

# Chapter 6

## User-defined types

The user is allowed to define his/her own data types. With this facility, there is no need to encode the data structures that must be manipulated by a program into lists (as in Lisp) or into arrays (as in Fortran). Furthermore, early detection of type errors is enforced, since user-defined data types reflect precisely the needs of the algorithms.

Types are either:

- *product* types,
- or *sum* types.

We have already seen examples of both kinds of types: the `bool` and `list` types are sum types (they contain values with different shapes and are defined and matched using several alternatives). The cartesian product is an example of a product type: we know statically the shape of values belonging to cartesian products.

In this chapter, we will see how to define and use new types in Caml.

### 6.1 Product types

Product types are *finite labeled* products of types. They are a generalization of cartesian product. Elements of product types are called *records*.

#### 6.1.1 Defining product types

An example: suppose we want to define a data structure containing information about individuals. We could define:

```
#let jean=("Jean",23,"Student","Paris");;  
jean : string * int * string * string = "Jean", 23, "Student", "Paris"
```

and use pattern-matching to extract any particular information about the person `jean`. The problem with such usage of cartesian product is that a function `name_of` returning the name field of a value representing an individual would have the same type as the general first projection for 4-tuples (and indeed would be the same function). This type is not precise enough since it allows for the application of the function to any 4-uple, and not only to values such as `jean`.

Instead of using cartesian product, we define a `person` data type:

```
#type person =
# {Name:string; Age:int; Job:string; City:string};;
Type person defined.
```

The type `person` is the *product* of `string`, `int`, `string` and `string`. The field names provide type information and also documentation: it is much easier to understand data structures such as `jean` above than arbitrary tuples.

We have *labels* (i.e. `Name`, ...) to refer to components of the products. The order of appearance of the products components is not relevant: labels are sufficient to uniquely identify the components. The Caml system finds a canonical order on labels to represent and print record values. The order is always the order which appeared in the definition of the type.

We may now define the individual `jean` as:

```
#let jean = {Job="Student"; City="Paris";
#           Name="Jean"; Age=23};;
jean : person = {Name="Jean"; Age=23; Job="Student"; City="Paris"}
```

This example illustrates the fact that order of labels is not relevant.

### 6.1.2 Extracting products components

The canonical way of extracting product components is *pattern-matching*. Pattern-matching provides a way to mention the shape of values and to give (local) names to components of values. In the following example, we name `n` the numerical value contained in the field `Age` of the argument, and we choose to forget values contained in other fields (using the `_` character).

```
#let age_of = function
#   {Age=n; Name=_; Job=_; City=_} -> n;;
age_of : person -> int = <fun>

#age_of jean;;
- : int = 23
```

It is also possible to access the value of a single field, with the `.` (dot) operator:

```
#jean.Age;;
- : int = 23

#jean.Job;;
- : string = "Student"
```

Labels always refer to the most recent type definition: when two record types possess some common labels, then these labels always refer to the most recently defined type. When using modules (see section 11.2) this problem arises for types defined in the same module. For types defined in different modules, the full name of labels (i.e. with the name of the modules prepended) disambiguates such situations.

### 6.1.3 Parameterized product types

It is important to be able to define parameterized types in order to define *generic* data structures. The `list` type is parameterized, and this is the reason why we may build lists of any kind of values. If we want to define the cartesian product as a Caml type, we need type parameters because we want to be able to build cartesian product of *any* pair of types.

```
#type ('a,'b) pair = {Fst:'a; Snd:'b};;
Type pair defined.

#let first x = x.Fst and second x = x.Snd;;
first : ('a, 'b) pair -> 'a = <fun>
second : ('a, 'b) pair -> 'b = <fun>

#let p={Snd=true; Fst=1+2};;
p : (int, bool) pair = {Fst=3; Snd=true}

#first(p);;
- : int = 3
```

Warning: the `pair` type is similar to the Caml cartesian product `*`, but it is a different type.

```
#fst p;;
Toplevel input:
>fst p;;
> ^
This expression has type (int, bool) pair,
but is used with type 'a * 'b.
```

Actually, any two type definitions produce different types. If we enter again the previous definition:

```
#type ('a,'b) pair = {Fst:'a; Snd:'b};;
Type pair defined.
```

we cannot any longer extract the `Fst` component of `x`:

```
#p.Fst;;
Toplevel input:
>p.Fst;;
> ^
This expression has type (int, bool) pair,
but is used with type ('a, 'b) pair.
```

since the label `Fst` refers to the *latter* type `pair` that we defined, while `p`'s type is the *former* `pair`.

## 6.2 Sum types

A *sum* type is the *finite labeled* disjoint union of several types. A sum type definition defines a type as being the union of some other types.

### 6.2.1 Defining sum types

Example: we want to have a type called `identification` whose values can be:

- either strings (name of an individual),
- or integers (encoding of social security number as a pair of integers).

We then need a type containing *both* `int * int` and character strings. We also want to preserve static type-checking, we thus want to be able to distinguish pairs from character strings even if they are injected in order to form a single type.

Here is what we would do:

```
#type identification = Name of string
#                       | SS of int * int;;
Type identification defined.
```

The type `identification` is the labeled disjoint union of `string` and `int * int`. The labels `Name` and `SS` are *injections*. They respectively inject `int * int` and `string` into a single type `identification`.

Let us use these injections in order to build new values:

```
#let id1 = Name "Jean";;
id1 : identification = Name "Jean"

#let id2 = SS (1670728,280305);;
id2 : identification = SS (1670728, 280305)
```

Values `id1` and `id2` belong to the same type. They may for example be put into lists as in:

```
#[id1;id2];;
- : identification list = [Name "Jean"; SS (1670728, 280305)]
```

Injections may possess one argument (as in the example above), or none. The latter case corresponds<sup>1</sup> to *enumerated types*, as in Pascal. An example of enumerated type is:

```
#type suit = Heart
#           | Diamond
#           | Club
#           | Spade;;
Type suit defined.

#Club;;
- : suit = Club
```

The type `suit` contains only 4 distinct elements. Let us continue this example by defining a type for cards.

---

<sup>1</sup>In Caml Light, there is no implicit order on values of sum types.

```
#type card = Ace of suit
#           | King of suit
#           | Queen of suit
#           | Jack of suit
#           | Plain of suit * int;;
Type card defined.
```

The type `card` is the disjoint union of:

- `suit` under the injection `Ace`,
- `suit` under the injection `King`,
- `suit` under the injection `Queen`,
- `suit` under the injection `Jack`,
- the product of `int` and `suit` under the injection `Plain`.

Let us build a list of cards:

```
#let figures_of c = [Ace c; King c; Queen c; Jack c]
#and small_cards_of c =
#  map (function n -> Plain(c,n)) [7;8;9;10];;
figures_of : suit -> card list = <fun>
small_cards_of : suit -> card list = <fun>

#figures_of Heart;;
- : card list = [Ace Heart; King Heart; Queen Heart; Jack Heart]

#small_cards_of Spade;;
- : card list =
  [Plain (Spade, 7); Plain (Spade, 8); Plain (Spade, 9); Plain (Spade, 10)]
```

### 6.2.2 Extracting sum components

Of course, pattern-matching is used to extract sum components. In case of sum types, pattern-matching uses dynamic tests for this extraction. The next example defines a function `color` returning the name of the color of the suit argument:

```
#let color = function Diamond -> "red"
#           | Heart -> "red"
#           | _ -> "black";;
color : suit -> string = <fun>
```

Let us count the values of cards (assume we are playing “belote”):

```
#let count trump = function
#  Ace _ -> 11
#  | King _ -> 4
```

```
# | Queen _      -> 3
# | Jack c       -> if c = trump then 20 else 2
# | Plain (c,10) -> 10
# | Plain (c,9)  -> if c = trump then 14 else 0
# | _           -> 0;;
count : suit -> card -> int = <fun>
```

### 6.2.3 Recursive types

Some types possess a naturally recursive structure, lists, for example. It is also the case for tree-like structures, since trees have subtrees that are trees themselves.

Let us define a type for abstract syntax trees of a simple arithmetic language<sup>2</sup>. An arithmetic expression will be either a numeric constant, or a variable, or the addition, multiplication, difference, or division of two expressions.

```
#type arith_expr = Const of int
#                | Var of string
#                | Plus of args
#                | Mult of args
#                | Minus of args
#                | Div of args
#and args = {Arg1:arith_expr; Arg2:arith_expr};;
Type arith_expr defined.
Type args defined.
```

The two types `arith_expr` and `args` are simultaneously defined, and `arith_expr` is recursive since its definition refers to `args` which itself refers to `arith_expr`. As an example, one could represent the abstract syntax tree of the arithmetic expression “`x+(3*y)`” as the Caml value:

```
#Plus {Arg1=Var "x";
#      Arg2=Mult{Arg1=Const 3; Arg2=Var "y"}};;
- : arith_expr = Plus {Arg1=Var "x"; Arg2=Mult {Arg1=Const 3; Arg2=Var "y"}}
```

The recursive definition of types may lead to types such that it is hard or impossible to build values of these types. For example:

```
#type stupid = {Next:stupid};;
Type stupid defined.
```

Elements of this type are *infinite* data structures. Essentially, the only way to construct one is:

```
#let rec stupid_value = {Next=stupid_value};;
stupid_value : stupid =
  {Next=
    {Next=
```

---

<sup>2</sup>Syntax trees are said to be *abstract* because they are pieces of *abstract syntax* contrasting with *concrete syntax* which is the “string” form of programs: analyzing (parsing) concrete syntax usually produces abstract syntax.

```

    {Next=
      {Next=
        {Next=
          {Next=
            {Next=
              {Next=
                {Next={Next={Next={Next={Next={Next={Next={Next=.}}}}}}}}}}}}
  }}

```

Recursive type definitions should be *well-founded* (i.e. possess a non-recursive case, or *base case*) in order to work well with call-by-value.

### 6.2.4 Parameterized sum types

Sum types may also be parameterized. Here is the definition of a type isomorphic to the `list` type:

```

#type 'a sequence = Empty
#           | Sequence of 'a * 'a sequence;;
Type sequence defined.

```

A more sophisticated example is the type of generic binary trees:

```

#type ('a,'b) btree = Leaf of 'b
#           | Btree of ('a,'b) node
#and ('a,'b) node = {Op:'a;
#           Son1: ('a,'b) btree;
#           Son2: ('a,'b) btree};;
Type btree defined.
Type node defined.

```

A binary tree is either a leaf (holding values of type `'b`) or a node composed of an operator (of type `'a`) and two sons, both of them being binary trees.

Binary trees can also be used to represent abstract trees for arithmetic expressions (with only binary operators and only one kind of leaves). The abstract syntax tree `t` of “1+2” could be defined as:

```

#let t = Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2};;
t : (string, int) btree = Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2}

```

Finally, it is time to notice that pattern-matching is not restricted to function bodies, i.e. constructs such as:

$$\begin{array}{l}
 \text{function } P_1 \quad \rightarrow E_1 \\
 | \quad \dots \\
 | \quad P_n \quad \rightarrow E_n
 \end{array}$$

but there is also a construct dedicated to pattern-matching of actual values:

$$\begin{array}{l}
 \text{match } E \text{ with } P_1 \quad \rightarrow E_1 \\
 | \quad \dots \\
 | \quad P_n \quad \rightarrow E_n
 \end{array}$$

which matches the value of the expression  $E$  against each of the patterns  $P_i$ , selecting the first one that matches, and giving control to the corresponding expression. For example, we can match the tree  $t$  previously defined by:

```
#match t with Btree{Op=x; Son1=_; Son2=_} -> x
#           | Leaf l -> "No operator";;
- : string = "+"
```

### 6.2.5 Data constructors and functions

One may ask: “What is the difference between a sum data constructor and a function?”. At first sight, they look very similar. We assimilate constant data constructors (such as `Heart`) to constants. Similarly, in Caml Light, sum data constructors with arguments also possess a functional type:

```
#Ace;;
- : suit -> card = <fun>
```

However, a data constructor possesses particular properties that a general function does not possess, and it is interesting to understand these differences. From the mathematical point of view, a sum data constructor is known to be an *injection* while a Caml function is a general function without further information. A mathematical injection  $f : A \rightarrow B$  admits an inverse function  $f^{-1}$  from its image  $f(A) \subset B$  to  $A$ .

From the examples above, if we consider the `King` constructor, then:

```
#let king c = King c;;
king : suit -> card = <fun>
```

`king` is the general function associated to the `King` constructor, and:

```
#function King c -> c;;
Toplevel input:
>function King c -> c;;
>~~~~~
Warning: this matching is not exhaustive.
- : card -> suit = <fun>
```

is the inverse function for `king`. It is a partial function, since pattern-matching may fail.

### 6.2.6 Degenerate cases: when sums meet products

What is the status of a sum type with a single case such as:

```
#type counter1 = Counter of int;;
Type counter1 defined.
```

Of course, the type `counter1` is isomorphic to `int`. The injection function `x -> Counter x` is a *total* function from `int` to `counter1`. It is thus a *bijection*.

Another way to define a type isomorphic to `int` would be:

```
#type counter2 = {Counter: int};;
Type counter2 defined.
```

The types `counter1` and `counter2` are isomorphic to `int`. They are at the same time sum and product types. Their pattern-matching does not perform any run-time test.

The possibility of defining arbitrary complex data types permits the easy manipulation of abstract syntax trees in Caml (such as the `arith_expr` type above). These abstract syntax trees are supposed to represent programs of a language (e.g. a language of arithmetic expressions). These kind of languages which are defined in Caml are called *object-languages* and Caml is said to be their *metalanguage*.

## 6.3 Summary

- New types can be introduced in Caml.
- Types may be *parameterized* by type variables. The syntax of type parameters is:

```
<params> ::
  | <tvar>
  | ( <tvar> [, <tvar>]* )
```

- Types can be *recursive*.
- Product types:
  - Mathematical product of several types.
  - The construct is:

```
type <params> <tname> =
  {<Field>: <type>; ...}
```

where the `<type>`s may contain type variables appearing in `<params>`.

- Sum types:
  - Mathematical disjoint union of several types.
  - The construct is:

```
type <params> <tname> =
  <Injection> [of <type>] | ...
```

where the `<type>`s may contain type variables appearing in `<params>`.

## Exercises

**Exercise 6.1** Define a function taking as argument a binary tree and returning a pair of lists: the first one contains all operators of the tree, the second one contains all its leaves.

**Exercise 6.2** Define a function `map_btree` analogous to the `map` function on lists. The function `map_btree` should take as arguments two functions `f` and `g`, and a binary tree. It should return a new binary tree whose leaves are the result of applying `f` to the leaves of the tree argument, and whose operators are the results of applying the `g` function to the operators of the argument.

**Exercise 6.3** We can associate to the `list` type definition an canonical iterator in the following way. We give a functional interpretation to the data constructors of the `list` type.

We change the list constructors `[]` and `::` respectively into a constant `a` and an operator  $\oplus$  (used as a prefix identifier), and abstract with respect to these two operators, obtaining the list iterator satisfying:

$$\begin{aligned} \text{list\_it } \oplus \text{ a } [] &= \text{a} \\ \text{list\_it } \oplus \text{ a } (x_1 :: \dots :: x_n :: []) &= x_1 \oplus (\dots \oplus (x_n \oplus \text{a}) \dots) \end{aligned}$$

Its Caml definition would be:

```
#let rec list_it f a =
#   function [] -> a
#   | x::l -> f x (list_it f a l);;
list_it : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

As an example, the application of `it_list` to the functional composition and to its neutral element (the identity function), computes the composition of lists of functions (try it!).

Define, using the same method, a canonical iterator over binary trees.

## **Part II**

# **Caml Light specifics**

