Polymorphic Variants, Labels

OCaml PRO – Benjamin Canou

March 9-13 2015

Outline

Labeled & Optional arguments

Labeled arguments
Optional arguments

Let's define a function that adds a prefix and a postfix to a string:

```
let delimit pre s post = pre ^ s ^ post
```

In the signature, it appears as:

```
val delimit : string -> string -> string -> string
(** [delimit pre s post ]adds a prefix
    and a postfix to a string [s] *)
```

The function type is ambiguous, it needs extra documentation in english.

Labeled arguments 2 / 4

Alternative definition using labels:

```
let delimit ~pre s ~post = pre ^ s ^ post
```

In the signature, it appears as:

```
val delimit
  : pre:string ->
    string ->
    post:string ->
    string
```

Problem solved !

Labeled arguments

Total application:

```
delimit "<" "xx" ">" ;;
delimit ~post:">" "xx" ~pre:"<" ;;
delimit ~pre:"<" "xx" ~post:">" ;;
delimit ~pre:"<" ~post:">" "xx" ;;
delimit "xx" ~pre:"<" ~post:">" ;;
```

Labels can be omitted if all arguments are passed.
Labeled arguments can be put in any order, mixed with others.
Other arguments must be passed in declaration order.

Labeled arguments

Partial application:

```
let prefix = delimit ~post:""
let postfix = delimit ~pre:""
let wrap = delimit ~pre:"(" ~post:")"
```

Labels are required for partial application.

Bonus: ~pre:pre simplifies to ~pre.

```
let pre = "<" and post = ">" in
delimit ~pre ~post "-"
```

Let's define a function that concatenates the strings in a list, with a separator:

```
let rec cat sep = function
  | [] -> ""
  | [ single ] -> single
  | word :: words -> word ^ sep ^ cat sep words
```

Nine out of ten times, we end up writing: `cat "" strings`

We could have written:

```
let cat
  : ?sep:string -> string list -> string list
  = fun ?sep strs ->
  let rec cat strs = match strs, sep with
  | [], _ -> ""
  | [ single ], _ -> single
  | word :: words, Some sep -> word ^ sep ^ cat words
  | word :: words, None -> word ^ cat words in
  cat strs
```

The argument sep has `string option` type.

- When passed, with `cat ~sep:"," words`, it is `Some ","`.
- When not passed, with `cat words`, it is `None`.

Optional arguments

Or alternatively:

```
let cat ?sep strs = match strs, sep with
  | [], _ -> ""
  | [ single ], _ -> single
  | word :: words, Some s -> word ^ s ^ cat ?sep words
  | word :: words, None -> word ^ cat ?sep words
```

When possible, default values can be defined, to get rid of `option` types.

```
let rec cat ?(sep = "") = function
  | [] -> ""
  | [ single ] -> single
  | word :: words -> word ^ sep ^ cat ~sep words
```

Syntax:

- `~arg:val` passes an `'a` value in `var` to an optional argument `?arg:'a`.
- `?arg:val` optionally passes an `'a option` value to an argument `?arg:'a`.
- `~pre:pre` simplifies to `~arg`.
- `?arg:arg` simplifies to `?arg`.

End of application:

- Application starts once a non optional, non labeled argument is passed.
- Functions must have a non optional arguments after the optional ones.
- A placeholder `()` can be used.
  Example: `let translate ?x ?y ?z () = (* ... *)`.
  The body is executed once the `()` is passed.

Polymorphic Variants

Sum types with infered definition

Basic use case: introduce constructors without type definition.

```
let arrow
  : int -> [ `LEFT | `RIGHT | `BOTH ] -> string
  = fun len head ->
  match head with
  | `LEFT -> "<" ^  String.make len '-'
  | `RIGHT -> String.make len '-' ^ ">"
  | `BOTH -> "<" ^ String.make len '-' ^ ">"
```

Syntax:

- Constructors prefixed with a bqckquote `` `constructor ``.
- Type alias: `` [ `constructor of type | `constructor | ... ] ``

Constructor sharing

Two polymorphic variant types can define the same constructor.

```
let arrow
  : int -> [ `LEFT | `RIGHT | `BOTH ] -> string
  = fun len head ->
  match head with
  | `LEFT -> "<" ^  String.make len '-'
  | `RIGHT -> String.make len '-' ^ ">"
  | `BOTH -> "<" ^ String.make len '-' ^ ">"
let parse_arrow_direction
  : char -> [ `LEFT | `RIGHT | `KNEE ]
  = function
    | '<' -> `LEFT | '>' -> `RIGHT
    | _ -> `KNEE
let arrow_from_direction =
  match parse_arrow_direction '>' with
  | (`LEFT | `RIGHT) as dir -> arrow 20 dir
  | `KNEE -> failwith "in the knee"
```

The dir value is considered of both variant types.

Intuitively, a polymorphic variant constructor:

- Can be seen as a singleton type.
  `` `U `` is the only value of type `` [ `U ] ``.
- Can be included in bigger polymorphic variant types.
- `` [ `U | `V ] `` is the union of singletons `` [ `U ] `` and `` [ `V ] ``,

This generalizes to a partial order (subtyping relation) order on variant types.

- `` [ `U ] `` is included in `` [ `U | `V ] ``;
- `` [ `U | `V ] `` is included in `` [ `U | `V | `W ] ``;
- `` [ `U | `V ] `` and `` [ `V | `W ] `` are not comparable ; etc.

As with object, this is structural subtyping.

- Polymorphic variant types do not need names.
- Their type is their structure, not their name.

The type system handles these subtyping relations though row variables, as with objects.

In objects, the `..` only meant and possibly other method.
For polymorphic variant types, more notations are avaiable:

- `[ `U | `V | `W]`
  Reads: a type whose constructors are `` `U ``, `` `V `` and `` `W ``.
- `[> `U | `V | `W]`
  Reads: at least `` `U ``, `` `V `` and `` `W ``.
- `[< `U | `V | `W]`
  Reads: at most `` `U ``, `` `V `` and `` `W ``.
- `[< `U | `V | `W > `U | `V |]`
  Reads: `` `U ``, `` `V `` and optionally `` `W ``.

Intuition:

- The form `` [< `U | `V | `W ] `` is used for expected values
  Exemple: function argument

```
let f : [< `U | `V | `W ] list -> unit = fun l ->
  List.iter (function `U -> () | `V -> () | `W -> ()) l
let () = f ([ `U ; `U ] : [ `U ] list)
```

  - The value passed may be always `` `U ``;
  - but it may never be `` `Z ``.
  - This is called a covariant position.
- The form `` [> `U | `V | `W ] `` is used for result values,
  Exemple: direct value definition

```
let l : [> `U | `W ] = [ `U ; `W ]
List.iter (function `U -> () | `V -> () | `W -> ()) l
```

  - Not treating a value `` `U ``, `` `V `` or `` `Z `` would cause an error ;
  - you can try and treat any other constructor safely.
  - This is called a contravariant position.

OCaml always infers the most generic type,
including row variables and their variance.

This is easily seen with pattern matching.

```
let f = function `X -> `A | `Y -> `B
```

Is infered to be of type: `[< `X | `Y ] -> [> `A | `B ]`.

If the type is too generic, one can refine it with:

- a (checked) annotation:

```
let f : [ `X | `Y ] -> [ `A | `B ]
  = function `X -> `A | `Y -> `B
```

- an interface:

```
module M
 : sig val f : [ `X ] -> [ `A | `B ] end
 = struct let f = function `X -> `A | `Y -> `B end
```

  For instance if `` `Y `` is reserved for module internal use.

- or a coercion:

```
let u = (`U :> [ `U | `V ])
```

Aliases

Polymorphic variant types can be named:

```
type xyz = [ `X | `Y | `Z ]
```

These names can be used:

- As any other type alias.
- Prefixed with a hash to open the type:
  `#xyz` means `[> `X | `Y | `Z]`
- In other variant definitions.

```
type wxyz = [ `W | xyz ]
```

- In match cases, as a shorcut for all constructors.

```
match v with #xyz -> ()
```

Patterns

Suppose we have an expression type with operators over bools and ints.

```
type expr =
  [ `And of bool * bool
  | `Not of bool
  | `Add of int * int
  | `Neg of int ]
```

Split & join cases 2 / 3

We can write two dedicated evaluation functions for clarity:

```
let eval_bool_op = function
  | `And (x, y) -> x && y
  | `Not x -> not x
let eval_int_op = function
  | `Add (x, y) -> x + y
  | `Neg x -> - x
```

Whose types are:

```
val eval_bool_op
  : [< `And of bool * bool | `Not of bool ] -> bool
val eval_int_op
  : [< `Add of int * int | `Neg of int ] -> int
```

Split & join cases

And then a join & split main evaluation function:

```
let eval_op = function
  | `And _ | `Not _ as bool_op ->
    `Bool (eval_bool_op bool_op)
  | `Add _ | `Neg _ as int_op ->
    `Int (eval_int_op int_op)
```

Whose type is:

```
val eval_op :
  [< `And of bool * bool | `Not of bool
  | `Add of int * int | `Neg of int ] ->
  [> `Bool of bool | `Int of int ] = <fun>
```

Objects, Labels & Variants

Object oriented APIS can be made more appealing with this combination.
One of the goal is to simulate constructor overloading.

Example:

```
class hbox :
  ?spacing:int ->
  ?homogeneous:bool ->
  ?vertical_align: [< `TOP | `BOTTOM | `CENTER | `FILL ]
  #component list -> component
```

Phantom types with variant tags

Phantom type parameters are ones that unused in the definition.
Using module abstraction, these variables can be forced.
This is used to encode human checked typing properties.

See `TyXML` that encodes XHTML well formedness this way.

Phantom types with variant tags 2 / 2

Example: a library for classifying ints.

```
module Checked_string : sig
  type +'a t
  val positive : int -> [> `P ] t
  val negative : int -> [> `N ] t
  val (mod) : [< `P ] t -> [< `P ] t -> [> `P ] t
  val abs : [< `P | `N ] t -> [> `P ] t
  val to_int : [< `P | `N ] t
end= struct
  type 'a t = int
  let positive n =
    if n < 0 then invalid_arg "positive" else n
  let negative n =
    if n > 0 then invalid_arg "negative" else n
  let (mod) x n = x mod n
  let abs n = abs n
  let to_int n = n
end
```

Property: once an integer classified, all operations on it won't fail.

Outline

Labeled & Optional arguments
Polymorphic Variants
Patterns