Chapter 8

Row polymorphism

Consider the following code:

```
type name_home = {name: string; home: string}
type name_mobile = {name : string; mobile: string}

let jane = {name = "Jane"; home = "01234-654321"}

let john = {name = "John"; mobile = "07654-123456"}

let print_name r = print_endline ("Name: " ^ r.name)

let () =
  print_name jane;
  print_name john
```

We create two records, one for `jane` and one for `john`, and then we print the name fields of these records. These all seems pretty reasonable, but it doesn't work:

```
    print_name jane;
               ^^^^

Error: This expression has type name_home
       but an expression was expected of type name_mobile
```

The problem is that `jane` and `john` have different record field labels. Whilst both records have fields labelled `name`, `jane` also has a field labelled `home` whereas `john` has a field labelled `mobile`. Since they have different fields, `jane` and `john` have different types, but `print_name` only works on a single type (in this case `name_mobile`) hence the error message.

Since `print_name` only uses the `name` field, we would like to be able to use it with any type that has such a field. In other words, we would like the type of `print_name` to be polymorphic in such a way that it can be instantiated to any type with a `name` field.

More generally, we would like all operations on records to have polymorphic types, which can be instantiated to any record type which meets the requirements of that operation. There are three basic operations on records, from which we can build up any more complex operations:

1. An empty record (`empty`)
2. Extend a record with a field (`extend`)
3. Access the contents of a field (`access`)

8.1 Presence variables

8.1.1 Presence and absence

For a finite set of field labels we can define polymorphic record types by representing an absent field using a nullary type constructor `absent` and a present field using a unary type constructor `present` whose argument is the type of the field.

For example, records with fields labelled either `name`, `home` or `mobile` can be encoded as:

```
type absent = Absent

type 'a present = Present : 'a -> 'a present

type ('name, 'home, 'mobile) record =
  { name: 'name;
    home: 'home;
    mobile: 'mobile; }
```

The basic operations on these records are defined as:

```
let empty =
  { name = Absent; home = Absent; mobile = Absent }

let extend_name name { home; mobile; _ } =
  { name = Present name; home; mobile }

let extend_home home { name; mobile; _ } =
  { name; home = Present home; mobile }

let extend_mobile mobile { name; home; _ } =
  { name; home; mobile = Present mobile }

let access_name { name = Present name; _ } = name

let access_home { home = Present home; _ } = home
let access$_{mobile}$ { mobile = Present mobile; _ } = mobile
```

which have types:

```
val empty : (absent, absent, absent) record

val extend$_{name}$ : 'd -> ('a, 'b, 'c) record ->
                      ('d present, 'b, 'c) record

val extend$_{home}$ : 'd -> ('a, 'b, 'c) record ->
                      ('a, 'd present, 'c) record

val extend$_{mobile}$ : 'd -> ('a, 'b, 'c) record ->
                        ('a, 'b, 'd present) record

val access$_{name}$ : ('a present, 'b, 'c) record -> 'a

val access$_{home}$ : ('a, 'b present, 'c) record -> 'b

val access$_{mobile}$ : ('a, 'b, 'c present) record -> 'c
```

As you can see, the functions for extending and accessing records are now polymorphic in the presence of the fields that they do not operate on.

Using this scheme, we can encode our `print_name` example as follows:

```
let jane = extend$_{name}$ “Jane”
             (extend$_{home}$ “01234-654321” empty)

let john = extend$_{name}$ “John”
             (extend$_{mobile}$ “07654-123456” empty)

let print_name r =
  print_endline (“Name: ” ^ (access$_{name}$ r))

let () =
  print_name jane;
  print_name john
```

8.1.2 Polymorphic record extension

It is clear that field access should only work on records for which the field is present. However, should record extension always return a record where the field is present?

Consider the following code:

```
let person p = if p then jane else john
```

With our original encoding this does not type-check because `jane` and `john` do not have the same set of fields. However, it should be safe to give `person` the type `bool -> (string present, absent, absent) record` because both `jane` and `john` are records with a `name` field of type `string`. In general, it should always be safe to treat a record as if it had fewer fields than it actually has.

Whilst it is safe to treat a record with a `home` field of type `string` as if it did not have a `home` field, it is not safe to treat it as if the `home` field had type `int`. So record extension can be polymorphic in the presence of a field but not in the type of a field. This means that we would like the definition of record extension to use *higher-kinded* polymorphism.

Since higher-kinded polymorphism in OCaml requires the module system – which is quite verbose – we will express the type of record extension using System Fω.:

$$\begin{aligned}\text{extend}_{home} \; : \; & \forall\alpha{:}*.\ \forall\beta{:}*.\ \forall\gamma{:}*.\ \forall\delta{:}*.\ \forall\varphi{:}* \Rightarrow *.\\ & \delta \to \text{Record}\ \alpha\ \beta\ \gamma \to \text{Record}\ (\varphi\ \delta)\ \beta\ \gamma\end{aligned}$$

where φ is a type constructor variable that can be instantiated with either `Present` or ($\lambda\alpha{:}*.$ `Absent`). We call such higher-kinded type variables *presence variables.*

8.1.3 Ensuring record types are well-formed

Polymorphic record types using presence variables allow some type expressions that don't make sense by instantiating the presence variables with type constructors other than `Present` or `Absent` (e.g. `Record Int (Present String) (Present String)`) or by using `Present` or `Absent` outside of a record type (e.g. `List (Present Int)`). This is still type-safe because all the operations on record types only work on well-formed record types, however it is undesirable to allow ill-formed types to even be expressed.

Such ill-formed types are usually prevented using a kind system by creating a new kind `presence` such that:

$$\begin{aligned}\text{Absent} &: \text{presence}\\ \text{Present} &: * \Rightarrow \text{presence}\\ \text{Record} &: \text{presence} \Rightarrow \text{presence} \Rightarrow \text{presence} \Rightarrow *\end{aligned}$$

8.2 Row variables

In the previous section we were able to encode polymorphic records for a fixed finite set of field labels. This requires us to know in advance all the field labels that will be used with our polymorphic records, which prevents separate compilation. What we really want is to be able to encode polymorphic records for an infinite set of field labels (e.g. all possible strings of alphanumeric characters). We will write `L` to represent some infinite set of labels.

If we assume the existence of infinite record types, containing one field for each of the field labels, then we can use the same encoding as in the previous section to encode polymorphic records. We will write `{…; foo: bar; …}` to represent such a record type with the `foo` field having type `bar`, and we will use `l` and `m` to represent generic labels.

$$\text{empty} : \{\dots\ ;\ \text{l} : \text{Absent}\ ;\ \dots\}$$

$$\begin{aligned}\text{extend}_m : {} & \forall\alpha{:}*.\ \forall\beta{:}\text{presence}.\ \dots\ .\forall\ \gamma_l{:}\text{presence}.\ \dots \\ & \forall\ \varphi{:}* \Rightarrow \text{presence}. \\ & \quad \alpha \to \{\dots\ ;\ \text{m} : \beta\ ;\ \dots\ ;\ \text{l} : \gamma_l\ ;\ \dots\} \to \\ & \qquad \{\dots\ ;\ \text{m} : \varphi\ \alpha\ ;\ \dots\ ;\ \text{l} : \gamma_l\ ;\ \dots\}\end{aligned}$$

$$\begin{aligned}\text{access}_m : {} & \forall\alpha{:}*.\ \dots\ .\forall\ \beta_l{:}\text{presence}.\ \dots \\ & \{\dots\ ;\ \text{m} : \text{Present}\ \alpha\ ;\ \dots\ ;\ \text{l} : \beta_l\ ;\ \dots\} \to \\ & \alpha\end{aligned}$$

Unfortunately, we cannot use such infinite record types directly, because they have an infinite number of type parameters. This means, for example, that unification of two of these types will not terminate. However, the type schemes for `empty`, `extend_m` and `access_m` above only use infinite record types of a particular form. Each record type appearing above can be divided into two parts:

1. A finite part
2. A co-finite part where either every type parameter is a free variable or every type parameter is `Absent`.

The infinite record types in the above definitions are divided as follows:

Record type	Finite part	Co-finite part
$\{\dots\ ;\ \text{l} : \text{Absent}\ ;\ \dots\}$	$\{\ \}$	$\{\ \dots\ ;\ \text{l} : \text{Absent}\ ;\ \dots\}$
$\{\dots\ ;\ \text{m} : \beta\ ;\ \dots\ ;\ \text{l} : \gamma_l\ ;\ \dots\}$	$\{\text{m} : \beta\}$	$\{\dots\ ;\ \text{l} : \gamma_l\ ;\ \dots\}$
$\{\dots\ ;\ \text{m} : \varphi\,\alpha\ ;\ \dots\ ;\ \text{l} : \gamma_l\ ;\ \dots\}$	$\{\text{m} : \varphi\,\alpha\}$	$\{\dots\ ;\ \text{l} : \gamma_l\ ;\ \dots\}$
$\{\dots\ ;\ \text{m} : \text{Present}\ \alpha\ ;\ \dots\ ;\ \text{l} : \beta_l\ ;\ \dots\}$	$\{\ \text{m} : \text{Present}\ \alpha\ \}$	$\{\dots\ ;\ \text{l} : \beta_l\ ;\ \dots\}$

By splitting our infinite record types up like this we can give them a finite representation. If the co-finite part has every type parameter equal to `absent` then we represent the record type by just its finite part. If the co-finite part has type variables for every parameter then we represent the record type as a pair of the finite part and a single type variable to represent the co-finite part. For convenience we write this pair as `{ finite-part |` ρ `}`.

The type variables ρ representing co-finite records are called *row variables*. As with presence variables, the kind system is usually used to ensure that record types are well formed. A row variable reprsenting a co-finite record without labels $\texttt{m}_1$ to $\texttt{m}_n$ is given kind $\texttt{row(m}_1\texttt{, ..., m}_n\texttt{)}$. This prevents types which unify row variables with regular types (e.g. `{ m : Present Float | Int }`), or record types including multiple occurrences of the same label (e.g. `{ m : Present Float | { m: Present Int } }`).

Using this finite representation, our operations have the following type schemes:

$$
\begin{aligned}
&\text{empty} : \{\}\\[1em]
&\text{extend}_m : \forall\alpha{:}*.\ \forall\beta{:}\text{presence}.\ \forall\ \rho{:}\text{row(m)}.\\
&\qquad\qquad\qquad \forall\ \varphi{:}* \Rightarrow \text{presence}.\\
&\qquad\qquad\qquad\quad \alpha \to \{\text{m} : \beta \mid \rho\} \to \{\text{m} : \varphi\ \alpha \mid \rho\}\\[1em]
&\text{access}_m : \forall\alpha{:}*.\ \forall\rho{:}\text{row(m)}.\\
&\qquad\qquad\quad \{\text{m} : \text{Present}\ \alpha \mid \rho\} \to \alpha
\end{aligned}
$$

With this finite representation we can perform unification using variations of the standard ML algorithms. These algorithms maintain all the good properties that ML type inference has, in particular they still have principal types. Infinite record types which can be represented in this way are closed under unification, which means that we never have to handle a record type which is not of this form.

8.3 Polymorphic variants

Consider the following code:

```
type number =
  | Int : int -> number
  | Float : float -> number

let square = function
  | Int i -> Int (i * i)
  | Float f -> Float (f *. f)

type constant =
  | Int : int -> constant
  | Float : float -> constant
  | String : string -> constant

let print_constant = function
  | Int i -> print_int i
  | Float f -> print_float f
  | String s -> print_string s

let () = print_constant (square (Int 5))
```

We define a function `square` which squares a number that is either an `int` or a `float`, and a function `print_constant` which prints something that is either an `int`, a `float` or a `string`. We then use these functions to square 5 and print the result.

This code seems reasonable, but will not type check.

```
Characters 24-40:
```

```
let () = print_constant (square (Int 5));;
                        ^^^^^^^^^^^^^^^^
Error: This expression has type number but an expression was expected
       of type constant
```

The `square` function and the `print_constant` function work on different variant constructor labels. Whilst they both work on constructors labelled `Int` and `Float`, `print_constant` also works on constructors labelled `String`. Since they work on different constructors, `square` and `print_constant` work on different types, hence the error.

You may notice some similarity between the above example and `jane` and `john` example from the beginning of this chapter. This comes from the duality between records and variants. This duality implies that we can also use row variables and presence variables to support this example.

As with records, what we would like is for all operations on variants to have polymorphic types, which can be instantiated to any variant type which meets the requirements of that operation. The three basic operations on variants, dual to the three basic operations on records, are as follows:

1. Match a variant with no constructors (`match_empty`)
2. Extend a match with a variant constructor (`extend_match`)
3. Use a variant constructor (`create`)

Using these operations, we can write `square` and `print_constant` as follows:

```
let square =
  extend_match_Int (fun i -> create_Int (i * i))
    (extend_match_Float (fun f -> create_Float (f *. f))
      match_empty)

let print_constant =
  extend_match_Int (fun i -> print_int i)
    (extend_match_Float (fun f -> print_float f)
      (extend_match_String (fun s -> print_string s)
        match_empty))

let () = print_constant (square (create_Int 5))
```

To distinguish polymorphic variant types from polymorphic record types we will write them using the syntax `[Foo : bar | ρ]` where `Foo` is a constructor label, `bar` is the type of the constructor labelled `Foo` and ρ is the (optional) row variable.

Using row variables and presence variables, we can give the basic operations on variants the following types:

$$\text{match\_empty} : \forall\ \alpha{:}*.\ [\ ] \to \alpha$$

```
extend_match$_M$: ∀α:∗. ∀β:presence. ∀ γ:∗.
                          ∀ ρ:row(M). ∀ φ: ∗ ⇒ presence.
                             (α → γ) → ([M : β | ρ] → γ) →
                               [M : φ α | ρ] → γ

create$_M$ : ∀α:∗. ∀ρ:∗row(M).
                  α → [M : Present α| ρ]
```

8.4 Row polymorphism in OCaml

Regular OCaml records and variants are not polymorphic. There are two good reasons for this:

1. Monomorphic records and variants can be implemented more efficiently than their polymorphic counterparts.
2. Polymorphic records and variants are more flexible, but this additional flexibility makes types more complicated and type errors more difficult to understand.

However, OCaml does also provide *objects* (similar to polymorphic records) and *polymorphic variants*. Both of these are typed using forms of row polymorphism. In both cases the form of row polymorphism is more restrictive the row polymorphism described in this chapter, and this section discusses the limitations of these restricted forms.

8.4.1 OCaml's objects

The `jane` and `john` example from the beginning of the chapter can be implemented using OCaml objects:

```
let jane = object
  method name = “Jane”
  method home = “01234−654321”
end

let john = object
  method name = “John”
  method mobile = “07654−123456”
end

let print_name r = print_endline (“Name: ” ^ r#name)

let () =
  print_name jane;
  print_name john
```

The OCaml object type syntax comes in two forms:

- `< foo : int; bar : float >` represents an object type where the method `foo` has type `int` and the method `bar` has type `float`. Both methods are present, and all other methods are absent.
- `< foo: int; bar : float; .. >` is the same as above, except the object may contain other methods besides `foo` and `bar`. In other words, the `..` represents an unnamed row variable.

Using this syntax, the type schemes of `jane`, `john` and `print_name` are as follows:

```
val jane : < home : string; name : string >

val john : < mobile : string; name : string >

val print_name : < name : string; .. > -> unit
```

8.4.2 OCaml's Polymorphic variants

Using OCaml's polymorphic variants we can write the `square` example:

```
let square = function
  | `Int i -> `Int (i * i)
  | `Float f -> `Float (f *. f)

let print_constant = function
  | `Int i -> print_int i
  | `Float f -> print_float f
  | `String s -> print_string s

let () = print_constant (square (`Int 5))
```

The OCaml polymorphic variant type syntax is a bit complicated and comes in four forms:

- ``[ `Foo of int | `Bar of float ]`` represents a variant type where the constructor `` `Foo`` has type `int` and the constructor `` `Bar`` has type `float`. Both constructors are definitely present.
- ``[< `Foo of int | `Bar of float ]`` is the same as above, except that it is polymorphic in the presence of both constructors. In other words, the `<` represents two unnamed presence variables.
- ``[< `Foo of int | `Bar of float > `Bar ]`` is the same as above, except that it is only polymorphic in the presence of the `` `Foo`` constructor – the `` `Bar`` constructor is definitely present. In other words, the `<` represents a single unnamed presence variable associated with `` `Foo``.

- `[> `Foo of int | `Bar of float]` is the same as the first form, except their may be more constructors than just `` `Foo `` and `` `Bar ``. In other words, the `>` represents an unnamed row variable.

Using this syntax, the type schemes of the `square` and `print_constant` functions are as follows:

```
val square : [< `Float of float | `Int of int ] ->
               [> `Float of float | `Int of int ]

val print_constant :
    [< `Float of float | `Int of int | `String of string ] ->
      unit
```

8.4.3 Limitations

Hidden row variables

Both object types and polymorphic variant types in OCaml hide the row variables. This simplifies the types and avoids the need to expose the user to type variables of kinds other than `*`. However, it also places some limitations on how objects and polymorphic records can be used – relative to the more explicit row polymorphism described in the previous sections.

Not allowing row variables to be named means that row variables cannot be shared between different object types. Looking at the types of the basic operations on records, we can see that this means we cannot express the type of polymorphic record extension ($\texttt{extend}_m$).

Instead of supporting polymorphic record extension, OCaml objects can only be created monomorphically using object literals. This is equivalent to providing a family of operations of the form:

```
val create_{l,m,n} : 'a -> 'b -> 'c ->
                          < l : 'a; m : 'b; n : 'c >
```

instead of `empty` and $\texttt{extend}_m$.

In practical terms, this means that we cannot take a generic object and extend it with a new method. We can only create a new record if we statically know all of the its methods.

As with object types, row variables cannot be shared between different variant types. Looking at the types of the basic operations on variants, we can see that this means we cannot express the type of polymorphic match extension ($\texttt{extend\_match}_M$).

Instead of supporting polymorphic match extension, polymorphic variants can only be matched using a single match statement. This is equivalent to providing a family of operations of the form:

```
val match_{L,M,N} :
  ('a -> 'd) -> ('b -> 'd) -> ('c -> 'd) ->
    [< `L of 'a | `M of 'b | `N of 'c] -> 'd
```

instead of `match_empty` and `extend_match`$_M$.

In practical terms, this means we cannot take a generic function on polymorphic variants and extend it with a new case for a variant constructor.

Objects and presence variables

OCaml's objects (unlike OCaml's polymorphic variants) do not support presence variables. This simplifies the types for objects, but means that the `person` example cannot be type-checked using OCaml objects:

```
let person p = if p then jane else john
```

```
Characters 35-39:
  let person p = if p then jane else john
                                     ^^^^
Error: This expression has type < mobile : string; name : string >
       but an expression was expected of type
         < home : string; name : string >
       The second object type has no method mobile
```

However, separately from row polymorphism, OCaml does provide an explicit subtyping coercion operator `:>`, which can be used in this case:

```
# let person p =
     if p then (jane :> < name : string >)
     else (john :> < name : string >);;
val person : bool -> < name : string > = <fun>
```

Subtyping is a related but separate concept from row polymorphism. For a good overview of subtyping see Part III of *Types and Programming Languages* by Pierce.