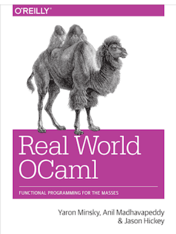


Buy in print and eBook.

Table of Contents

Chapter 5. Records

One of OCaml's best features is its concise and expressive system for declaring new data types, and records are a key element of that system. We discussed records briefly in Chapter 1, *A Guided Tour*, but this chapter will go into more depth, covering the details of how records work, as well as advice on how to use them effectively in your software designs.

A record represents a collection of values stored together as one, where each component is identified by a different field name. The basic syntax for a record type declaration is as follows:

```
type <record-name> =
  { <field> : <type> ;
    <field> : <type> ;
    ...
  }
```

Syntax ∗ records/record.syntax ∗ all code

Note that record field names must start with a lowercase letter.

Here's a simple example, a `host_info` record that summarizes information about a given computer:

```
# type host_info =
    { hostname   : string;
      os_name    : string;
      cpu_arch   : string;
      timestamp  : Time.t;
    };;
 type host_info = {
   hostname : string;
   os_name : string;
   cpu_arch : string;
   timestamp : Time.t;
 }
```

OCaml Utop ∗ records/main.topscript ∗ all code

We can construct a `host_info` just as easily. The following code uses the `Shell` module from `Core_extended` to dispatch commands to the shell to extract the information we need about the computer we're running on. It also uses the `Time.now` call from Core's `Time` module:

```
# #require "core_extended";;

# open Core_extended.Std;;

# let my_host =
    let sh = Shell.sh_one_exn in
    { hostname   = sh "hostname";
      os_name    = sh "uname -s";
      cpu_arch   = sh "uname -p";
      timestamp  = Time.now ();
    };;
 val my_host : host_info =
   {hostname = "flick.local"; os_name = "Darwin"; cpu_arch = "i386";
    timestamp = 2013-11-05 08:49:38.850439-05:00}
```

OCaml Utop ∗ records/main.topscript , continued (part 1) ∗ all code

You might wonder how the compiler inferred that `my_host` is of type `host_info`. The hook that the compiler uses in this case to figure out the type is the record field name. Later in the chapter, we'll talk about what happens when there is more than one record type in scope with the same field name.

Once we have a record value in hand, we can extract elements from the record field using dot notation:

```
# my_host.cpu_arch;;
 - : string = "i386"
```

OCaml Utop ∗ records/main.topscript , continued (part 2) ∗ all code

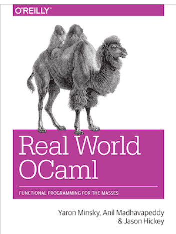


Table of Contents

When declaring an OCaml type, you always have the option of parameterizing it by a polymorphic type. Records are no different in this regard. So, for example, here's a type one might use to timestamp arbitrary items:

```
# type 'a timestamped = { item: 'a; time: Time.t };;
 type 'a timestamped = { item : 'a; time : Time.t; }
```

OCaml Utop ∗ records/main.topscript , continued (part 3) ∗ all code

We can then write polymorphic functions that operate over this parameterized type:

```
# let first_timestamped list =
    List.reduce list ~f:(fun a b -> if a.time < b.time then a else b)
  ;;
 val first_timestamped : 'a timestamped list -> 'a timestamped option = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 4) ∗ all code

PATTERNS AND EXHAUSTIVENESS

Another way of getting information out of a record is by using a pattern match, as in the definition of `host_info_to_string`:

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts;
                          } =
       sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
 val host_info_to_string : host_info -> string = <fun>
# host_info_to_string my_host;;
 - : string = "flick.local (Darwin / i386, on 2013-11-05 08:49:38)"
```

OCaml Utop ∗ records/main.topscript , continued (part 5) ∗ all code

Note that the pattern we used had only a single case, rather than using several cases separated by `|`'s. We needed only one pattern because record patterns are *irrefutable*, meaning that a record pattern match will never fail at runtime. This makes sense, because the set of fields available in a record is always the same. In general, patterns for types with a fixed structure, like records and tuples, are irrefutable, unlike types with variable structures like lists and variants.

Another important characteristic of record patterns is that they don't need to be complete; a pattern can mention only a subset of the fields in the record. This can be convenient, but it can also be error prone. In particular, this means that when new fields are added to the record, code that should be updated to react to the presence of those new fields will not be flagged by the compiler.

As an example, imagine that we wanted to add a new field to our `host_info` record called `os_release`:

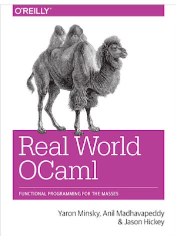
```
# type host_info =
    { hostname   : string;
      os_name    : string;
      cpu_arch   : string;
      os_release : string;
      timestamp  : Time.t;
    } ;;
 type host_info = {
   hostname : string;
   os_name : string;
   cpu_arch : string;
   os_release : string;
   timestamp : Time.t;
 }
```

OCaml Utop ∗ records/main.topscript , continued (part 6) ∗ all code

The code for `host_info_to_string` would continue to compile without change. In this particular case, it's pretty clear that you might want to update `host_info_to_string` in order to include `os_release`, and it would be nice if the type system would give you a warning about the change.

Happily, OCaml does offer an optional warning for missing fields in a record pattern. With that warning turned on (which you can do in the toplevel by typing `#warnings "+9"`), the compiler will warn about the missing field:

```
# #warnings "+9";;
```



Buy in print and eBook.

Login with GitHub to view and add comments

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts;
                          } =
    sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;

 Characters 24-139:
 Warning 9: the following labels are not bound in this record pattern:
 os_release
 Either bind these labels explicitly or add '; _' to the pattern.val host_info_to_st
```

OCaml Utop ∗ records/main.topscript , continued (part 7) ∗ all code

We can disable the warning for a given pattern by explicitly acknowledging that we are ignoring extra fields. This is done by adding an underscore to the pattern:

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts; _
                          } =
    sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
 val host_info_to_string : host_info -> string = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 8) ∗ all code

It's a good idea to enable the warning for incomplete record matches and to explicitly disable it with an _ where necessary.

Compiler Warnings

The OCaml compiler is packed full of useful warnings that can be enabled and disabled separately. These are documented in the compiler itself, so we could have found out about warning 9 as follows:

```
$ ocaml -warn-help | egrep '\b9\b'
   9 Missing fields in a record pattern.
   R Synonym for warning 9.
```

Terminal ∗ records/warn_help.out ∗ all code

You should think of OCaml's warnings as a powerful set of optional static analysis tools, and you should eagerly enable them in your build environment. You don't typically enable all warnings, but the defaults that ship with the compiler are pretty good.

The warnings used for building the examples in this book are specified with the following flag: `-w @A-4-33-41-42-43-34-44`.

The syntax of this can be found by running `ocaml -help`, but this particular invocation turns on all warnings as errors, disabling only the numbers listed explicitly after the `A`.

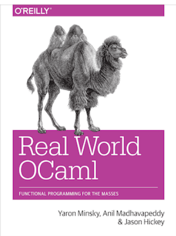
Treating warnings as errors (i.e., making OCaml fail to compile any code that triggers a warning) is good practice, since without it, warnings are too often ignored during development. When preparing a package for distribution, however, this is a bad idea, since the list of warnings may grow from one release of the compiler to another, and so this may lead your package to fail to compile on newer compiler releases.

FIELD PUNNING

When the name of a variable coincides with the name of a record field, OCaml provides some handy syntactic shortcuts. For example, the pattern in the following function binds all of the fields in question to variables of the same name. This is called *field punning*:

```
# let host_info_to_string { hostname; os_name; cpu_arch; timestamp; _ } =
     sprintf "%s (%s / %s) <%s>" hostname os_name cpu_arch
       (Time.to_string timestamp);;
 val host_info_to_string : host_info -> string = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 9) ∗ all code



Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

Field punning can also be used to construct a record. Consider the following code for generating a `host_info` record:

```
# let my_host =
    let sh cmd = Shell.sh_one_exn cmd in
    let hostname   = sh "hostname" in
    let os_name    = sh "uname -s" in
    let cpu_arch   = sh "uname -p" in
    let os_release = sh "uname -r" in
    let timestamp  = Time.now () in
    { hostname; os_name; cpu_arch; os_release; timestamp };;
 val my_host : host_info =
   {hostname = "flick.local"; os_name = "Darwin"; cpu_arch = "i386";
    os_release = "13.0.0"; timestamp = 2013-11-05 08:49:41.499579-05:00}
```

OCaml Utop ∗ records/main.topscript , continued (part 10) ∗ all code

In the preceding code, we defined variables corresponding to the record fields first, and then the record declaration itself simply listed the fields that needed to be included.

You can take advantage of both field punning and label punning when writing a function for constructing a record from labeled arguments:

```
# let create_host_info ~hostname ~os_name ~cpu_arch ~os_release =
    { os_name; cpu_arch; os_release;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
 val create_host_info :
   hostname:string ->
   os_name:string -> cpu_arch:string -> os_release:string -> host_info = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 11) ∗ all code

This is considerably more concise than what you would get without punning:

```
# let create_host_info
    ~hostname:hostname ~os_name:os_name
    ~cpu_arch:cpu_arch ~os_release:os_release =
    { os_name = os_name;
      cpu_arch = cpu_arch;
      os_release = os_release;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
 val create_host_info :
   hostname:string ->
   os_name:string -> cpu_arch:string -> os_release:string -> host_info = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 12) ∗ all code

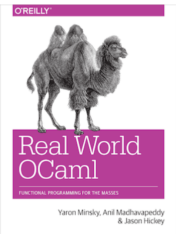
Together, labeled arguments, field names, and field and label punning encourage a style where you propagate the same names throughout your codebase. This is generally good practice, since it encourages consistent naming, which makes it easier to navigate the source.

REUSING FIELD NAMES

Defining records with the same field names can be problematic. Let's consider a simple example: building types to represent the protocol used for a logging server.

We'll describe three message types: `log_entry`, `heartbeat`, and `logon`. The `log_entry` message is used to deliver a log entry to the server; the `logon` message is sent to initiate a connection and includes the identity of the user connecting and credentials used for authentication; and the `heartbeat` message is periodically sent by the client to demonstrate to the server that the client is alive and connected. All of these messages include a session ID and the time the message was generated:

```
# type log_entry =
    { session_id: string;
      time: Time.t;
      important: bool;
      message: string;
    }
  type heartbeat =
    { session_id: string;
      time: Time.t;
      status_message: string;
    }
  type logon =
```



Buy in print and eBook.

Table of Contents


```
    { session_id: string;
      time: Time.t;
      user: string;
      credentials: string;
    }
;;
 type log_entry = {
   session_id : string;
   time : Time.t;
   important : bool;
   message : string;
 }
 type heartbeat = {
   session_id : string;
   time : Time.t;
   status_message : string;
 }
 type logon = {
   session_id : string;
   time : Time.t;
   user : string;
   credentials : string;
 }
```

OCaml Utop ∗ records/main.topscript , continued (part 13) ∗ all code

Reusing field names can lead to some ambiguity. For example, if we want to write a function to grab the `session_id` from a record, what type will it have?

```
# let get_session_id t = t.session_id;;
 val get_session_id : logon -> string = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 14) ∗ all code

In this case, OCaml just picks the most recent definition of that record field. We can force OCaml to assume we're dealing with a different type (say, a `heartbeat`) using a type annotation:

```
# let get_heartbeat_session_id (t:heartbeat) = t.session_id;;
 val get_heartbeat_session_id : heartbeat -> string = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 15) ∗ all code

While it's possible to resolve ambiguous field names using type annotations, the ambiguity can be a bit confusing. Consider the following functions for grabbing the session ID and status from a heartbeat:

```
# let status_and_session t = (t.status_message, t.session_id);;
 val status_and_session : heartbeat -> string * string = <fun>
# let session_and_status t = (t.session_id, t.status_message);;
 Characters 44-58:
 Error: The record type logon has no field status_message
# let session_and_status (t:heartbeat) = (t.session_id, t.status_message);;
 val session_and_status : heartbeat -> string * string = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 16) ∗ all code

Why did the first definition succeed without a type annotation and the second one fail? The difference is that in the first case, the type-checker considered the `status_message` field first and thus concluded that the record was a `heartbeat`. When the order was switched, the `session_id` field was considered first, and so that drove the type to be considered to be a `logon`, at which point `t.status_message` no longer made sense.

We can avoid this ambiguity altogether, either by using nonoverlapping field names or, more generally, by minting a module for each type. Packing types into modules is a broadly useful idiom (and one used quite extensively by Core), providing for each type a namespace within which to put related values. When using this style, it is standard practice to name the type associated with the module `t`. Using this style we would write:

```
# module Log_entry = struct
    type t =
      { session_id: string;
        time: Time.t;
        important: bool;
        message: string;
      }
  end
  module Heartbeat = struct
```

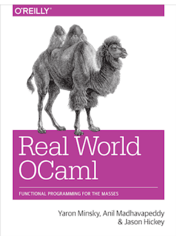


Table of Contents


```
    type t =
      { session_id: string;
        time: Time.t;
        status_message: string;
      }
  end
  module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
  end;;
module Log_entry :
  sig
    type t = {
      session_id : string;
      time : Time.t;
      important : bool;
      message : string;
    }
  end
module Heartbeat :
  sig
    type t = { session_id : string; time : Time.t; status_message : string; }
  end
module Logon :
  sig
    type t = {
      session_id : string;
      time : Time.t;
      user : string;
      credentials : string;
    }
  end
```

OCaml Utop ∗ records/main.topscript , continued (part 17) ∗ all code

Now, our log-entry-creation function can be rendered as follows:

```
# let create_log_entry ~session_id ~important message =
     { Log_entry.time = Time.now (); Log_entry.session_id;
       Log_entry.important; Log_entry.message }
  ;;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 18) ∗ all code

The module name `Log_entry` is required to qualify the fields, because this function is outside of the `Log_entry` module where the record was defined. OCaml only requires the module qualification for one record field, however, so we can write this more concisely. Note that we are allowed to insert whitespace between the module path and the field name:

```
# let create_log_entry ~session_id ~important message =
     { Log_entry.
       time = Time.now (); session_id; important; message }
  ;;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 19) ∗ all code

This is not restricted to constructing a record; we can use the same trick when pattern matching:

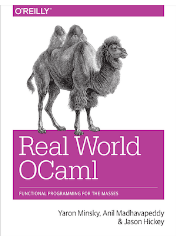
```
# let message_to_string { Log_entry.important; message; _ } =
    if important then String.uppercase message else message
  ;;
val message_to_string : Log_entry.t -> string = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 20) ∗ all code

When using dot notation for accessing record fields, we can qualify the field by the module directly:

```
# let is_important t = t.Log_entry.important;;
val is_important : Log_entry.t -> bool = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 21) ∗ all code



Buy in print and eBook.

Table of Contents

The syntax here is a little surprising when you first encounter it. The thing to keep in mind is that the dot is being used in two ways: the first dot is a record field access, with everything to the right of the dot being interpreted as a field name; the second dot is accessing the contents of a module, referring to the record field `important` from within the module `Log_entry`. The fact that `Log_entry` is capitalized and so can't be a field name is what disambiguates the two uses.

For functions defined within the module where a given record is defined, the module qualification goes away entirely.

FUNCTIONAL UPDATES

Fairly often, you will find yourself wanting to create a new record that differs from an existing record in only a subset of the fields. For example, imagine our logging server had a record type for representing the state of a given client, including when the last heartbeat was received from that client. The following defines a type for representing this information, as well as a function for updating the client information when a new heartbeat arrives:

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     last_heartbeat_time: Time.t;
   };;
 type client_info = {
   addr : UnixLabels.inet_addr;
   port : int;
   user : string;
   credentials : string;
   last_heartbeat_time : Time.t;
 }
# let register_heartbeat t hb =
      { addr = t.addr;
        port = t.port;
        user = t.user;
        credentials = t.credentials;
        last_heartbeat_time = hb.Heartbeat.time;
      };;
 val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 22) ∗ all code

This is fairly verbose, given that there's only one field that we actually want to change, and all the others are just being copied over from `t`. We can use OCaml's *functional update* syntax to do this more tersely. The syntax of a functional update is as follows:

```
{ <record> with <field> = <value>;
                <field> = <value>;
                ...
}
```

Syntax ∗ records/functional_update.syntax ∗ all code

The purpose of the functional update is to create a new record based on an existing one, with a set of field changes layered on top.

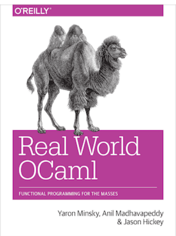
Given this, we can rewrite `register_heartbeat` more concisely:

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time = hb.Heartbeat.time };;
 val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 23) ∗ all code

Functional updates make your code independent of the identity of the fields in the record that are not changing. This is often what you want, but it has downsides as well. In particular, if you change the definition of your record to have more fields, the type system will not prompt you to reconsider whether your code needs to change to accommodate the new fields. Consider what happens if we decided to add a field for the status message received on the last heartbeat:

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
```



Buy in print and eBook.

Table of Contents


```
     last_heartbeat_time: Time.t;
     last_heartbeat_status: string;
   };;
 type client_info = {
   addr : UnixLabels.inet_addr;
   port : int;
   user : string;
   credentials : string;
   last_heartbeat_time : Time.t;
   last_heartbeat_status : string;
 }
```

OCaml Utop ∗ records/main.topscript , continued (part 24) ∗ all code

The original implementation of `register_heartbeat` would now be invalid, and thus the compiler would effectively warn us to think about how to handle this new field. But the version using a functional update continues to compile as is, even though it incorrectly ignores the new field. The correct thing to do would be to update the code as follows:

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time   = hb.Heartbeat.time;
             last_heartbeat_status = hb.Heartbeat.status_message;
    };;
 val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 25) ∗ all code

MUTABLE FIELDS

Like most OCaml values, records are immutable by default. You can, however, declare individual record fields as mutable. In the following code, we've made the last two fields of `client_info` mutable:

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     mutable last_heartbeat_time: Time.t;
     mutable last_heartbeat_status: string;
   };;
 type client_info = {
   addr : UnixLabels.inet_addr;
   port : int;
   user : string;
   credentials : string;
   mutable last_heartbeat_time : Time.t;
   mutable last_heartbeat_status : string;
 }
```

OCaml Utop ∗ records/main.topscript , continued (part 26) ∗ all code

The `<-` operator is used for setting a mutable field. The side-effecting version of `register_heartbeat` would be written as follows:

```
# let register_heartbeat t hb =
    t.last_heartbeat_time   <- hb.Heartbeat.time;
    t.last_heartbeat_status <- hb.Heartbeat.status_message
  ;;
 val register_heartbeat : client_info -> Heartbeat.t -> unit = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 27) ∗ all code

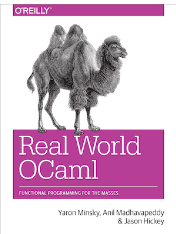
Note that mutable assignment, and thus the `<-` operator, is not needed for initialization because all fields of a record, including mutable ones, are specified when the record is created.

OCaml's policy of immutable-by-default is a good one, but imperative programming is an important part of programming in OCaml. We go into more depth about how (and when) to use OCaml's imperative features in the section called “Imperative Programming”.

FIRST-CLASS FIELDS

Consider the following function for extracting the usernames from a list of `Logon` messages:

```
# let get_users logons =
     List.dedup (List.map logons ~f:(fun x -> x.Logon.user));;
 val get_users : Logon.t list -> string list = <fun>
```



Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

OCaml Utop ∗ records/main.topscript , continued (part 28) ∗ all code

Here, we wrote a small function `(fun x -> x.Logon.user)` to access the `user` field. This kind of accessor function is a common enough pattern that it would be convenient to generate it automatically. The `fieldslib` syntax extension that ships with Core does just that.

The `with fields` annotation at the end of the declaration of a record type will cause the extension to be applied to a given type declaration. So, for example, we could have defined `Logon` as follows:

```
# module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
    with fields
  end;;
 module Logon :
   sig
     type t = {
       session_id : string;
       time : Time.t;
       user : string;
       credentials : string;
     }
     val credentials : t -> string
     val user : t -> string
     val time : t -> Time.t
     val session_id : t -> string
     module Fields :
       sig
         val names : string list
         val credentials :
           ([< `Read | `Set_and_create ], t, string) Field.t_with_perm
         val user :
           ([< `Read | `Set_and_create ], t, string) Field.t_with_perm
         val time :
           ([< `Read | `Set_and_create ], t, Time.t) Field.t_with_perm
         val session_id :
           ([< `Read | `Set_and_create ], t, string) Field.t_with_perm

         [ ... many definitions omitted ... ]

       end
   end
```

OCaml Utop ∗ records/main-29.rawscript ∗ all code

Note that this will generate *a lot* of output because `fieldslib` generates a large collection of helper functions for working with record fields. We'll only discuss a few of these; you can learn about the remainder from the documentation that comes with `fieldslib`.

One of the functions we obtain is `Logon.user`, which we can use to extract the user field from a logon message:

```
# let get_users logons = List.dedup (List.map logons ~f:Logon.user);;
 val get_users : Logon.t list -> string list = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 30) ∗ all code

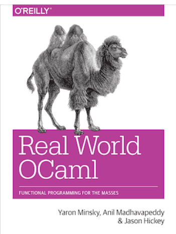
In addition to generating field accessor functions, `fieldslib` also creates a submodule called `Fields` that contains a first-class representative of each field, in the form of a value of type `Field.t`. The `Field` module provides the following functions:

`Field.name`
Returns the name of a field

`Field.get`
Returns the content of a field

`Field.fset`
Does a functional update of a field

`Field.setter`



Buy in print and eBook.

Table of Contents

Returns `None` if the field is not mutable or `Some f` if it is, where `f` is a function for mutating that field

A `Field.t` has two type parameters: the first for the type of the record, and the second for the type of the field in question. Thus, the type of `Logon.Fields.session_id` is `(Logon.t, string) Field.t`, whereas the type of `Logon.Fields.time` is `(Logon.t, Time.t) Field.t`. Thus, if you call `Field.get` on `Logon.Fields.user`, you'll get a function for extracting the `user` field from a `Logon.t`:

```
# Field.get Logon.Fields.user;;
 - : Logon.t -> string = <fun>
```
OCaml Utop ∗ records/main.topscript , continued (part 31) ∗ all code

Thus, the first parameter of the `Field.t` corresponds to the record you pass to `get`, and the second parameter corresponds to the value contained in the field, which is also the return type of `get`.

The type of `Field.get` is a little more complicated than you might naively expect from the preceding one:

```
# Field.get;;
 - : ('b, 'r, 'a) Field.t_with_perm -> 'r -> 'a = <fun>
```
OCaml Utop ∗ records/main.topscript , continued (part 32) ∗ all code

The type is `Field.t_with_perm` rather than `Field.t` because fields have a notion of access control that comes up in some special cases where we expose the ability to read a field from a record, but not the ability to create new records, and so we can't expose functional updates.

We can use first-class fields to do things like write a generic function for displaying a record field:

```
# let show_field field to_string record =
    let name = Field.name field in
    let field_string = to_string (Field.get field record) in
    name ^ ": " ^ field_string
  ;;
 val show_field :
   ('a, 'b, 'c) Field.t_with_perm -> ('c -> string) -> 'b -> string = <fun>
```
OCaml Utop ∗ records/main.topscript , continued (part 33) ∗ all code

This takes three arguments: the `Field.t`, a function for converting the contents of the field in question to a string, and a record from which the field can be grabbed.

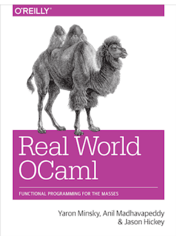
Here's an example of `show_field` in action:

```
# let logon = { Logon.
                session_id = "26685";
                time = Time.now ();
                user = "yminsky";
                credentials = "Xy2d9W"; }
  ;;
 val logon : Logon.t =
   {Logon.session_id = "26685"; time = 2013-11-05 08:49:43.946365-05:00;
    user = "yminsky"; credentials = "Xy2d9W"}
# show_field Logon.Fields.user Fn.id logon;;
 - : string = "user: yminsky"
# show_field Logon.Fields.time Time.to_string logon;;
 - : string = "time: 2013-11-05 08:49:43.946365-05:00"
```
OCaml Utop ∗ records/main.topscript , continued (part 34) ∗ all code

As a side note, the preceding example is our first use of the `Fn` module (short for "function"), which provides a collection of useful primitives for dealing with functions. `Fn.id` is the identity function.

`fieldslib` also provides higher-level operators, like `Fields.fold` and `Fields.iter`, which let you walk over the fields of a record. So, for example, in the case of `Logon.t`, the field iterator has the following type:

```
# Logon.Fields.iter;;
 - : session_id:(([< `Read | `Set_and_create ], Logon.t, string)
                 Field.t_with_perm -> 'a) ->
     time:(([< `Read | `Set_and_create ], Logon.t, Time.t) Field.t_with_perm ->
```



Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

```
          'b) ->
      user:(([< `Read | `Set_and_create ], Logon.t, string) Field.t_with_perm ->
          'c) ->
      credentials:(([< `Read | `Set_and_create ], Logon.t, string)
                    Field.t_with_perm -> 'd) ->
      'd
  = <fun>
```

OCaml Utop ∗ records/main.topscript , continued (part 35) ∗ all code

This is a bit daunting to look at, largely because of the access control markers, but the structure is actually pretty simple. Each labeled argument is a function that takes a first-class field of the necessary type as an argument. Note that `iter` passes each of these callbacks the `Field.t`, not the contents of the specific record field. The contents of the field, though, can be looked up using the combination of the record and the `Field.t`.

Now, let's use `Logon.Fields.iter` and `show_field` to print out all the fields of a `Logon` record:

```
# let print_logon logon =
    let print to_string field =
      printf "%s\n" (show_field field to_string logon)
    in
    Logon.Fields.iter
      ~session_id:(print Fn.id)
      ~time:(print Time.to_string)
      ~user:(print Fn.id)
      ~credentials:(print Fn.id)
  ;;
 val print_logon : Logon.t -> unit = <fun>
# print_logon logon;;


 session_id: 26685
 time: 2013-11-05 08:49:43.946365-05:00
 user: yminsky
 credentials: Xy2d9W
 - : unit = ()
```

OCaml Utop ∗ records/main.topscript , continued (part 36) ∗ all code

One nice side effect of this approach is that it helps you adapt your code when the fields of a record change. If you were to add a field to `Logon.t`, the type of `Logon.Fields.iter` would change along with it, acquiring a new argument. Any code using `Logon.Fields.iter` won't compile until it's fixed to take this new argument into account.

Field iterators are useful for a variety of record-related tasks, from building record-validation functions to scaffolding the definition of a web form from a record type. Such applications can benefit from the guarantee that all fields of the record type in question have been considered.

< Previous Next >