Row polymorphism

Record operations

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

Presence variables

$$\texttt{extend}_{home} : \forall\alpha{:}*.\ \forall\beta{:}*.\ \forall\gamma{:}*.\ \forall\delta{:}*.\ \forall\varphi{:}* \Rightarrow *.\ \delta \to \texttt{Record}\ \alpha\ \beta\ \gamma \to \texttt{Record}\ (\varphi\ \delta)\ \beta\ \gamma$$

where φ is a type constructor variable that can be instantiated with:

- `Present`
- $(\lambda\alpha{:}*.$ `Absent`$)$

Ill-formed records

Polymorphic record types allow some ill-formed type expressions:

- `Record Int (Present String) (Present String)`
- `List (Present Int)`

These are prevented using the kind system by creating a new kind `presence` such that:

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

Infinite records

What if we had infinite record types:

$$\{\ldots;\ \mathsf{foo}:\ \mathsf{bar};\ \ldots\}$$

Infinite records

$$\mathsf{empty} : \{\ldots ; \mathsf{l} : \mathsf{Absent} ; \ldots\}$$

$$\mathsf{extend}_m :$$
$$\forall\, \alpha{:}*.\ \forall\, \beta{:}\mathsf{presence}.\ \ldots\ .\forall\, \gamma_l{:}\mathsf{presence}.\ \ldots$$
$$\forall\, \varphi{:}* \Rightarrow \mathsf{presence}.$$
$$\alpha \to \{\ldots ; \mathsf{m} : \beta ; \ldots ; \mathsf{l} : \gamma_l ; \ldots\} \to$$
$$\{\ldots ; \mathsf{m} : \varphi\ \alpha ; \ldots ; \mathsf{l} : \gamma_l ; \ldots\}$$

$$\mathsf{access}_m :$$
$$\forall\, \alpha{:}*.\ \ldots\ .\forall\, \beta_l{:}\mathsf{presence}.\ \ldots$$
$$\{\ldots ; \mathsf{m} : \mathsf{Present}\ \alpha ; \ldots ; \mathsf{l} : \beta_l ; \ldots\} \to$$
$$\alpha$$

Infinite records

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`.

Infinite records

$$\{\ldots\ ;\ l\ :\ \mathsf{Absent}\ ;\ \ldots\}$$

Infinite records

$$\{\ldots\ ;\ l\ :\ \mathsf{Absent}\ ;\ \ldots\}$$

$\{\ \}$ $\{\ \ldots\ ;\ l\ :\ \mathsf{Absent}\ ;\ \ldots\}$

Finite Co-finite

Infinite records

$$\{\ldots \;;\; \mathsf{m} : \beta \;;\; \ldots \;;\; \mathsf{l} : \gamma_l \;;\; \ldots\}$$

Infinite records

$$\{\ldots \;;\; \mathsf{m} : \beta \;;\; \ldots \;;\; \mathsf{l} : \gamma_l \;;\; \ldots\}$$

$\{\mathsf{m} : \beta\}$ $\{\ldots \;;\; \mathsf{l} : \gamma_l \;;\; \ldots\}$

Finite Co-finite

Infinite records

$$\{\ldots \;;\; \mathsf{m} : \varphi\ \alpha \;;\; \ldots \;;\; \mathsf{l} : \gamma_l \;;\; \ldots\}$$

Infinite records

$$\{\ldots \;;\; \mathsf{m} : \varphi\ \alpha \;;\; \ldots \;;\; \mathsf{l} : \gamma_l \;;\; \ldots\}$$

$\{\mathsf{m} : \varphi\ \alpha\}$ Finite

$\{\ldots \;;\; \mathsf{l} : \gamma_l \;;\; \ldots\}$ Co-finite

Infinite records

$$\{\ldots\ ;\ \mathsf{m}\ :\ \mathsf{Present}\ \alpha\ ;\ \ldots\ ;\ \mathsf{l}\ :\ \beta_l\ ;\ \ldots\}$$

Infinite records

$$\{\ldots\ ;\ \mathsf{m}\ :\ \mathsf{Present}\ \alpha\ ;\ \ldots\ ;\ \mathsf{l}\ :\ \beta_l\ ;\ \ldots\}$$

$$\{\ \mathsf{m}\ :\ \mathsf{Present}\ \alpha\ \}\qquad \{\ldots\ ;\ \mathsf{l}\ :\ \beta_l\ ;\ \ldots\}$$

Finite

Co-finite

Row variables

$$\{ \ldots ; \mathsf{l} : \mathsf{Absent} ; \ldots \}$$

Row variables

$$\{ \ \ldots \ ; \ \mathsf{l} \ : \ \mathsf{Absent} \ ; \ \ldots \}$$

$$\Downarrow$$

$$\{ \ \}$$

Row variables

$$\{\ldots\ ;\ \mathsf{m}\ :\ \mathsf{Present}\ \alpha\ ;\ \ldots\ ;\ \mathsf{l}\ :\ \beta_l\ ;\ \ldots\}$$

Row variables

$$\{\ldots\ ;\ \mathsf{m}\ :\ \mathsf{Present}\ \alpha\ ;\ \ldots\ ;\ \mathsf{l}\ :\ \beta_l\ ;\ \ldots\}$$

$$\Downarrow$$

$$\{\mathsf{m}\ :\ \mathsf{Present}\ \alpha\ \mid\ \rho\}$$

Row variables

$$\mathsf{empty} \;:\; \{\}$$

$$\mathsf{extend}_m \;:\; \forall \alpha : * .\; \forall\, \beta : \mathsf{presence} .\; \forall\; \rho : \mathsf{row}(\mathsf{m}) .\; \forall\; \varphi : * \Rightarrow \mathsf{presence} .\; \alpha \to \{\mathsf{m} \;:\; \beta \;\mid\; \rho\} \to \{\mathsf{m} \;:\; \varphi\; \alpha \;\mid\; \rho\}$$

$$\mathsf{access}_m \;:\; \forall \alpha : * .\; \forall\; \rho : \mathsf{row}(\mathsf{m}) .\; \{\mathsf{m} \;:\; \mathsf{Present}\; \alpha \;\mid\; \rho\} \to \alpha$$

Variant operations

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`)

Variant operations

```
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))
```

Variant operations

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

$$\mathsf{extend\_match}_M:$$
$$\forall\ \alpha{:}*.\ \forall\ \beta{:}\mathsf{presence}.\ \forall\ \gamma{:}*.$$
$$\forall\ \rho{:}\mathsf{row(M)}.\ \forall\ \varphi{:}\ *\Rightarrow \mathsf{presence}.$$
$$(\alpha \to \gamma) \to ([\mathsf{M}:\ \beta\ |\ \rho] \to \gamma) \to$$
$$[\mathsf{M}:\ \varphi\ \alpha\ |\ \rho] \to \gamma$$

$$\mathsf{create}_M\ :\ \forall\alpha{:}*.\ \forall\,\rho{:}\mathsf{row(M)}.$$
$$\alpha \to [\mathsf{M}:\ \mathsf{Present}\ \alpha|\ \rho]$$

Object types

```
< foo : int; bar : float >
```

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.

Object types

```
< foo : int; bar : float; .. >
```

The object may contain other methods besides `foo` and `bar`. In other words, the `..` represents an unnamed row variable.

Object limitations

Instead of `extend`$_m$ we have:

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

Polymorphic variant types

```
[ `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.

Polymorphic variant types

```
[< `Foo of int | `Bar of float ]
```

The variant is polymorphic in the presence of both constructors. In other words, the < represents two unnamed presence variables.

Polymorphic variant types

```
[< `Foo of int | `Bar of float > `Bar]
```

The variant 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.

Polymorphic variant types

```
[> `Foo of int | `Bar of float ]
```

The variant may contain more constructors than just `Foo and `Bar. In other words, the > represents an unnamed row variable. Constructors `Foo and `Bar are definitely present.

Variant limitations

Instead of `extend_match`$_m$ we have:

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