Chapter 7

Mutable data structures

The definition of a sum or product type may be annotated to allow physical (destructive) update on data structures of that type. This is the main feature of the imperative programming style. Writing values into memory locations is the fundamental mechanism of imperative languages such as Pascal. The Lisp language, while mostly functional, also provides the dangerous functions rplaca and rplacd to physically modify lists. Mutable structures are required to implement many efficient algorithms. They are also very convenient to represent the current state of a state machine.

7.1 User-defined mutable data structures

Assume we want to define a type person as in the previous chapter. Then, it seems natural to allow a person to change his/her age, job and the city that person lives in, but not his/her name. We can do this by annotating some labels in the type definition of person by the mutable keyword:

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

We can build values of type person in the very same way as before:

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

But now, the value jean may be physically modified in the fields specified to be mutable in the definition (and only in these fields).

We can modify the field Field of an expression <expr1> in order to assign it the value of <expr2> by using the following construct:

```ocaml
<expr1>.Field <- <expr2>
```

For example; if we want jean to become one year older, we would write:

```ocaml
#jean.Age <- jean.Age + 1;;
- : unit = ()
```
Now, the value `jean` has been modified into:

```ocaml
# jean;;
- : person = {Name="Jean"; Age=24; Job="Student"; City="Paris"}
```

We may try to change the `Name` of `jean`, but we won’t succeed: the typechecker will not allow us to do that.

```ocaml
# jean.Name <- "Paul";;
Toplevel input:
> jean.Name <- "Paul";;
>
The label Name is not mutable.
```

It is of course possible to use such constructs in functions as in:

```ocaml
# let get_older ({Age=n; _} as p) = p.Age <- n + 1;;
get_older : person -> unit = <fun>
```

In that example, we named `n` the current `Age` of the argument, but we also named `p` the argument. This is an *alias* pattern: it saves us the bother of writing:

```ocaml
# let get_older p =
#  match p with {Age=n} -> p.Age <- n + 1;;
get_older : person -> unit = <fun>
```

Notice that in the two previous expressions, we did not specify all fields of the record `p`. Other examples would be:

```ocaml
# let move p new_city = p.City <- new_city
# and change_job p j = p.Job <- j;;
move : person -> string -> unit = <fun>
change_job : person -> string -> unit = <fun>
```

```ocaml
# change_job jean "Teacher"; move jean "Cannes";;
# get_older jean; jean;;
- : person = {Name="Jean"; Age=25; Job="Teacher"; City="Cannes"}
```

We used the `;` character between the different changes we imposed to `jean`. This is the *sequencing* of evaluations: it permits to evaluate successively several expressions, discarding the result of each (except the last one). This construct becomes useful in the presence of *side-effects* such as physical modifications and input/output, since we want to explicitly specify the order in which they are performed.

### 7.2 The ref type

The *ref* type is the predefined type of mutable indirection cells. It is present in the Caml system for reasons of compatibility with earlier versions of Caml. The *ref* type could be defined as follows (we don’t use the *ref* name in the following definition because we want to preserve the original *ref* type):

```ocaml
let ref_type
```
#type 'a reference = {mutable Ref: 'a};;
Type reference defined.

Example of building a value of type ref:

#let r = ref (1+2);

r : int ref = ref 3

The ref identifier is syntactically presented as a sum data constructor. The definition of r should be read as “let r be a reference to the value of 1+2”. The value of r is nothing but a memory location whose contents can be overwritten.

We consult a reference (i.e. read its memory location) with the “!” symbol:

#!r + 1;;
- : int = 4

We modify values of type ref with the := infix function:

#r:=!r+1;;
- : unit = ()

#r;;
- : int ref = ref 4

Some primitives are attached to the ref type, for example:

#incr;;
- : int ref -> unit = <fun>
#decr;;
- : int ref -> unit = <fun>

which increments (resp. decrements) references on integers.

7.3 Arrays

Arrays are modifiable data structures. They belong to the parameterized type 'a vect. Array expressions are bracketed by [ | and | ], and elements are separated by semicolons:

#let a = [ | 10; 20; 30 | ];;
a : int vect = [ | 10; 20; 30 | ]

The length of an array is returned by with the function vect_length:

#vect_length a;;
- : int = 3
CHAPTER 7. MUTABLE DATA STRUCTURES

7.3.1 Accessing array elements

Accesses to array elements can be done using the following syntax:

```ocaml
# a.(0);;
- : int = 10
```
or, more generally: \( e_1.(e_2) \), where \( e_1 \) evaluates to an array and \( e_2 \) to an integer. Alternatively, the function `vect_item` is provided:

```ocaml
# vect_item;;
- : 'a vect -> int -> 'a = <fun>
```
The first element of an array is at index 0. Arrays are useful because accessing an element is done in constant time: an array is a contiguous fragment of memory, while accessing list elements takes linear time.

7.3.2 Modifying array elements

Modification of an array element is done with the construct:

```ocaml
e_1.(e_2) <- e_3
```
where \( e_3 \) has the same type as the elements of the array \( e_1 \). The expression \( e_2 \) computes the index at which the modification will occur.

As for accessing, a function for modifying array elements is also provided:

```ocaml
# vect_assign;;
- : 'a vect -> int -> 'a -> unit = <fun>
```

For example:

```ocaml
# a.(0) <- (a.(0)-1);;
- : unit = ()
# a;;
- : int vect = [\|9; 20; 30\|]
# vect_assign a 0 ((vect_item a 0) - 1);;
- : unit = ()
# a;;
- : int vect = [\|8; 20; 30\|]
```

7.4 Loops: while and for

Imperative programming (i.e. using side-effects such as physical modification of data structures) traditionally makes use of sequences and explicit loops. Sequencing evaluation in Caml Light is done by using the semicolon “;”. Evaluating expression \( e_1 \), discarding the value returned, and then evaluating \( e_2 \) is written:
7.4. **LOOPS: WHILE AND FOR**

If $e_1$ and $e_2$ perform side-effects, this construct ensures that they will be performed in the specified order (from left to right). In order to emphasize sequential side-effects, instead of using parentheses around sequences, one can use **begin** and **end**, as in:

```ocaml
#let x = ref 1 in
# begin
#  x := !x + 1;
#  x := !x * !x
# end;;
- : unit = ()
```

The keywords **begin** and **end** are equivalent to opening and closing parentheses. The program above could be written as:

```ocaml
#let x = ref 1 in
# (x := !x + 1; x := !x * !x);;
- : unit = ()
```

Explicit loops are not strictly necessary *per se*: a recursive function could perform the same work. However, the usage of an explicit loop locally emphasizes a more imperative style. Two loops are provided:

- **while**: `while $e_1$ do $e_2$ done` evaluates $e_1$ which must return a boolean expression, if $e_1$ return `true`, then $e_2$ (which is usually a sequence) is evaluated, then $e_1$ is evaluated again and so on until $e_1$ returns `false`.

- **for**: two variants, increasing and decreasing
  
  - `for $v$= $e_1$ to $e_2$ do $e_3$ done`
  
  - `for $v$= $e_1$ downto $e_2$ do $e_3$ done`

where $v$ is an identifier. Expressions $e_1$ and $e_2$ are the bounds of the loop: they must evaluate to integers. In the case of the increasing loop, the expressions $e_1$ and $e_2$ are evaluated producing values $n_1$ and $n_2$: if $n_1$ is strictly greater than $n_2$, then nothing is done. Otherwise, $e_3$ is evaluated $(n_2 - n_1) + 1$ times, with the variable $v$ bound successively to $n_1$, $n_1 + 1$, ..., $n_2$.

The behavior of the decreasing loop is similar: if $n_1$ is strictly smaller than $n_2$, then nothing is done. Otherwise, $e_3$ is evaluated $(n_1 - n_2) + 1$ times with $v$ bound to successive values decreasing from $n_1$ to $n_2$.

Both loops return the value `()` of type **unit**.

```ocaml
#for i=0 to (vect_length a) - 1 do a.(i) <- i done;;
- : unit = ()

#a;;
- : int vect = [|0; 1; 2|]
```
7.5 Polymorphism and mutable data structures

There are some restrictions concerning polymorphism and mutable data structures. One cannot enclose polymorphic objects inside mutable data structures.

```
#let r = ref [];;
r : '_a list ref = ref []
```

The reason is that once the type of `r`, `('a list) ref`, has been computed, it cannot be changed. But the value of `r` can be changed: we could write:

```
r:= [1; 2];;
```

and now, `r` would be a reference on a list of numbers while its type would still be `('a list) ref`, allowing us to write:

```
r := true::!r;;
```

making `r` a reference on `[true; 1; 2]`, which is an illegal Caml object.

Thus the Caml typechecker imposes that modifiable data structures appearing at toplevel must possess monomorphic types (i.e. not polymorphic).

Exercises

**Exercise 7.1** Give a mutable data type defining the Lisp type of lists and define the four functions `car`, `cdr`, `rplaca` and `rplacd`.

**Exercise 7.2** Define a `stamp` function, of type `unit -> int`, that returns a fresh integer each time it is called. That is, the first call returns 1; the second call returns 2; and so on.

**Exercise 7.3** Define a `quick_sort` function on arrays of floating point numbers following the quicksort algorithm [13]. Information about the quicksort algorithm can be found in [33], for example.