Part V

Appendices

A

Cyclic Types

Objective Caml's type system would be much simpler if the language were purely functionnal. Alas, language extensions entail extensions to the type language, and to the inference mechanism, of which we saw the illustration with the weak type variables (see page 74), made unavoidable by imperative extensions.

Object typing introduces the notion of *cyclic type*, associated with the keyword **as** (see page 454), which can be used independently of any concept of object oriented programming. The present appendix describes this extension of the type language, available through an option of the compiler.

Cyclic types

In Objective Caml, it is possible to declare recursive data structures: such a structure may contain a value with precisely the same structure.

```
# type sum_ex1 = Ctor of sum_ex1 ;;
type sum_ex1 = | Ctor of sum_ex1

# type record_ex1 = { field : record_ex1 } ;;
type record_ex1 = { field: record_ex1 }
```

How to build values with such types is not obvious, since we need a value before building one! The recursive declaration of values allows to get out of this vicious circle.

```
# let rec sum_val = Ctor sum_val ;;
val sum_val : sum_ex1 = Ctor (Ctor (Ctor (Ctor (Ctor ...))))

# let rec val_record_1 = { field = val_record_2 }
```

```
  and     val_record_2 = { field = val_record_1 } ;;
val val_record_1 : record_ex1 = {field={field={field={field={field=...}}}}}
val val_record_2 : record_ex1 = {field={field={field={field={field=...}}}}}
```

Arbitrary planar trees can be represented by such a data structure.

```
# type 'a tree = Vertex of 'a * 'a tree list ;;
type 'a tree = | Vertex of 'a * 'a tree list
# let height_1 = Vertex (0,[]) ;;
val height_1 : int tree = Vertex (0, [])
# let height_2 = Vertex (0,[ Vertex (1,[]); Vertex (2,[]); Vertex (3,[]) ]) ;;
val height_2 : int tree =
  Vertex (0, [Vertex (1, []); Vertex (2, []); Vertex (3, [])])
# let height_3 = Vertex (0,[ height_2; height_1 ]) ;;
val height_3 : int tree =
  Vertex
   (0,
    [Vertex (0, [Vertex (...); Vertex (...); Vertex (...)]); Vertex (0, [])])

(* same with a record *)
# type 'a tree_rec = { label:'a ; sons:'a tree_rec list } ;;
type 'a tree_rec = { label: 'a; sons: 'a tree_rec list }
# let hgt_rec_1 = { label=0; sons=[] } ;;
val hgt_rec_1 : int tree_rec = {label=0; sons=[]}
# let hgt_rec_2 = { label=0; sons=[hgt_rec_1] } ;;
val hgt_rec_2 : int tree_rec = {label=0; sons=[{label=0; sons=[]}]}
```

We might think that an enumerated type with only one constructor is not useful, but by default, Objective Caml does not accept recursive type abbreviations.

```
# type 'a tree = 'a * 'a tree list ;;
Characters 7-34:
The type abbreviation tree is cyclic
```

We can define values with such a structure, but they do not have the same type.

```
# let tree_1 = (0,[]) ;;
val tree_1 : int * 'a list = 0, []
# let tree_2 = (0,[ (1,[]); (2,[]); (3,[]) ]) ;;
val tree_2 : int * (int * 'a list) list = 0, [1, []; 2, []; 3, []]
# let tree_3 = (0,[ tree_2; tree_1 ]) ;;
val tree_3 : int * (int * (int * 'a list) list) list =
  0, [0, [...; ...; ...]; 0, []]
```

In the same way, Objective Caml is not able to infer a type for a function whose argument is a value of this form.

```
# let max_list = List.fold_left max 0 ;;
val max_list : int list -> int = <fun>
```

```
# let rec height = function
      Vertex (_,[]) → 1
    | Vertex (_,sons) → 1 + (max_list (List.map height sons)) ;;
val height : 'a tree -> int = <fun>


# let rec height2 = function
      (_,[]) → 1
    | (_,sons) → 1 + (max_list (List.map height2 sons)) ;;
Characters 95-99:
This expression has type 'a list but is here used with type
  ('b * 'a list) list
```

The error message tells us that the function `height2` could be typed, if we had type equality between *'a* and *'b * 'a list*, and precisely this equality was denied to us in the declaration of the type abbreviation *tree*.

However, object typing allows to build values, whose type is *cyclic*. Let us consider the following function, and try to guess its type.

```
# let f x = x#copy = x ;;
```

The type of `x` is a class with method `copy`. The type of this method should be the same as that of `x`, since equality is tested between them. So, if *foo* is the type of `x`, it has the form: *< copy : foo ; .. >*. From what has been said above, the type of this function is cyclic, and it should be rejected; but it is not:

```
# let f x = x#copy = x ;;
val f : (< copy : 'a; .. > as 'a) -> bool = <fun>
```

Objective Caml does accept this function, and notes the type cyclicity using **as**, which identifies *'a* with a type containing *'a*.

In fact, the problems are the same, but by default, Objective Caml will not accept such types unless objects are concerned. The function `height` is typable if it gives a cyclicity on the type of an object.

```
# let rec height a = match a#sons with
      [] → 1
    | l → 1 + (max_list (List.map height l)) ;;
val height : (< sons : 'a list; .. > as 'a) -> int = <fun>
```

Option `-rectypes`

With a compiler option, we can avoid this restriction to objects in cyclic types.

```
$ ocamlc -rectypes  ...
$ ocamlopt -rectypes  ...
$ ocaml -rectypes
```

If we take up the above examples in a toplevel started with this option, here is what we get.

```
# type 'a tree = 'a * 'a tree list ;;
type 'a tree = 'a * 'a tree list

# let rec height = function
      (_,[]) → 1
    | (_,sons) → 1 + (max_list (List.map height sons)) ;;
val height : ('b * 'a list as 'a) -> int = <fun>
```

The values `tree_1`, `tree_2` and `tree_3` previously defined don't have the same type, but they all have a type compatible with that of `height`.

```
# height tree_1 ;;
- : int = 1
# height tree_2 ;;
- : int = 2
# height tree_3 ;;
- : int = 3
```

The keyword **as** belongs to the type language, and as such, it can be used in a type declaration.

Syntax : **type** *nom* **=** *typedef* **as** *'var* **;;**

We can use this syntax to define type *tree*.

```
# type 'a tree = ( 'a * 'vertex list ) as 'vertex ;;
type 'a tree = 'a * 'a tree list
```

Warning If this mode may be useful in some cases, it tends to accept the typing of too many values, giving them types that are not easy to read.

Without the option `-rectypes`, the function below would have been rejected by the typing system.

```
# let inclus l1 l2 =
    let rec mem x = function
        [] → false
      | a::l → (l=x) || (mem x a) (* an error on purpose: a and l inverted *)
    in List.for_all (fun x → mem x l2) l1  ;;
val inclus : ('a list as 'a) list list -> ('b list as 'b) -> bool = <fun>
```

Although a quick examination of the type allows to conclude to an error, we no longer have an error message to help us locating this error.