission; there's no reasonable way to define a useful `Map.t_of_sexp`, because a comparator witness isn't something that can be parsed out of the s-expression.

Happily, there's another way of writing the type of a map that does work with the various `[@@deriving]` extensions, which you can see below.

```
# type string_int_map = int Map.M(String).t [@@deriving sexp];;
type string_int_map = int Base.Map.M(Base.String).t
val string_int_map_of_sexp : Sexp.t -> string_int_map = <fun>
val sexp_of_string_int_map : string_int_map -> Sexp.t = <fun>
```

Here, we use a functor, `Map.M`, to define the type we need. While this looks different than the ordinary type signature, the meaning of the type is the same, as we can see below.

```
# let m = Map.singleton (module String) "one" 1;;
val m : (string, int, String.comparator_witness) Map.t = <abstr>
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
# (m : int Map.M(String).t);;
- : int Base.Map.M(Base.String).t = <abstr>
```

This same type works with other derivers as well, like those for comparison and hash functions. Since this way of writing the type is also shorter, it's what you should use most of the time.

15.1.7 Trees

As we've discussed, maps carry within them the comparator that they were created with. Sometimes, for space efficiency reasons, you want a version of the map data structure that doesn't include the comparator. You can get such a representation with `Map.Using_comparator.to_tree`, which returns just the tree underlying the map, without the comparator.

```
# let ord_tree = Map.Using_comparator.to_tree ord_map;;
val ord_tree :
  (string, int, String.comparator_witness)
    Map.Using_comparator.Tree.t =
  <abstr>
```

Even though the tree doesn't physically include a comparator, it does include the comparator in its type. This is what is known as a *phantom type*, because it reflects something about the logic of the value in question, even though it doesn't correspond to any values directly represented in the underlying physical structure of the value.

Since the comparator isn't included in the tree, we need to provide the comparator explicitly when we, say, search for a key, as shown below:

```
# Map.Using_comparator.Tree.find ~comparator:String.comparator
    ord_tree "snoo";;
- : int option = Some 3
```

The algorithm of `Map.Tree.find` depends on the fact that it's using the same comparator when looking up a value as you were when you stored it. That's the invariant that the phantom type is there to enforce. As you can see in the following example, using the wrong comparator will lead to a type error:

```
# Map.Using_comparator.Tree.find ~comparator:Reverse.comparator
    ord_tree "snoo";;
Line 1, characters 63-71:
Error: This expression has type
         (string, int, String.comparator_witness)
    Map.Using_comparator.Tree.t
       but an expression was expected of type
         (string, int, Reverse.comparator_witness)
         Map.Using_comparator.Tree.t
       Type String.comparator_witness is not compatible with type
         Reverse.comparator_witness
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

15.2 Hash Tables

Hash tables are the imperative cousin of maps. We walked through a basic hash table implementation in Chapter 9 (Imperative Programming), so in this section we'll mostly discuss the pragmatics of Core's `Hashtbl` module. We'll cover this material more briefly than we did with maps because many of the concepts are shared.

Hash tables differ from maps in a few key ways. First, hash tables are mutable, meaning that adding a key/value pair to a hash table modifies the table, rather than creating a new table with the binding added. Second, hash tables generally have better time-complexity than maps, providing constant-time lookup and modifications, as opposed to logarithmic for maps. And finally, just as maps depend on having a comparison function for creating the ordered binary tree that underlies a map, hash tables depend on having a *hash function*, i.e., a function for converting a key to an integer.

15.2.1 Time Complexity of Hash Tables

The statement that hash tables provide constant-time access hides some complexities. First of all, most hash table implementations, OCaml's included, need to resize the table when it gets too full. A resize requires allocating a new backing array for the hash table and copying over all entries, and so it is quite an expensive operation. That means adding a new element to the table is only *amortized* constant, which is to say, it's constant on average over a long sequence of operations, but some of the individual operations can cost more.

Another hidden cost of hash tables has to do with the hash function you use. If you end up with a pathologically bad hash function that hashes all of your data to the same number, then all of your insertions will hash to the same underlying bucket, meaning you no longer get constant-time access at all. `Base`'s hash table implementation uses binary trees for the hash-buckets, so this case only leads to logarithmic time, rather than linear for a traditional implementation.

The logarithmic behavior of Base's hash tables in the presence of hash collisions also helps protect against some denial-of-service attacks. One well-known type of attack is to send queries to a service with carefully chosen keys to cause many collisions. This, in combination with the linear behavior of most hashtables, can cause the service to become unresponsive due to high CPU load. Base's hash tables would be much less susceptible to such an attack because the amount of degradation would be far less.

We create a hashtable in a way that's similar to how we create maps, by providing a first-class module from which the required operations for building a hashtable can be obtained.

```
# let table = Hashtbl.create (module String);;
val table : (string, '_weak1) Base.Hashtbl.t = <abstr>
# Hashtbl.set table ~key:"three" ~data:3;;
- : unit = ()
# Hashtbl.find table "three";;
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
- : int option = Some 3
```

As with maps, most modules in Base are ready to be used for this purpose, but if you want to create a hash table from one of your own types, you need to do some work to prepare it. In order for a module to be suitable for passing to `Hashtbl.create`, it has to match the following interface.

```
# #show Base.Hashtbl.Key.S;;
module type S =
  sig
    type t
    val compare : t -> t -> int
    val sexp_of_t : t -> Sexp.t
    val hash : t -> int
  end
```

Note that there's no equivalent to the comparator witness that came up for maps and sets. That's because the requirement for multiple objects to share a comparison function or a hash function mostly just doesn't come up for hash tables. That makes building a module suitable for use with a hash table simpler.

```
# module Book = struct
    type t = { title: string; isbn: string }
    [@@deriving compare, sexp_of, hash]
  end;;
module Book :
  sig
    type t = { title : string; isbn : string; }
    val compare : t -> t -> int
    val sexp_of_t : t -> Sexp.t
    val hash_fold_t :
      Base_internalhash_types.state -> t ->
     Base_internalhash_types.state
    val hash : t -> int
  end
# let table = Hashtbl.create (module Book);;
val table : (Book.t, '_weak2) Base.Hashtbl.t = <abstr>
```

You can also create a hashtable based on OCaml's polymorphic hash and comparison functions.

```
# let table = Hashtbl.Poly.create ();;
val table : ('_weak3, '_weak4) Base.Hashtbl.t = <abstr>
# Hashtbl.set table ~key:("foo",3,[1;2;3]) ~data:"random data!";;
- : unit = ()
# Hashtbl.find table ("foo",3,[1;2;3]);;
- : string option = Some "random data!"
```

This is highly convenient, but polymorphic comparison can behave in surprising ways, so it's generally best to avoid this for code where correctness matters.

15.2.2 Collisions with the Polymorphic Hash Function

The polymorphic hash function, like polymorphic compare, has problems that derive from the fact that it doesn't pay any attention to the type, just blindly walking down

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

the structure of a data type and computing a hash from what it sees. That means that for data structures like maps and sets where equivalent instances can have different structures, it will do the wrong thing.

But there's another problem with polymorphic hash, which is that it is prone to creating hash collisions. OCaml's polymorphic hash function works by walking over the data structure it's given using a breadth-first traversal that is bounded in the number of nodes it's willing to traverse. By default, that bound is set at 10 "meaningful" nodes.

The bound on the traversal means that the hash function may ignore part of the data structure, and this can lead to pathological cases where every value you store has the same hash value. We'll demonstrate this below, using the function `List.range` to allocate lists of integers of different length:

```
# Hashtbl.Poly.hashable.hash (List.range 0 9);;
- : int = 209331808
# Hashtbl.Poly.hashable.hash (List.range 0 10);;
- : int = 182325193
# Hashtbl.Poly.hashable.hash (List.range 0 11);;
- : int = 182325193
# Hashtbl.Poly.hashable.hash (List.range 0 100);;
- : int = 182325193
```

As you can see, the hash function stops after the first 10 elements. The same can happen with any large data structure, including records and arrays. When building hash functions over large custom data structures, it is generally a good idea to write one's own hash function, or to use the ones provided by `[@@deriving]`, which don't have this problem, as you can see below.

```
# [%hash: int list] (List.range 0 9);;
- : int = 999007935
# [%hash: int list] (List.range 0 10);;
- : int = 195154657
# [%hash: int list] (List.range 0 11);;
- : int = 527899773
# [%hash: int list] (List.range 0 100);;
- : int = 594983280
```

Note that rather than declaring a type and using `[@@deriving hash]` to invoke ppx_hash, we use `[%hash]`, a shorthand for creating a hash function inline in an expression.

15.3 Choosing Between Maps and Hash Tables

Maps and hash tables overlap enough in functionality that it's not always clear when to choose one or the other. Maps, by virtue of being immutable, are generally the default choice in OCaml. OCaml also has good support for imperative programming, though, and when programming in an imperative idiom, hash tables are often the more natural choice.

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

Programming idioms aside, there are significant performance differences between maps and hash tables. For code that is dominated by updates and lookups, hash tables are a clear performance win, and the win is clearer the larger the amount of data.

The best way of answering a performance question is by running a benchmark, so let's do just that. The following benchmark uses the `core_bench` library, and it compares maps and hash tables under a very simple workload. Here, we're keeping track of a set of 1,000 different integer keys and cycling over the keys and updating the values they contain. Note that we use the `Map.change` and `Hashtbl.change` functions to update the respective data structures:

```
open Base
open Core_bench

let map_iter ~num_keys ~iterations =
  let rec loop i map =
    if i <= 0
    then ()
    else
      loop
        (i - 1)
        (Map.change map (i % num_keys) ~f:(fun current ->
             Some (1 + Option.value ~default:0 current)))
  in
  loop iterations (Map.empty (module Int))

let table_iter ~num_keys ~iterations =
  let table = Hashtbl.create (module Int) ~size:num_keys in
  let rec loop i =
    if i <= 0
    then ()
    else (
      Hashtbl.change table (i % num_keys) ~f:(fun current ->
          Some (1 + Option.value ~default:0 current));
      loop (i - 1))
  in
  loop iterations

let tests ~num_keys ~iterations =
  let t name f = Bench.Test.create f ~name in
  [ t "table" (fun () -> table_iter ~num_keys ~iterations)
  ; t "map" (fun () -> map_iter ~num_keys ~iterations)
  ]

let () =
  tests ~num_keys:1000 ~iterations:100_000
  |> Bench.make_command
  |> Core.Command.run
```

The results show the hash table version to be around four times faster than the map version:

```
(executable
  (name      map_vs_hash)
  (libraries base core_bench))
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
$ dune build map_vs_hash.exe
$ ./_build/default/map_vs_hash.exe -ascii -quota 1 -clear-columns
    time speedup
Estimated testing time 2s (2 benchmarks x 1s). Change using -quota
    SECS.

  Name    Time/Run   Speedup
 ------- ---------- ---------
  table    13.34ms      1.00
  map      44.54ms      3.34
```

We can make the speedup smaller or larger depending on the details of the test; for example, it will vary with the number of distinct keys. But overall, for code that is heavy on sequences of querying and updating a set of key/value pairs, hash tables will significantly outperform maps.

Hash tables are not always the faster choice, though. In particular, maps excel in situations where you need to keep multiple related versions of the data structure in memory at once. That's because maps are immutable, and so operations like `Map.add` that modify a map do so by creating a new map, leaving the original undisturbed. Moreover, the new and old maps share most of their physical structure, so keeping multiple versions around can be space-efficient.

Here's a benchmark that demonstrates this. In it, we create a list of maps (or hash tables) that are built up by iteratively applying small updates, keeping these copies around. In the map case, this is done by using `Map.change` to update the map. In the hash table implementation, the updates are done using `Hashtbl.change`, but we also need to call `Hashtbl.copy` to take snapshots of the table:

```
open Base
open Core_bench

let create_maps ~num_keys ~iterations =
  let rec loop i map =
    if i <= 0
    then []
    else (
      let new_map =
        Map.change map (i % num_keys) ~f:(fun current ->
            Some (1 + Option.value ~default:0 current))
      in
      new_map :: loop (i - 1) new_map)
  in
  loop iterations (Map.empty (module Int))

let create_tables ~num_keys ~iterations =
  let table = Hashtbl.create (module Int) ~size:num_keys in
  let rec loop i =
    if i <= 0
    then []
    else (
      Hashtbl.change table (i % num_keys) ~f:(fun current ->
          Some (1 + Option.value ~default:0 current));
      let new_table = Hashtbl.copy table in
      new_table :: loop (i - 1))
```

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press

```
  in
  loop iterations

let tests ~num_keys ~iterations =
  let t name f = Bench.Test.create f ~name in
  [ t "table" (fun () -> ignore (create_tables ~num_keys ~iterations))
  ; t "map" (fun () -> ignore (create_maps ~num_keys ~iterations))
  ]

let () =
  tests ~num_keys:50 ~iterations:1000
  |> Bench.make_command
  |> Core.Command.run
```

Unsurprisingly, maps perform far better than hash tables on this benchmark, in this case by more than a factor of 10:

```
(executable
  (name      map_vs_hash2)
  (libraries core_bench))
```

```
$ dune build map_vs_hash2.exe
$ ./_build/default/map_vs_hash2.exe -ascii -clear-columns time speedup
Estimated testing time 20s (2 benchmarks x 10s). Change using -quota
    SECS.

  Name      Time/Run   Speedup
 ------- ------------ ---------
  table   4_453.95us     25.80
  map       172.61us      1.00
```

These numbers can be made more extreme by increasing the size of the tables or the length of the list.

As you can see, the relative performance of trees and maps depends a great deal on the details of how they're used, and so whether to choose one data structure or the other will depend on the details of the application.

https://doi.org/10.1017/9781009129220.018 Published online by Cambridge University Press