6 Records

One of OCaml's best features is its concise and expressive system for declaring new data types. *Records* are a key element of that system. We discussed records briefly in Chapter 2 (A Guided Tour), but this chapter will go into more depth, covering more of the technical details, as well as providing advice on how to use records 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>;
      ...
    }
```

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

Here's a simple example: a `service_info` record that represents an entry from the `/etc/services` file on a typical Unix system. That file is used for keeping track of the well-known port and protocol name for protocols such as FTP or SSH. Note that we're going to open `Core` in this example rather than `Base`, since we're using the Unix API, which you need `Core` for.

```
open Core
type service_info =
  { service_name : string;
    port         : int;
    protocol     : string;
  }
```

We can construct a `service_info` just as easily as we declared its type. The following function tries to construct such a record given as input a line from `/etc/services` file. To do this, we'll use `Re`, a regular expression engine for OCaml. If you don't know how regular expressions work, you can just think of them as a simple pattern language you can use for parsing a string. (You may need to install it first by running `opam install re`.)

```
# #require "re";;
# let service_info_of_string line =
    let matches =
      let pat = "([a-zA-Z]+)[ \t]+([0-9]+)/([a-zA-Z]+)" in
```

https://doi.org/10.1017/9781009129220.008 Published online by Cambridge University Press

```
        Re.exec (Re.Posix.compile_pat pat) line
      in
      { service_name = Re.Group.get matches 1;
        port = Int.of_string (Re.Group.get matches 2);
        protocol = Re.Group.get matches 3;
      }
    ;;
  val service_info_of_string : string -> service_info = <fun>
```

We can now construct a concrete record by calling the function on a line from the file.

```
# let ssh = service_info_of_string "ssh 22/udp # SSH Remote Login
    Protocol";;
val ssh : service_info = {service_name = "ssh"; port = 22; protocol =
    "udp"}
```

You might wonder how the compiler inferred that our function returns a value of type `service_info`. In this case, the compiler bases its inference on the field names used in constructing the record. That inference is most straightforward when each field name belongs to only one record type. We'll discuss later in the chapter what happens when field names are shared across different record types.

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

```
# ssh.port;;
- : int = 22
```

When declaring an OCaml type, you always have the option of parameterizing it by a polymorphic type. Records are no different in this regard. As an example, here's a type that represents an arbitrary item tagged with a line number.

```
type 'a with_line_num = { item: 'a; line_num: int }
```

We can then write polymorphic functions that operate over this parameterized type. For example, this function takes a file and parses it as a series of lines, using the provided function for parsing each individual line.

```
# let parse_lines parse file_contents =
    let lines = String.split ~on:'\n' file_contents in
    List.mapi lines ~f:(fun line_num line ->
      { item = parse line;
        line_num = line_num + 1;
      })
  ;;
val parse_lines : (string -> 'a) -> string -> 'a with_line_num list =
    <fun>
```

We can then use this function for parsing a snippet of a real `/etc/services` file.

```
# parse_lines service_info_of_string
    "rtmp              1/ddp     # Routing Table Maintenance Protocol
     tcpmux            1/udp     # TCP Port Service Multiplexer
     tcpmux            1/tcp     # TCP Port Service Multiplexer";;
- : service_info with_line_num list =
[{item = {service_name = "rtmp"; port = 1; protocol = "ddp"};
    line_num = 1};
```

```
   {item = {service_name = "tcpmux"; port = 1; protocol = "udp"};
      line_num = 2};
   {item = {service_name = "tcpmux"; port = 1; protocol = "tcp"};
      line_num = 3}]
```

The polymorphism lets us use the same function when parsing a different format, like this function for parsing a file containing an integer on every line.

```
# parse_lines Int.of_string "1\n10\n100\n1000";;
- : int with_line_num list =
[{item = 1; line_num = 1}; {item = 10; line_num = 2};
 {item = 100; line_num = 3}; {item = 1000; line_num = 4}]
```

6.1 Patterns and Exhaustiveness

Another way of getting information out of a record is by using a pattern match, as shown in the following function.

```
# let service_info_to_string
    { service_name = name; port = port; protocol = prot  }
    =
    sprintf "%s %i/%s" name port prot
  ;;
val service_info_to_string : service_info -> string = <fun>
# service_info_to_string ssh;;
- : string = "ssh 22/udp"
```

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. That's 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 change our `service_info` record so that it preserves comments. We can do this by providing a new definition of `service_info` that includes a `comment` field:

```
type service_info =
  { service_name : string;
    port         : int;
    protocol     : string;
    comment      : string option;
  }
```

The code for `service_info_to_string` would continue to compile without change. But in this case, we should probably update the code so that the generated string includes the comment if it's there. It would be nice if the type system would warn us that we should consider updating the function.

Happily, OCaml offers an optional warning for missing fields in record patterns. With that warning turned on (which you can do in the toplevel by typing `#warnings "+9"`), the compiler will indeed warn us.

```
# #warnings "+9";;
# let service_info_to_string
    { service_name = name; port = port; protocol = prot  }
    =
    sprintf "%s %i/%s" name port prot
  ;;
Line 2, characters 5-59:
Warning 9 [missing-record-field-pattern]: the following labels are
    not bound in this record pattern:
comment
Either bind these labels explicitly or add '; _' to the pattern.
val service_info_to_string : service_info -> string = <fun>
```

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 service_info_to_string
    { service_name = name; port = port; protocol = prot; _ }
    =
    sprintf "%s %i/%s" name port prot
  ;;
val service_info_to_string : service_info -> string = <fun>
```

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-record-field-pattern] Missing fields in a record pattern.
  R Alias for warning 9.
```

You can think of OCaml's warnings as a powerful set of optional static analysis tools. They're enormously helpful in catching all sorts of bugs, and you should 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-4-33-40-41-42-43-34-44`.

The syntax of `-w` 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.

6.2 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 service_info_to_string { service_name; port; protocol; comment
    } =
    let base = sprintf "%s %i/%s" service_name port protocol in
    match comment with
    | None -> base
    | Some text -> base ^ " #" ^ text;;
val service_info_to_string : service_info -> string = <fun>
```

Field punning can also be used to construct a record. Consider the following updated version of `service_info_of_string`.

```
# let service_info_of_string line =
    (* first, split off any comment *)
    let (line,comment) =
      match String.rsplit2 line ~on:'#' with
      | None -> (line,None)
      | Some (ordinary,comment) -> (ordinary, Some comment)
    in
    (* now, use a regular expression to break up the
       service definition *)
    let matches =
      Re.exec
        (Re.Posix.compile_pat
           "([a-zA-Z]+)[ \t]+([0-9]+)/([a-zA-Z]+)")
        line
    in
    let service_name = Re.Group.get matches 1 in
    let port = Int.of_string (Re.Group.get matches 2) in
    let protocol = Re.Group.get matches 3 in
    { service_name; port; protocol; comment };;
val service_info_of_string : string -> service_info = <fun>
```

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_service_info ~service_name ~port ~protocol ~comment =
    { service_name; port; protocol; comment };;
val create_service_info :
  service_name:string ->
  port:int -> protocol:string -> comment:string option ->
    service_info =
  <fun>
```

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

```
# let create_service_info
        ~service_name:service_name ~port:port
        ~protocol:protocol ~comment:comment =
    { service_name = service_name;
      port = port;
      protocol = protocol;
      comment = comment;
    };;
val create_service_info :
  service_name:string ->
  port:int -> protocol:string -> comment:string option ->
    service_info =
  <fun>
```

Together, 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.

6.3 Reusing Field Names

Defining records with the same field names can be problematic. As a simple example, let's consider a collection of types representing the protocol of 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 when initiating 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_ns.t;
    important: bool;
    message: string;
  }
type heartbeat =
  { session_id: string;
    time: Time_ns.t;
    status_message: string;
  }
type logon =
  { session_id: string;
    time: Time_ns.t;
    user: string;
    credentials: string;
  }
```

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>
```

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>
```

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);;
Line 1, characters 45-59:
Error: This expression has type logon
       There is no field status_message within type logon
```

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.

Adding a type annotation resolves the ambiguity, no matter what order the fields are considered in.

```
# let session_and_status (t:heartbeat) = (t.session_id,
    t.status_message);;
val session_and_status : heartbeat -> string * string = <fun>
```

We can avoid the ambiguity altogether, either by using nonoverlapping field names or by putting different record types in different modules. Indeed, packing types into modules is a broadly useful idiom (and one used quite extensively by `Base`), 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`. So, we would write:

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

```
end
module Logon = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      user: string;
      credentials: string;
    }
end
```

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

```
# let create_log_entry ~session_id ~important message =
    { Log_entry.time = Time_ns.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>
```

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_ns.now (); session_id; important; message };;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

Earlier, we saw that you could help OCaml understand which record field was intended by adding a type annotation. We can use that here to make the example even more concise.

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

This is not restricted to constructing a record; we can use the same approaches 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>
```

When using dot notation for accessing record fields, we can qualify the field by the module as well.

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

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.

Qualifying a record field by the module it comes from can be awkward. Happily, OCaml doesn't require that the record field be qualified if it can otherwise infer the type of the record in question. In particular, we can rewrite the above declarations by adding type annotations and removing the module qualifications.

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

This feature of the language, known by the somewhat imposing name of *type-directed constructor disambiguation*, applies to variant tags as well as record fields, as we'll see in Chapter 7 (Variants).

6.4 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.

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

We could define a function for updating the client information when a new heartbeat arrives as follows.

```
# 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>
```

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 following shows how we can use functional updates to 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>
```

The `with` keyword marks that this is a functional update, and the value assignments on the right-hand side indicate the changes to be made to the record on the left-hand side of the `with`.

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;
    last_heartbeat_time: Time_ns.t;
    last_heartbeat_status: string;
  }
```

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>
```

These downsides notwithstanding, functional updates are very useful, and a good choice for cases where it's not important that you consider every field of the record when making a change.

6.5 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_ns.t;
    mutable last_heartbeat_status: string;
  }
```

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

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 Chapter 9 (Imperative Programming).

6.6 First-Class Fields

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

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

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 `ppx_fields_conv` syntax extension that ships with `Core` does just that.

The `[@@deriving fields]` annotation at the end of the declaration of a record type will cause the extension to be applied to a given type declaration. We need to enable the extension explicitly,

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

at which point, we can define `Logon` as follows:

```
# module Logon = struct
    type t =
      { session_id: string;
        time: Time_ns.t;
        user: string;
        credentials: string;
      }
    [@@deriving fields]
  end;;
module Logon :
    sig
      type t = {
        session_id : string;
        time : Time_ns.t;
        user : string;
        credentials : string;
      }
      val credentials : t -> string
      val user : t -> string
      val time : t -> Time_ns.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_ns.t)
      Field.t_with_perm
  ...
        end
    end
```

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_and_sort ~compare:String.compare
  (List.map logons ~f:Logon.user);;
val get_users : Logon.t list -> string list = <fun>
```

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` 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>
```

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>
```

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>
```

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_ns.of_string "2017-07-21 10:11:45 EST";
                user = "yminsky";
                credentials = "Xy2d9W"; };;
val logon : Logon.t =
  {Logon.session_id = "26685"; time = 2017-07-21 15:11:45.000000000Z;
   user = "yminsky"; credentials = "Xy2d9W"}
# show_field Logon.Fields.user Fn.id logon;;
- : string = "user: yminsky"
# show_field Logon.Fields.time Time_ns.to_string logon;;
- : string = "time: 2017-07-21 15:11:45.000000000Z"
```

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 -> unit) ->
```

```
      time:(([< `Read | `Set_and_create ], Logon.t, Time_ns.t)
            Field.t_with_perm -> unit) ->
      user:(([< `Read | `Set_and_create ], Logon.t, string)
      Field.t_with_perm ->
            unit) ->
      credentials:(([< `Read | `Set_and_create ], Logon.t, string)
                   Field.t_with_perm -> unit) ->
      unit
  = <fun>
```

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_ns.to_string)
      ~user:(print Fn.id)
      ~credentials:(print Fn.id);;
val print_logon : Logon.t -> unit = <fun>
# print_logon logon;;
session_id: 26685
time: 2017-07-21 15:11:45.000000000Z
user: yminsky
credentials: Xy2d9W
- : unit = ()
```

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.