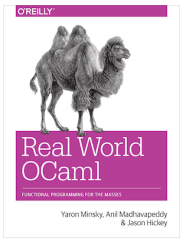


Table of Contents

Chapter 13. 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:

```
# 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")]
```

OCaml Utop ∗ maps-and-hash-tables/main.topscript , continued (part 1) ∗ all code

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

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

OCaml Utop ∗ maps-and-hash-tables/main.topscript , continued (part 2) ∗ all code

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.

MAPS

Let's consider an example of how one might use a map in practice. In Chapter 4, *Files, Modules, and Programs*, we showed a module `Counter` for keeping frequency counts on a set of strings. Here's the interface:

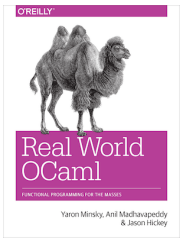
```
open Core.Std

(** 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 strings
   in the list will show up at most once, and the integers will be at least
   1. *)
val to_list : t -> (string * int) list
```

OCaml ∗ files-modules-and-programs-freq-fast/counter.mli ∗ all code

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 Core.Std

type t = int String.Map.t
```




```
let empty = String.Map.empty

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.add t ~key:s ~data:(count + 1)
```

OCaml ∗ files-modules-and-programs-freq-fast/counter.ml ∗ all code

Note that in some places the preceding code refers to `String.Map.t`, and in others `Map.t`. This has to do with the fact that maps are implemented as ordered binary trees, and as such, need a way of comparing keys.

To deal with this, a map, once created, stores the necessary comparison function within the data structure. Thus, operations like `Map.find` or `Map.add` that access the contents of a map or create a new map from an existing one, do so by using the comparison function embedded within the map.

But in order to get a map in the first place, you need to get your hands on the comparison function somehow. For this reason, modules like `String` contain a `Map` submodule that has values like `String.Map.empty` and `String.Map.of_alist` that are specialized to strings, and thus have access to a string comparison function. Such a `Map` submodule is included in every module that satisfies the `Comparable.S` interface from Core.

Creating Maps with Comparators

The specialized `Map` submodule is convenient, but it's not the only way of creating a `Map.t`. The information required to compare values of a given type is wrapped up in a value called a *comparator* that can be used to create maps using the `Map` module directly:

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

OCaml Utop ∗ maps-and-hash-tables/main.topscript , continued (part 3) ∗ all code

The preceding code uses `Map.of_alist_exn`, which creates a map from an association list, throwing an exception if there are duplicate keys in the list.

The comparator is only required for operations that create maps from scratch. Operations that update an existing map simply inherit the comparator of the map they start with:

```
# let zilch_map = Map.add digit_map ~key:0 ~data:"zilch";;
 val zilch_map : (int, string, Int.comparator) Map.t = <abstr>
```

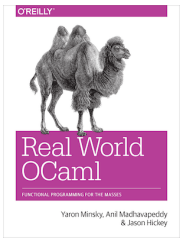
OCaml Utop ∗ maps-and-hash-tables/main.topscript , continued (part 4) ∗ all code

The type `Map.t` has three type parameters: one for the key, one for the value, and one to identify the comparator. Indeed, the type `'a Int.Map.t` is just a type alias for `(int,'a,Int.comparator) Map.t`.

Including the comparator in the type is important because operations that work on multiple maps at the same time often require that the maps share their comparison function. Consider, for example, `Map.symmetric_diff`, which computes a summary of the differences between two maps:

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

OCaml Utop ∗ maps-and-hash-tables/main.topscript , continued (part 5) ∗ all code



The type of `Map.symmetric_diff`, which follows, requires that the two maps it compares have the same comparator type. Each comparator has a fresh abstract type, so the type of a comparator identifies the comparator uniquely:

```
# Map.symmetric_diff;;
 - : ('k, 'v, 'cmp) Map.t ->
     ('k, 'v, 'cmp) Map.t ->
     data_equal:('v -> 'v -> bool) ->
     ('k * [ `Left of 'v | `Right of 'v | `Unequal of 'v * 'v ]) list
 = <fun>
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 6) ⁎ all code

This constraint is important because the algorithm that `Map.symmetric_diff` uses depends for its correctness on the fact that both maps have the same comparator.

We can create a new comparator using the `Comparator.Make` functor, which takes as its input a module containing the type of the object to be compared, sexp converter functions, and a comparison function. The sexp converters are included in the comparator to make it possible for users of the comparator to generate better error messages. Here's an example:

```
# module Reverse = Comparator.Make(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);;
 module Reverse :
   sig
     type t = string
     val compare : t -> t -> int
     val t_of_sexp : Sexp.t -> t
     val sexp_of_t : t -> Sexp.t
     type comparator
     val comparator : (t, comparator) Comparator.t_
   end
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 7) ⁎ all code

As you can see in the following code, both `Reverse.comparator` and `String.comparator` 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 ~comparator:String.comparator alist;;
 val ord_map : (string, int, String.comparator) Map.t = <abstr>
# let rev_map = Map.of_alist_exn ~comparator:Reverse.comparator alist;;
 val rev_map : (string, int, Reverse.comparator) Map.t = <abstr>
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 8) ⁎ all code

`Map.min_elt` returns the key and value for the smallest key in the map, which lets us see 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)
```

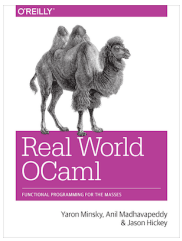
OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 9) ⁎ all code

Accordingly, if we try to use `Map.symmetric_diff` on these two maps, we'll get a compile-time error:

```
# Map.symmetric_diff ord_map rev_map;;
 Characters 27-34:
 Error: This expression has type (string, int, Reverse.comparator) Map.t
        but an expression was expected of type
          (string, int, String.comparator) Map.t
        Type Reverse.comparator is not compatible with type String.comparator
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 10) ⁎ all code

Trees



Buy in print and eBook.

As we've discussed, maps carry within them the comparator that they were created with. Sometimes, often 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.to_tree`, which returns just the tree underlying the map, without the comparator:

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

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 11) ⁎ all code

Even though a `Map.Tree.t` 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.Tree.find ~comparator:String.comparator ord_tree "snoo";;
 - : int option = Some 3
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 12) ⁎ all code

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.Tree.find ~comparator:Reverse.comparator ord_tree "snoo";;
 Characters 45-53:
 Error: This expression has type (string, int, String.comparator) Map.Tree.t
        but an expression was expected of type
          (string, int, Reverse.comparator) Map.Tree.t
        Type String.comparator is not compatible with type Reverse.comparator
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 13) ⁎ all code

The Polymorphic Comparator

We don't need to generate specialized comparators for every type we want to build a map on. We can instead use a comparator based on OCaml's built-in polymorphic comparison function, which was discussed in Chapter 3, *Lists and Patterns*. This comparator is found in the `Comparator.Poly` module, allowing us to write:

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

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 14) ⁎ all code

Or, equivalently:

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

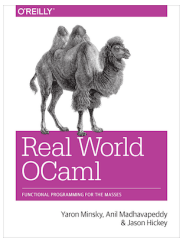
OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 15) ⁎ all code

Note that maps based on the polymorphic comparator are not equivalent to those based on the type-specific comparators from the point of view of the type system. Thus, the compiler rejects the following:

```
# Map.symmetric_diff (Map.Poly.singleton 3 "three")
                     (Int.Map.singleton  3 "four" ) ;;
 Characters 72-99:
 Error: This expression has type
          string Int.Map.t = (int, string, Int.comparator) Map.t
        but an expression was expected of type
          (int, string, Comparator.Poly.comparator) Map.t
        Type Int.comparator is not compatible with type
          Comparator.Poly.comparator
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 16) ⁎ all code

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 highly convenient, but it has serious downsides as well and should be used with care. In particular, polymorphic compare has a fixed algorithm for comparing values of any type, and that algorithm can sometimes yield surprising results.

To understand what's wrong with polymorphic compare, you need to understand a bit about how it works. Polymorphic compare is *structural*, in that it operates directly on the runtime representation of OCaml values, walking the structure of the values in question without regard for their type.

This is convenient because it provides a comparison function that works for most OCaml values and largely behaves as you would expect. For example, on `int`s and `float`s, it acts as you would expect a numeric comparison function to act. For simple containers like strings and lists and arrays, it operates as a lexicographic comparison. And except for functions and values from outside of the OCaml heap, it works on almost every OCaml type.

But sometimes, a structural comparison is not what you want. Sets are a great example of this. Consider the following two sets:

```
# let (s1,s2) = (Int.Set.of_list [1;2],
                 Int.Set.of_list [2;1]);;
 val s1 : Int.Set.t = <abstr>
 val s2 : Int.Set.t = <abstr>
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 18) ⁎ all code

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

```
# Set.equal s1 s2;;
 - : bool = true
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 19) ⁎ all code

But because the elements were added in different orders, the layout of the trees underlying the sets will be different. As such, a structural comparison function will conclude that they're different.

Let's see what happens if we use polymorphic compare to test for equality by way of the = operator. Comparing the maps directly will fail at runtime because the comparators stored within the sets contain function values:

```
# s1 = s2;;
 Exception: (Invalid_argument "equal: functional value").
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 20) ⁎ all code

We can, however, use the function `Set.to_tree` to expose the underlying tree without the attached comparator:

```
# Set.to_tree s1 = Set.to_tree s2;;
 - : bool = false
```

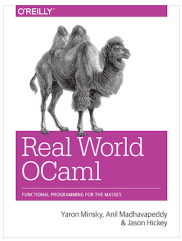
OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 21) ⁎ all code

This can cause real and quite subtle bugs. If, for example, you use a map whose keys contain sets, then the map built with the polymorphic comparator will behave incorrectly, separating out keys that should be aggregated together. Even worse, it will work sometimes and fail others; since if the sets are built in a consistent order, then they will work as expected, but once the order changes, the behavior will change.

Sets

Sometimes, instead of keeping track of a set of key/value pairs, you just want a data type for keeping track of a set of keys. You could build this on top of a map by representing a set of values by a map whose data type is `unit`. But a more idiomatic (and efficient) solution is to use Core's set type, which is similar in design and spirit to the map type, while having an API better tuned to working with sets and a lower memory footprint. Here's a simple example:

```
# let dedup ~comparator l =
    List.fold l ~init:(Set.empty ~comparator) ~f:Set.add
    |> Set.to_list
  ;;
```




```
 val dedup :
   comparator:('a, 'b) Core_kernel.Comparator.t_ -> 'a list -> 'a list = <fun>
# dedup ~comparator:Int.comparator [8;3;2;3;7;8;10];;
 - : int list = [2; 3; 7; 8; 10]
```

OCaml Utop ∗ maps-and-hash-tables/main.topscript , continued (part 17) ∗ all code

In addition to the operators you would expect to have for maps, sets support the traditional set operations, including union, intersection, and set difference. And, as with maps, we can create sets based on type-specific comparators or on the polymorphic comparator.

Satisfying the Comparable.S Interface

Core's `Comparable.S` interface includes a lot of useful functionality, including support for working with maps and sets. In particular, `Comparable.S` requires the presence of the `Map` and `Set` submodules, as well as a comparator.

`Comparable.S` is satisfied by most of the types in Core, but the question arises of how to satisfy the comparable interface for a new type that you design. Certainly implementing all of the required functionality from scratch would be an absurd amount of work.

The module `Comparable` contains a number of functors to help you automate this task. The simplest one of these is `Comparable.Make`, which takes as an input any module that satisfies the following interface:

```
module type Comparable = sig
  type t
  val sexp_of_t : t -> Sexp.t
  val t_of_sexp : Sexp.t -> t
  val compare : t -> t -> int
end
```

OCaml ∗ maps-and-hash-tables/comparable.ml ∗ all code

In other words, it expects a type with a comparison function, as well as functions for converting to and from *s-expressions*. S-expressions are a serialization format used commonly in Core and are required here to enable better error messages. We'll discuss s-expressions more in Chapter 17, *Data Serialization with S-Expressions*, but in the meantime, we'll use the `with sexp` declaration that comes from the Sexplib syntax extension. This declaration kicks off the automatic generation of s-expression conversion functions for the marked type.

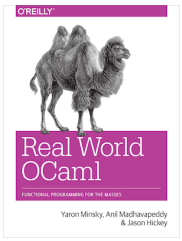
The following example shows how this all fits together, following the same basic pattern for using functors described in the section called "Extending Modules":

```
# module Foo_and_bar : sig
    type t = { foo: Int.Set.t; bar: string }
    include Comparable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: Int.Set.t; bar: string } with sexp
      let compare t1 t2 =
        let c = Int.Set.compare t1.foo t2.foo in
        if c <> 0 then c else String.compare t1.bar t2.bar
    end
    include T
    include Comparable.Make(T)
  end;;
 module Foo_and_bar :
 sig
   type t = { foo : Int.Set.t; bar : string; }
   val ( >= ) : t -> t -> bool
   val ( <= ) : t -> t -> bool
   val ( = ) : t -> t -> bool

     ...

 end
```

OCaml Utop ∗ maps-and-hash-tables/main-22.rawscript ∗ all code

We don't include the full response from the toplevel because it is quite lengthy, but `Foo_and_bar` does satisfy `Comparable.S`.

In the preceding code we wrote the comparison function by hand, but this isn't strictly necessary. Core ships with a syntax extension called `comparelib`, which will create a comparison function from a type definition. Using it, we can rewrite the previous example as follows:




```
# module Foo_and_bar : sig
    type t = { foo: Int.Set.t; bar: string }
    include Comparable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: Int.Set.t; bar: string } with sexp, compare
    end
    include T
    include Comparable.Make(T)
  end;;
module Foo_and_bar :
sig
  type t = { foo : Int.Set.t; bar : string; }
  val ( >= ) : t -> t -> bool
  val ( <= ) : t -> t -> bool
  val ( = ) : t -> t -> bool

  ...

end
```

OCaml Utop ⋆ maps-and-hash-tables/main-23.rawscript ⋆ all code

The comparison function created by `comparelib` for a given type will call out to the comparison functions for its component types. As a result, the `foo` field will be compared using `Int.Set.compare`. This is different, and saner than the structural comparison done by polymorphic compare.

If you want your comparison function to behave in a specific way, you should still write your own comparison function by hand; but if all you want is a total order suitable for creating maps and sets with, then `comparelib` is a good way to go.

You can also satisfy the `Comparable.S` interface using polymorphic compare:

```
# module Foo_and_bar : sig
    type t = { foo: int; bar: string }
    include Comparable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: int; bar: string } with sexp
    end
    include T
    include Comparable.Poly(T)
  end;;
module Foo_and_bar :
sig
  type t = { foo : int; bar : string; }
  val ( >= ) : t -> t -> bool
  val ( <= ) : t -> t -> bool
  val ( = ) : t -> t -> bool

  ...

end
```

OCaml Utop ⋆ maps-and-hash-tables/main-24.rawscript ⋆ all code

That said, for reasons we discussed earlier, polymorphic compare should be used sparingly.

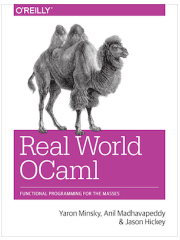
=, ==, and phys_equal

If you come from a C/C++ background, you'll probably reflexively use == to test two values for equality. In OCaml, the == operator tests for *physical* equality, while the = operator tests for *structural* equality.

The physical equality test will match if two data structures have precisely the same pointer in memory. Two data structures that have identical contents but are constructed separately will not match using ==.

The = structural equality operator recursively inspects each field in the two values and tests them individually for equality. Crucially, if your data structure is cyclical (that is, a value recursively points back to another field within the same structure), the = operator will never terminate, and your program will hang! You therefore must use the physical equality operator or write a custom comparison function when comparing cyclic values.

It's quite easy to mix up the use of = and ==, so Core disables the == operator and provides the more explicit `phys_equal` function instead. You'll see a type error if you use == anywhere in code



that opens `Core.Std`:

```
# open Core.Std ;;

# 1 == 2 ;;
 Characters -1-1:
 Error: This expression has type int but an expression was expected of type
          [ `Consider_using_phys_equal ]
# phys_equal 1 2 ;;
 - : bool = false
```

OCaml Utop ⁎ maps-and-hash-tables/core_phys_equal.topscript ⁎ all code

If you feel like hanging your OCaml interpreter, you can verify what happens with recursive values and structural equality for yourself:

```
# type t1 = { foo1:int; bar1:t2 } and t2 = { foo2:int; bar2:t1 } ;;
 type t1 = { foo1 : int; bar1 : t2; }
 and t2 = { foo2 : int; bar2 : t1; }
# let rec v1 = { foo1=1; bar1=v2 } and v2 = { foo2=2; bar2=v1 } ;;
 <lots of text>
# v1 == v1;;
 - : bool = true
# phys_equal v1 v1;;
 - : bool = true
# v1 = v1 ;;
 <press ^Z and kill the process now>
```

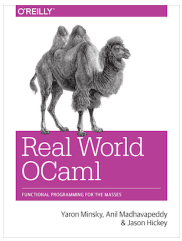
OCaml Utop ⁎ maps-and-hash-tables/phys_equal.rawscript ⁎ all code

HASH TABLES

Hash tables are the imperative cousin of maps. We walked over a basic hash table implementation in Chapter 8, *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.

> **Time Complexity of Hash Tables**
>
> The statement that hash tables provide constant-time access hides some complexities. First of all, any hash table implementation, OCaml's included, needs 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 be quite expensive.
>
> 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. Core'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 hash table.
>
> The logarithmic behavior of Core'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. Core's hash tables would be much less susceptible to such an attack because the amount of degradation would be far less.



When creating a hash table, we need to provide a value of type *hashable*, which includes among other things the function for hashing the key type. This is analogous to the comparator used for creating maps:

```
# let table = Hashtbl.create ~hashable:String.hashable ();;
 val table : (string, '_a) Hashtbl.t = <abstr>
# Hashtbl.replace table ~key:"three" ~data:3;;
 - : unit = ()
# Hashtbl.find table "three";;
 - : int option = Some 3
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 25) ⁎ all code

The `hashable` value is included as part of the `Hashable.S` interface, which is satisfied by most types in Core. The `Hashable.S` interface also includes a `Table` submodule which provides more convenient creation functions:

```
# let table = String.Table.create ();;
 val table : '_a String.Table.t = <abstr>
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 26) ⁎ all code

There is also a polymorphic `hashable` value, corresponding to the polymorphic hash function provided by the OCaml runtime, for cases where you don't have a hash function for your specific type:

```
# let table = Hashtbl.create ~hashable:Hashtbl.Poly.hashable ();;
 val table : ('_a, '_b) Hashtbl.t = <abstr>
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 27) ⁎ all code

Or, equivalently:

```
# let table = Hashtbl.Poly.create ();;
 val table : ('_a, '_b) Hashtbl.t = <abstr>
```

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 28) ⁎ all code

Note that, unlike the comparators used with maps and sets, hashables don't show up in the type of a `Hashtbl.t`. That's because hash tables don't have operations that operate on multiple hash tables that depend on those tables having the same hash function, in the way that `Map.symmetric_diff` and `Set.union` depend on their arguments using the same comparison function.

Collisions with the Polymorphic Hash Function

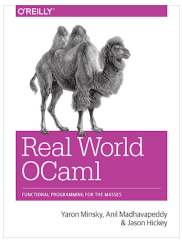
OCaml's polymorphic hash function works by walking over the data structure it's given using a breadth-first traversal that is bounded in the number of nodes it's willing to traverse. By default, that bound is set at 10 "meaningful" nodes.

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:

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

OCaml Utop ⁎ maps-and-hash-tables/main.topscript , continued (part 29) ⁎ all code

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.



Satisfying the Hashable.S Interface

Most types in Core satisfy the `Hashable.S` interface, but as with the `Comparable.S` interface, the question remains of how one should satisfy this interface when writing a new module. Again, the answer is to use a functor to build the necessary functionality; in this case, `Hashable.Make`. Note that we use OCaml's `lxor` operator for doing the "logical" (i.e., bitwise) exclusive Or of the hashes from the component values:

```
# module Foo_and_bar : sig
    type t = { foo: int; bar: string }
    include Hashable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: int; bar: string } with sexp, compare
      let hash t =
        (Int.hash t.foo) lxor (String.hash t.bar)
    end
    include T
    include Hashable.Make(T)
  end;;
module Foo_and_bar :
sig
  type t = { foo : int; bar : string; }
  module Hashable : sig type t = t end
  val hash : t -> int
  val compare : t -> t -> int
  val hashable : t Pooled_hashtbl.Hashable.t

  ...

end
```

OCaml Utop ∗ maps-and-hash-tables/main-30.rawscript ∗ all code

Note that in order to satisfy hashable, one also needs to provide a comparison function. That's because Core's hash tables use an ordered binary tree data structure for the hash-buckets, so that performance of the table degrades gracefully in the case of pathologically bad choice of hash function.

There is currently no analogue of `comparelib` for autogeneration of hash functions, so you do need to either write the hash function by hand, or use the built-in polymorphic hash function, `Hashtbl.hash`.

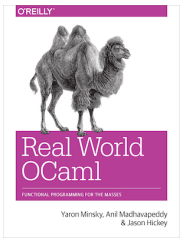
CHOOSING BETWEEN MAPS AND HASH TABLES

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.

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 Core.Std
open Core_bench.Std

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

let table_iter ~num_keys ~iterations =
  let table = Int.Table.create ~size:num_keys () in
  let rec loop i =
    if i <= 0 then ()
```




```
    else (
      Hashtbl.change table (i mod num_keys) (fun current ->
        Some (1 + Option.value ~default:0 current));
      loop (i - 1)
    )
  in
  loop iterations

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

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

OCaml ∗ maps-and-hash-tables/map_vs_hash.ml ∗ all code

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

```
$ corebuild -pkg core_bench map_vs_hash.native
$ ./map_vs_hash.native -ascii -clear-columns name time speedup
 Estimated testing time 20s (change using -quota SECS).

   Name       Time/Run   Speedup
  ------- ------------ ---------
   map     20_234_582      1.00
   table    4_429_771      4.57
```

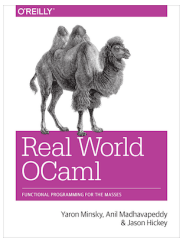
Terminal ∗ maps-and-hash-tables/run_map_vs_hash.out ∗ all code

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 are often more performant 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 multiple versions can be kept around efficiently.

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 Core.Std
open Core_bench.Std

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

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




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

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

OCaml ∗ maps-and-hash-tables/map_vs_hash2.ml ∗ all code

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

```
$ corebuild -pkg core_bench map_vs_hash2.native
$ ./map_vs_hash2.native -ascii -clear-columns name time speedup
 Estimated testing time 20s (change using -quota SECS).

  Name     Time/Run   Speedup
 ------- ----------- ---------
  map       147_208     11.28
  table   1_660_635      1.00
```

Terminal ∗ maps-and-hash-tables/run_map_vs_hash2.out ∗ all code

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.

< Previous Next >